Mitigating Obfuscation Attacks on Software Plagiarism Detectors via Subsequence Merging

Timur Sağlam, Nils Niehues, Sebastian Hahner, Larissa Schmid
Karlsruhe Institute of Technology, Germany
firstname.lastname@kit.edu

***Abstract*—Plagiarism is a significant challenge in computer science education. Thus, tool-based approaches are widely used to combat software plagiarism. However, especially due to the recent rise of automated obfuscation via algorithmic or AI-based techniques, these tools face difficulties due to increasingly sophisticated obfuscation techniques. To address this challenge, we present a novel defense mechanism against automated obfuscation attacks. This mechanism iteratively merges matching program subsequences to counteract the effects of the obfuscation. Our approach is language-independent, attack-agnostic, and integrates well into state-of-the-art software plagiarism detectors. The evaluation based on five real-world datasets indicates that our approach not only provides broader resilience against algorithmic and AI-based obfuscation attacks than the state-of-the-art but also improves the detection of fully AI-generated programs.**

***Index Terms*—Software Plagiarism Detection, Source Code Plagiarism, Plagiarism Obfuscation, Obfuscation Attacks**

I. Introduction

Plagiarism poses a continuous challenge within computer science education [1–4]. Plagiarism in coding assignments is particularly pronounced in beginner-level and mandatory courses, such as introductory programming courses [5]. Due to the substantial size of these courses, solely relying on manual inspection is infeasible [6–8]. Moreover, plagiarizers use techniques like renaming, reordering, or inserting [9–11] to *obfuscate* their plagiarism or collusion. In light of these issues, it is common for educators to use software plagiarism detectors to uphold academic integrity for programming assignments [12].

Software plagiarism detectors analyze sets of programs to detect pairs with a suspiciously high degree of similarity [13, 14]. Ultimately, it is a human decision to assess which of them qualify as plagiarism, given the underlying ethical considerations [15]. Currently, MOSS [16] and JPlag [14, 17] are the detectors most widely used by educators [18]. An enduring assumption was that evading detection requires more time than it takes to complete the actual assignments and requires a profound understanding of programming languages [12, 19]. However, this assumption has been invalidated with the emergence of automated obfuscation attacks and plagiarism generators [20, 21]. While designing such an attack takes time and programming proficiency, using it as a student to plagiarize requires neither. While early automated attacks relied on algorithmic approaches, for example, via repeated modification as done by MOSSAD [12], the challenge intensifies with the rise of generative artificial intelligence, especially Large Language Models (LLMs) [22, 23], making the obfuscation of plagiarism more effortless than ever before [24, 25].

State-of-the-art plagiarism detectors compare the code's structure [9, 26] by parsing and linearizing the program's parse tree. Matching fragments are identified on pairs of these linearized representations, which are then used to compute a similarity score and derive suspicious candidates. By omitting details like names during the linearization, the detectors are resilient against specific *obfuscation attacks* [25] such as renaming, retyping, and other lexical changes [14, 19]. However, this inherent resilience does only apply to some (automated) obfuscation attacks [21, 27]. While there is some research to counteract such automated obfuscation attacks, these approaches face two crucial challenges [21]. First, *language-dependence*: Defense mechanisms are often highly language-specific, hindering straightforward generalization or transferability across different programming languages. Second, *attack-type-dependence*: Defense mechanisms are only tailored to one specific attack vector, thus lacking broad resilience. While they are highly efficient against the intended attacks, they provide little resilience for other attack types. Thus, they serve little protection against unknown attacks. Such emerging attacks, however, constitute the most challenging scenarios. However, with the recent rise of large language models and their *commoditization* via tools like ChatGPT [28], addressing emerging attacks is more crucial than ever [22].

Approach: Given these challenges, this paper investigates a novel approach called *subsequence match merging* (SMM) to bolster the obfuscation resilience of today's state-of-the-art software plagiarism detectors. All obfuscation attacks, whether known or unknown, have to disrupt the matching of code fragments in order to be effective [12]. Thus, they must affect the detectors' internal program representation [21]. To this end, obfuscation attacks try to alter the structural properties of a plagiarized program. Our approach iteratively merges neighboring fragment matches in pairs of linearized programs according to a well-designed heuristic until no more neighboring pairs remain. This process effectively reverses the effects of obfuscation, enhancing the detection of obfuscated plagiarism while minimizing false positives. As our approach operates solely on the internal linearized representations of programs, it is fully language-independent and not limited to a single obfuscation attack type. We thus provide a robust and versatile approach for a broad spectrum of known and unknown obfuscation attacks. Alongside our approach, we

TABLE I: Original code (left) and modified variant (right) after inserting one statement (+) and altering one (∼).

Original	→	Variant
`printNumbers(int max){`		`printNumbers(int max){`
`  int[] n = range(0,max);`		`  int[] n = range(0,max);`
`  String result = "";`		`  String result = "";`
	(+)	`  int debug = n.length;`
`  for(int i=0; i<max; i++)`	(∼)	`  for(int number : n)`
`    result += n[i];`		`    result += number;`
`  println(result);`		`  println(result);`
`}`		`}`

introduce a comprehensive threat model highlighting the danger of automated obfuscation attacks. We categorize different attack types and relate them regarding their effectiveness and applicability. By examining the attack surface of software plagiarism detectors, we show that all obfuscation attacks must affect the internal program representation to disrupt the detection process.

Evaluation: We evaluated our defense mechanism using the open-source plagiarism detector JPlag [14, 17, 29]. For our evaluation, we employed five real-world datasets and four types of automated obfuscation attacks: Statement insertion, statement alteration, AI-based obfuscation, and AI-based generation. In total, we thus evaluate 758 original and 747 obfuscated programs, encompassing 288,865 pairwise comparisons. Our results demonstrate that our approach increases the median similarity of plagiarism pairs by up to 44 percentage points while maintaining a minimal impact of around 4pp on the original and unrelated pairs, thus significantly improving resilience against algorithmic and AI-based obfuscation attacks. It is therefore evident that our approach not only provides broader resilience against obfuscation attacks than the state-of-the-art, but it also improves the detection of fully AI-generated programs by increasing similarity scores among these programs.

Contributions: In this paper, we present three contributions:

C1 A comprehensive threat model for (automated) obfuscation attacks targeting software plagiarism detection systems.

C2 Subsequence match merging (SMM), a novel language-independent and attack-type-independent defense mechanism against a wide range of obfuscation attacks.

C3 A four-stage evaluation using real-world data sets with automatically obfuscated plagiarism instances based on both algorithmic and AI-based obfuscation.

II. Running Example

We utilize the programs depicted in Table I as our illustrative example in the remainder of this paper. Both programs print the concatenated numbers from 1 to a specified maximum value. Notably, the program on the right is a modified variant of the one on the left, produced via two structural changes that avoid altering the program's behavior. Specifically, a new variable named `debug` is inserted, and the array-style loop is replaced with a for-each loop. Although the similarities between the two programs remain obvious for such a small example, this may not hold true when applying similar changes at a large scale. Crucially, such alterations reduce the likelihood of detection by a plagiarism detector.

TABLE II: The two token sequences corresponding to the programs in Table I with matching subsequences highlighted.

id	Original Tokens	→	id	Variant Tokens
1	`variable`		1	`variable`
2	`apply`		2	`apply`
1	`variable`		1	`variable`
		(+)	1	`variable`
3	`loop start`		3	`loop start`
1	`variable`		1	`variable`
4	`assign`	(∼)		
4	`assign`		4	`assign`
5	`loop end`		5	`loop end`
2	`apply`		2	`apply`

Table II illustrates the internal, linearized representations of both programs. For simplicity, only the representations of the contents of both methods are depicted (signatures are omitted). State-of-the-art plagiarism detectors linearize the programs by parsing them and extracting a subset of the parse tree nodes as *tokens* [21]. The resulting *token sequences* consist solely of structural elements [14]. Details like names, types, values and formatting are omitted, thus providing some obfuscation resilience. Due to the modifications made to generate the variant, the token sequences of both programs are not identical. We can identify three matching subsequences (highlighted in grey), with each neighboring match interrupted by a single token. Note, that the differing tokens interrupt the matching.

III. Threat Model

In the following, we introduce a comprehensive threat model as our first contribution (**C1**). *Obfuscation attacks* aim to avoid detection by strategically altering a plagiarized program, thus obscuring the relation to its original [21]. As state-of-the-art detection approaches compare the structure of programs by identifying similarities between code fragments [26], obfuscation attacks try to alter the structural properties of the program, ideally without affecting its behavior [10, 30, 31]. The intended outcome is to disrupt the matching of fragments between programs (as seen in Table II), thus leading to a reduced similarity score [12]. Specifically, the goal is to prevent the matching of fragments above the detector's specified match length cut-off threshold. However, to impact the detection quality of a software plagiarism detector, the obfuscation must affect the detectors' linearized program representation, which in the case of token-based approaches is the token sequence [21]. Consequentially, modifications to the program code that do not affect the token sequence are inherently ineffective. For example, renaming classes, members, and other elements does not affect token-based approaches, as names are omitted during the tokenization [14, 25].

A. Adversary Model

Students have access to peers' solutions, which they may use as a basis for plagiarism [12]. As instructors can access all solutions, students try to avoid detection by minimizing the similarity between the plagiarized program and the original while retaining program behavior.

We propose the following adversary model. The adversary is typically a student aiming to pass off the program of a third party as their own. This third party is typically another student who may or may not willingly participate. To evade detection, the adversary strategically obfuscates the plagiarized program. The program is thus the target of the adversary. The educator and the plagiarism detector serve as the defenders in trying to detect plagiarism. Crucially, the defender has little knowledge of obfuscation attempts. It is not clear which programs were plagiarized. Additionally, it is unclear what kinds of obfuscation techniques an adversary employs and to which parts of the program they were applied. Finally, the defender does not even know if an obfuscation attempt occurred. This uncertainty adds a layer of complexity to plagiarism detection, making it challenging to identify plagiarized programs [32]. As a result, prioritizing the obfuscation resilience of plagiarism detectors is more effective than attempting to identify obfuscation attempts explicitly.

B. Attack Surface Analysis

Obfuscation attacks affect the detection quality during the pairwise comparison by splitting up subsequences until they fall under the detector's matching threshold and are thus omitted. As token-based detectors solely use the internal representation of the programs for subsequence matching, the token-sequence, being that representation, is the only attack surface. Consequentially, for *any* obfuscation attack to be effective, it needs to affect the token sequence. Moreover, to interrupt the subsequence matching enough to reduce the calculated similarity, the attack must broadly affect the token sequence, meaning consistently over the entire sequence length. Note that while the token sequence is the attack surface, it cannot be directly altered. Instead, the program code needs to be altered to lead to a different token sequence. For this reason, simple techniques like renaming have no effect on token-based approaches. While they change the program, names are not considered during the tokenization and thus do not affect the token sequence.

Table II illustrates how the effect of the obfuscation attempts in Table I affect the subsequence sequence matching. While only two tokens are affected, the matching is disrupted, and only three subsequences are matched. For a hypothetical cut-off threshold of 4 tokens, all three would be ignored, thus reducing the similarity from 100% to 0%. Note how renaming variables in the original programs would not affect the internal representation of the derived token sequences, as this type of information is not present.

Each token essentially stores the syntactic category of its corresponding program element, such as control structures or variable definitions. By assigning a unique number to each category, the token sequence can be seen as a sequence of integers. Table II shows this integer representation. Based on this, we can derive three possible atomic types of alterations: Deleting a token from the sequence, inserting a token into the sequence, and changing the position of a token in the sequence. The effects of all possible obfuscation attacks, no matter how simple or complex, can be broken down into a combination of these atomic changes.

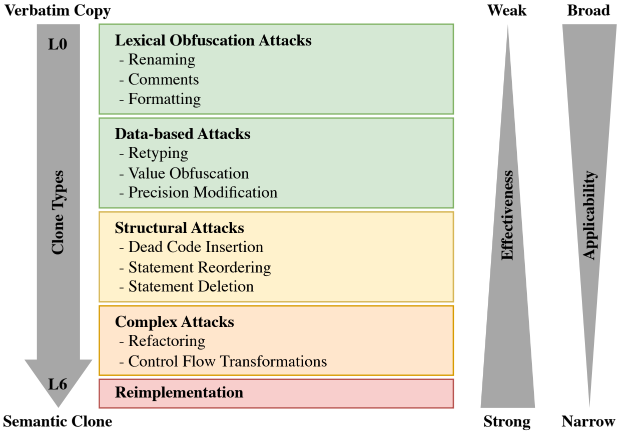

Fig. 1: Categorization of obfuscation attacks according to Karnalim [10] and Faidhi and Robinson [33], illustrating obfuscation effectiveness and applicability.

C. Categorization of Obfuscation Attacks

The challenge for an adversary is finding a combination of changes that maximize obfuscation but minimize the impact on program behavior. We classify potential obfuscation attacks on a conceptual scale based on Karnalim [10] and Faidhi and Robinson [33] from verbatim copying to semantic clones. The latter is inherently hard to distinguish from non-plagiarized programs. We define the following categories:

Lexical Attacks include renaming elements, changing comments, altering the code formatting, whitespace manipulation, and symbol substitution (e.g., parentheses and brackets).

Data-Based Attacks cover type changing data types, replacing literals with equivalent but differently represented values, and adjusting the precision of numeric values.

Structural Attacks involve fine-grained changes to the program structure, such as dead code insertion and the reordering of independent statements.

Complex Attacks include refactoring-based attacks like changing loop types, code (de-) fragmentation, inheritance hierarchy manipulation, and modifying the control flow structure.

Re-implementation includes both partial re-implementation, for example, replacing a sorting algorithm with another one, as well as full implementation, which produces semantic clones.

Note that these categories are not meant to be complete but rather showcase different manifestations of obfuscation attacks [9, 10]. In our example in Table I, statement insertion is a structural attack, while loop modification exemplifies a complex attack, albeit a simple instance.

Token-based approaches are inherently resilient to *lexical* obfuscation attacks [19], but an adversary may nevertheless employ them to alter the program's appearance to the human eye. Most approaches, provided they employ proper tokenization, are also resilient against data-based attacks. For instance, changing the name or value of the inserted debug statement in Table I would not impact the derived tokens in Table II. Figure 1 also illustrates the effect strength of these attack types. With rising complexity, the effect of the token sequence grows larger. However, with rising complexity, the applicability becomes less broad. For example, dead code insertion can be applied almost everywhere in a code base, while specific refactoring attacks can only be executed where the refactoring preconditions are met [21]. An important limitation of our threat model concerns semantic clones. Once a certain level of re-implementation is reached, the program code may look distinct but behave similarly. The challenge lies in accurately distinguishing these cases without erroneously flagging unrelated programs as false positives. It's worth noting that this limitation is intrinsic to plagiarism detection, whether conducted by humans or through automated tools.

D. Automation of Obfuscation Attacks

As discussed previously, automated obfuscation attacks refute the established assumption that evading detection requires more effort than completing the actual assignment [12, 19]. While designing such an automated attack is complicated, using it is not. A notable example of such automated attacks is MOSSAD [12], which repeatedly inserts (mostly dead) statements into a program to generate an obfuscated version. These statements are taken from the original program and a pool of predefined statements called *entropy*. As statements are selected randomly and inserted in random positions, this process is indeterministic and allows to generate multiple plagiarized programs from one original. For each iteration, MOSSAD checks if the code still compiles and checks on deviating behavior by comparing the assembly code of the compiled program. This process terminates when a chosen plagiarism detector computes a similarity between the original and obfuscated version of the program that is below a targeted threshold. In summary, this attack is algorithmic, indeterministic, and has a threshold-based termination criterion. Other attacks, in contrast, may be AI-based, e.g. via large language models, deterministic, or use an exhaustive termination criterion, e.g. insertion in every possible line in the code [21, 25]. This highlights the variety of potential automated attacks. Given this variety, defending against specific attack types alone is no longer sufficient.

Two key questions must be considered to assess any automated obfuscation attack: First, how effective are the underlying obfuscation attack types? Second, how easily can they be automated? Simpler, more fine-grained attacks, such as statement insertion, are easier to automate and can be applied more broadly across different sections of the code. However, combining different attack types is more effective in obfuscating a given program. In the past, complex attacks have been challenging to automate reliably due to their intricacy. However, with the rise of large language models, it is relatively easy to automatically apply various obfuscation attacks to a given program [23–25]. An additional consideration is semantics. Adversaries typically aim to avoid significantly altering program behavior to ensure their program still solves the given assignment. Consequently, the types of attacks are constrained, and the attack surface is limited. MOSSAD, for example, avoids inserting statements that change the program's behavior. However, a possible scenario is accepting a limited degree of deviating behavior to achieve stronger obfuscation. Considering the threat of detection, an adversary might accept not fully solving the assignment. In our work, we mainly consider semantic-preserving obfuscation attacks.

IV. SUBSEQUENCE MATCH MERGING

This section presents our main contribution: Subsequence match merging (**C2**). This approach heuristically searches among all matched subsequence pairs between two linearized programs to find *neighboring* matches that can be merged into a single one, subsuming the (unmatched) gap between them. This is done iteratively until no more neighboring pairs are found. This approach effectively reverts the effects of obfuscation attacks on the linearized program. As discussed in section III, all obfuscation attacks must affect the token sequence, thus interrupting the matching, to be effective. Our approach works solely on matching subsequences in the internal token-based representation, independent of the underlying programs. This makes it inherently language-independent. Furthermore, our approach relies only on the structural properties of all matched subsequences but does not consider the semantics of the corresponding tokens, thus making it attack-type-independent. Therefore, our approach provides resilience against all obfuscation attacks listed in Figure 1. The degree of resilience depends on the strength and type of the obfuscation attack.

As discussed, all token-based plagiarism detectors employ a cut-off threshold, below which matched code fragments are ignored to avoid false positives. This threshold, often called minimum match length (MML) [34], defines the minimal number of tokens required for two matching subsequences to be counted toward the similarity score. Thus, the MML controls the sensitivity of the comparison algorithm: if set too low, it increases the similarity but also the likelihood of false positives and vice versa. Obfuscation attacks affect the detection by splitting up matching subsequences of tokens until sufficiently many subsequences are small enough to fall below the MML and are thus ignored. Thus, our approach aims to reverse this by merging neighboring blocks of matches. The merging operates heuristically by assessing how close *neighboring* matches are, regardless of whether the alterations initially involved insertion, deletion, or reordering. By carefully selecting the matches to merge, we ensure a negligible impact on the false positive rate.

A. Neighboring Matches

A key concept in our approach is *neighborhood* of matches. We define matches as neighbors if they directly follow each other in the same order in both token sequences of a program pair. *Directly* refers to having no other matches in between on either side of the pair. This, however, does explicitly not include unmatched tokens. To illustrate, Figure 2 depicts matches between the token sequences of two programs. The matched subsequences are in the order `(A, B, C, D)` in the original program and `(B, A, C, D)` in the variant. Therefore, matches `A` and `B` would not be considered neighbors, as their subsequence order is inconsistent across both programs.

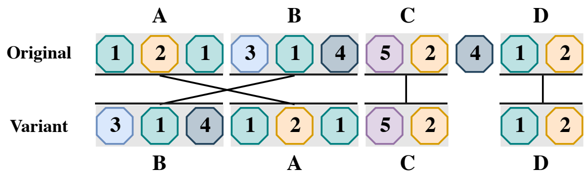

Fig. 2: Two tokenized programs with four subsequence matches (A-D), of which matches C and D are neighbors.

Matches A and C are not neighbors, as the corresponding subsequences in the original are interrupted by match B, which also is the case for B and C in the variant. In contrast, match C and match D are considered neighbors because they follow each other in the same order and are only separated by non-matching tokens.

Neighboring subsequence matches represent matching code fragments in the original programs. For instance, in Table I, lines 1-3 and 6-8 match in the original program and the variant. These lines appear in the same order and have no other matching segments in between, thus leading to a pair of neighboring matches in the token sequences. The non-identical statements that separate these neighboring matches are the inserted line 4 and the altered line 5.

B. Algorithm

Subsequence match merging operates as outlined in Algorithm 1. The algorithm takes the token sequences of two programs and their matching subsequences as input. Note that this includes all matches, including those who fall below the MML of the plagiarism detector. Initially, we compute which matching subsequences qualify as neighboring matches. If a pair of neighboring matches meets the merging criteria, we merge them and eliminate the gap in both token sequences. This involves effectively ignoring the unmatched tokens between the neighboring matches. After each merge, we recompute the neighbors and repeat this process until no more neighboring matches that fulfill the merging criteria are found. This means that merged matches can be merged with others in the following iterations. Our merging criteria are defined as follows:

Neighbor length: Both neighboring matches must exceed a specific length in tokens (e.g., matches with a length above two tokens).
Gap Size: The gap of unmatched tokens between these neighboring matches must not exceed a specific size (e.g., below six tokens).

For instance, as illustrated in Figure 3, the first pair of neighboring matches consists of two tokens in the first match and three in the second match. In the original program, no tokens separate these matches, whereas, in the variant, one token separates them. We merge pairs of neighboring matches when both have significant lengths and are separated by minimal tokens, indicating the pair represents a formerly single uninterrupted match. Our heuristic thus sets thresholds for minimum neighbor length and maximum gap size, merging matches that meet these criteria. Note that the neighbor length fulfills a purpose similar to the minimal match length. It controls the sensitivity of the approach.

Algorithm 1 Subsequence Match Merging

```
Require: $tokenSequences, matches$
$neighbors \leftarrow$ COMPUTENEIGHBORS($matches$)
repeat
    for each $neighborPair \in neighbors$ do
        if average size of $gapTokens \leq gapSizeThreshold$ then
            $mergedMatch \leftarrow$ MERGENEIGHBORS($neighborPair$)
            $matches \leftarrow matches \cup$ mergedMatch
            $matches \leftarrow matches \setminus$ neighborPair
        end if
    end for
    $neighbors \leftarrow$ COMPUTENEIGHBORS($matches$)
until no more valid merges
$prunedMatches \leftarrow$ PRUNEMATCHES($matches$)
return $prunedMatches$
```

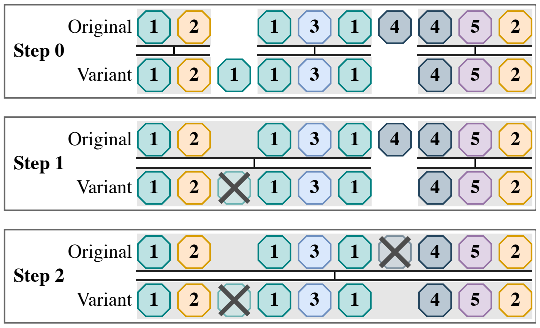

Fig. 3: Steps of the subsequence match merging for the running example in Table II (minimum match length = 4, minimum neighbor length = 2 and maximum gap-size = 1).

Finally, after merging, we need to filter any remaining matches that fall below the minimum match length to ensure that the resulting matches do not violate the assumptions of the plagiarism detector. This means some merged matches may not be included, as they may fall below the minimum match length threshold. By considering matches below the MML threshold, our approach can merge matches that the plagiarism detector would not have considered. Essentially, this allows the fragmentation of matches to be reversed, thus reverting the effects of obfuscation attacks.

Figure 3 illustrates the subsequence match merging algorithm for our running example in Table I. *Step 0* displays the token sequences of the original program and its obfuscated variant from Table II, along with the matching subsequences between them. For illustrative purposes, let the minimum match length threshold be four tokens, resulting in a similarity score of 0% because all three matches fall below this threshold. In *Step 1*, we consider the first pair of neighboring matches. After merging, the match length increases to 5, thus raising the similarity score to 58.8%. In *Step 2*, we assess the remaining pair of neighbors that also meet the threshold criteria. Merging them raises the similarity score to 100%, thus reverting the obfuscation demonstrated in the running example.

C. Hyperparameters

We intentionally avoided designing a defense mechanism that relies on too many hyperparameters, as it is always challenging for users to determine an optimal configuration in such cases [35, 36]. However, we provide two hyperparameters: the minimum neighbor length and the maximum gap size (see subsection IV-B). These can be adjusted to, for example, fit the algorithm to a specific dataset. Both thresholds are critical for tuning the heuristic's aggressiveness in deciding which neighboring matches to merge. Setting a low neighbor length and a high gap size threshold tends to merge unrelated matches, increasing false positives. Conversely, a high neighbor length and a low gap size threshold limit the effectiveness of our approach by failing to detect most instances of plagiarism. We conducted a grid search for a suitable default parameterization to address the trade-off between precision and recall. We explored neighbor length values from 1 to the minimum match length. The minimum match length was chosen as the upper threshold, as matches above this threshold are already detected. Furthermore, we explore gap size values from 1 to 20. We chose 20 as the upper threshold, representing a significant code fragment. During our grid search, this was confirmed, as the best results occur far below the threshold of 20. After evaluating each combination across various real-world datasets consisting of different assignment types, sizes, and different programming languages, as well as with varying obfuscation attacks, we observed that a minimum neighbor length of 2 and a maximum gap size of 6 yields the strongest resilience against obfuscation attacks. To recap, this means that neighboring matches are merged only if each match spans at least two tokens and six or fewer tokens separate them. Thus, we propose these values as default parameterization. However, we also recommend adjusting these hyperparameters for the dataset at hand when using our approach.

V. Evaluation

This section presents the evaluation of our defense mechanism (**C3**), where we use the state-of-the-art [9] plagiarism detector JPlag [14, 29] as the baseline. We employ five real-world datasets, totaling 758 original and 787 automatically obfuscated programs. This yields a total of 288,865 pairwise comparisons. We show that our approach provides resilience against automated obfuscation attacks. The code and the evaluation data are also provided in the supplementary material [37].

A. Methodology

The evaluation follows the *Goal-Question-Metric* (GQM) method [38, 39]. We define the following goals, evaluation questions, and metrics:

G1 Mitigating the impact of algorithmic obfuscation attacks.
- **Q1** Resilience to semantic-preserving attacks?
- **Q2** Resilience to (simulated) semantic-agnostic attacks?
- **M1/2** Similarity scores with SMM enabled vs. disabled.

G2 Mitigating the impact of AI-based obfuscation attacks.
- **Q3** Resilience to AI-based obfuscation?
- **Q4** Detection rate among AI-generated programs?
- **M3/4** Similarity scores with SMM enabled vs. disabled.

Both goals relate to the resilience that our approach, subsequence match merging (SMM), provides against automated obfuscation attacks as outlined in section III. The corresponding questions regard specific types of obfuscation attacks, thus evaluating the broadness and strength of the resilience. Software plagiarism detectors compare programs pairwise, thus computing pairwise similarity scores. To clearly distinguish the plagiarism instances from the unrelated programs during the human inspection, the similarity scores of plagiarism instances to their source must be high [21]. Vice versa, the similarity score for pairs or unrelated programs must be low. Thus, for all metrics, we evaluate whether the difference between plagiarism pairs and unrelated pairs increased when using our approach.

1) Datasets: For our evaluation, we employed five real-world datasets. They all come from an educational setting but stem from different courses and assignment types. Thus, they vary in the number and size of the programs, and especially in their programming language. First, we used two tasks from the publicly available collection *PROGpedia* [40]. Here, Task 19 covers the design of graph data structure and a depth-first search to analyze a social network. Task 56 is about minimum spanning trees using Prim's algorithm. Both datasets contain Java programs. Next, we used the *TicTacToe* dataset from [21], which contains command-line-based Java implementations of this paper-and-pencil game. Finally, we used two tasks from the publicly available homework dataset [41] by Ljubovic and Pajic [42]. While both tasks contain C++ programs, one pertains to managing student and laptop records within a university setting, whereas the other requires implementing a Fourier series. To prepare the datasets for our evaluation, we removed all solutions that did not compile, as JPlag requires valid input programs. We also removed all human plagiarism (if present) based on the labeling provided by the datasets. If no labeling was present, we removed verbatim copies. This notably reduces the size of some datasets. Consequently, we obtained the following five datasets, with size measured in lines of code (LOC), excluding comments and empty lines:

PROGpedia Task 19: 27 programs with a mean size of 131 LOC.
PROGpedia Task 56: 28 programs with a mean size of 85 LOC.
Homework Task 1: 59 programs with a mean size of 282 LOC.
Homework Task 5: 18 programs with a mean size of 123 LOC.
TicTacToe: 626 programs with a mean size of 236 LOC.

2) Obfuscation Attacks: We employ two algorithmic obfuscation attacks to evaluate the first goal (**G1**). The first is *PlagGen* [43], which inserts new, primarily dead, statements. Thus, it is similar to MOSSAD [12]. PlagGen is also semantic-preserving and indeterministic but uses an exhaustive termination strategy (see subsection III-D). The statement insertion uses statements from the original program and a pool of predefined statements. We use PlagGen for Java and MOSSAD for C++. Second, we simulate a semantic-agnostic obfuscation attack by randomly changing 25% of the tokens to a different one. While this attack is only simulated, it mirrors the effects of strong obfuscation attempts that alter a quarter of the program's code *without* considering program behavior. We include the results for other percentages in the supplementary material.

To evaluate the second goal (**G2**), we exploit OpenAI's GPT4 for automated plagiarism, which is currently the state-of-the-art LLM. There are generally two ways [25] of using generative AI to *cheat* for programming assignments: *AI-based*

TABLE III: Number of plagiarized programs per dataset and obfuscation attack type used in our evaluation (787 in total).

Obfuscation Attack Type	Pp-19	Pp-56	Hw-1	Hw-5	TT
Semantic Preserving Obf.	27	28	59	17	50
Semantic Agnostic Obf.	27	28	59	17	50
AI-based Obf. (15 Prompts)	75	75	75	75	75
AI-based Generation	-	-	-	-	50

obfuscation, where the adversary provides an AI model with a pre-existing program and tasks it to generate an obfuscated version. *AI-based generation*, where the adversary uses the assignment's description to generate a program from scratch via an AI model. We employ AI-based obfuscation as a third obfuscation attack alongside both algorithmic ones. We use different prompts, mimicking how students would ask GPT to obfuscate their plagiarism. Finally, we use full generation only for the TicTacToe dataset, as we require the full assignment description and test cases to test for the expected behavior. In sum, we use the following four techniques attacks to create 787 plagiarized programs (see Table III for details):

Semantic-Preserving Obfuscation: Inserting new and existing statements into the program (PlagGen [43] and MOSSAD [12]).
Semantic-Agnostic Obfuscation: Simulate changing large parts of the program by randomly changing 25% of the tokens.
AI-based Obfuscation: We obfuscate human solutions with GPT-4 based on 15 varying prompts requesting structural changes.
AI-based Generation: We fully generate AI-based solutions with GPT-4 based on the textual task description (TicTacToe only).

According to the threat categorization in Figure 1, semantic-preserving obfuscation using insertion is categorized as a structural attack. Semantic-agnostic obfuscation simulates structural and complex attacks but refactoring only to a lesser extent. AI-based obfuscation is multifaceted, mapping to both structural and complex attacks, especially including refactoring. AI-based generation is classified as full re-implementation, which is highly challenging to address. Our evaluation, therefore, covers a broad range of possible obfuscation attacks.

B. Evaluation Results

This section presents the evaluation results, demonstrating that our approach significantly improves resilience against all four obfuscation attack types. In the following, we will discuss the results of each obfuscation attack type individually.

1) Semantic-Preserving Obfuscation: Figure 4 shows the results for obfuscation attacks based on the insertion of statements into programs. Our approach shows strong improvements in resilience to these attacks. For all five datasets, our approach significantly increases the similarities of the plagiarism pairs. Specifically, the median similarities increase by between 16 percentage points (TicTacToe) and 39 percentage points (PROGpedia-56 and Homework-5). Regarding original pairs, the median similarities rise by 2 to 3 percentage points for each dataset, except for PROGpedia-56, where the median increases by around 9 percentage points. We hypothesize that this exception is attributable to the small program size of the PROGpedia-56 dataset. Overall, however, the similarity increase for the original pairs is relatively small and thus negligible. However, when looking at the separation of plagiarism and originals, we can see the similarity increases significantly more for the plagiarism pairs than the originals. In fact, the delta between the median similarities of both types of pairs increases by between 16 percentage points (TicTacToe) and 30 percentage points (PROGpedia-56). As previously mentioned, these values represent the improvement in separating plagiarized and original programs.

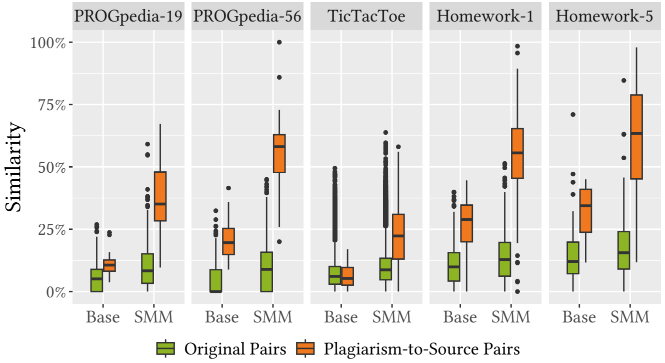

Fig. 4: Similarities for pairs of human programs and insertion-based plagiarism with and without our approach. Plagiarism pairs should be high, original pairs should be low.

Answer to Q1: *SMM significantly increases the resilience against semantic preserving obfuscation attacks, thus enabling separation between plagiarized and original programs.*

2) Semantic-Agnostic Obfuscation: Figure 5 shows the results for obfuscation attacks based on a simulated 25% change to the program. We include the results for 10% to 30% in the supplementary material. Our approach also shows strong resilience improvements for these attacks. Again, our approach significantly increases the similarities of the plagiarism pairs for all datasets. In detail, the median similarities increase by at least 30 percentage points (Homework-1) and up to 44 percentage points (PROGpedia-56). As the original pairs remain the same for all evaluation stages, the results still show a median similarity increase of 3 percentage points for all but one dataset. Again, regarding the separation of plagiarism and originals, we observe a notably greater increase in similarity for the plagiarism pairs compared to the original ones. Here, the delta between the median similarities of both types of pairs increases by at least 26 percentage points (TicTacToe) and up to 39 percentage points (PROGpedia-56). Consequently, the interquartile ranges of plagiarism and original pairs no longer overlap. This means that our approach facilitates detection despite randomly changing 25% of the linearized program.

Answer to Q2: *SMM significantly increases the resilience against semantic agnostic obfuscation attacks, thus clearly separating plagiarized and original programs.*

3) AI-based Obfuscation: Figure 7 shows the results for obfuscation attacks via GPT-4 prompts. We used 15 different

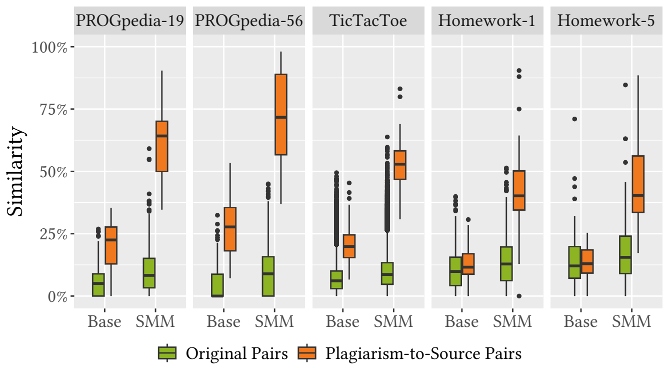

Fig. 5: Similarities for human programs and alteration-based plagiarism (25% of tokens) with and without SMM.

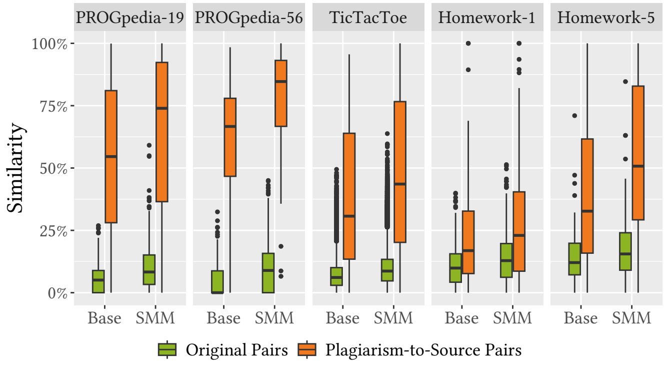

Fig. 6: Similarities for human programs and GPT-obfuscated plagiarism pairs with and without our approach.

prompts, asking the LLM to alter the program without changing its behavior. The results show that our approach improves resilience to these obfuscation attacks. For the plagiarism pairs, our approach increased median similarities by at least 6 percentage points (Homework-1) and up to 19 percentage points (PROGpedia-56). Regarding the separation of plagiarism and originals, the delta between the median similarities of both types of pairs increases by between 9 percentage points (PROGpedia-56) and 16 percentage points (PROGpedia-19) for all datasets, except for Homework-1. For this dataset, the median similarity delta only increases by 3 percentage points. This increase is less prominent than that observed with algorithmic attacks, which can be attributed to two factors. Firstly, the overall varying effectiveness of AI-based obfuscation. For plagiarized programs already exhibiting high similarity to their originals, there remains limited enhancement potential. Secondly, the broader range of modifications produced by generative AI. These diverse modifications alter token sequences more extensively, thereby limiting the effectiveness of subsequence merging. Notably, despite both factors, we observe an improvement in resilience against AI-based obfuscation.

Answer to Q3: *While the effectiveness of AI-based obfuscation depends on the dataset, SMM increases the resilience against these attacks, albeit to a lesser degree.*

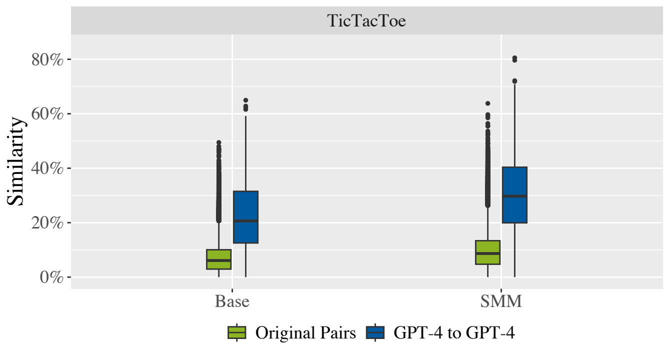

Fig. 7: Similarity values for pairs of human programs and GPT-generated programs with and without SMM.

4) AI-generated Programs: Figure 7 shows the results for programs generated via GPT-4 based on the assignment description. Unlike in the previous three evaluation stages, we technically do not employ obfuscation attacks. The generated programs are not derived from human-made programs. Thus, our comparison focuses solely on how the similarity among generated programs differs from that of unrelated human programs. Without SMM, the median similarity of pairs of unrelated human programs is 6%, while it is 21% for AI-generated programs. This means that AI-generated programs show *higher* inherent similarities than human solutions due to the deterministic nature of generative AI. With SMM enabled, the median similarity of the AI-generated programs increases by around 9 percentage points to 30%. In contrast, the similarities of unrelated human programs only change very little. This shows that our approach improves the detection of fully AI-generated programs by increasing the similarity values calculated for them while only minimally affecting pairs of unrelated human-made programs. These results are remarkable given that our approach is not designed to detect fully AI-generated programs. The fact that it improves detection for them highlights its versatility.

Answer to Q4: *SMM increases the separation between human and AI-generated programs, thus improving the detection of AI-generated programs.*

C. Statistical and Practical Significance

To test the significance of our results, we conducted statistical tests for all evaluation stages for JPlag with and without our approach. The detailed results are depicted in Table IV. Regarding statistical significance, the results show strong statistical significance for all evaluation stages with p-values at or below 2.133e-07. The effect sizes (0.645 and 0.929) are large for algorithmic obfuscation attacks and small to medium (0.214 and 0.320) for AI-based methods. The smaller effect size for AI-based obfuscation is caused by the high similarity variance inherent to this obfuscation method, as different prompts vary strongly in effectiveness. The results also show practical significance, as the effect sizes demonstrate the robustness of our approach in real-world scenarios. This is confirmed by the increased separation between plagiarism pairs and original pairs, aiding with distinguishing between plagiarized and original solutions.

TABLE IV: One-sided Wilcoxon rank-sum tests results (sig. level of $\alpha = 0.01$, alternative hypothesis $H1 = greater$, test statistic W, Cliff's delta δ, the 95 percent confidence interval CI, sample size n, variance σ^2) for Plagiarism-to-Source (P2S), Original Pairs (OP), and Fully Generated Pairs (FG).

Obfuscation	Pairs	p	W	δ	95% CI	n	σ^2
Semantic-Pres.	P2S	<2.2e-16	27,250	0.645	[0.55, 0.72]	364	5.09
	OP	<2.2e-16	2.4436e+10	0.243	[0.24, 0.25]	396,436	0.39
Semantic-Agn.	P2S	<2.2e-16	31,954	0.929	[0.88, 0.96]	364	5.06
	OP	<2.2e-16	2.4427e+10	0.243	[0.24, 0.25]	396,436	0.39
AI-based Obf.	P2S	2.133e-07	84,444	0.214	[0.13, 0.29]	746	10.12
	OP	<2.2e-16	2.4451e+10	0.243	[0.24, 0.25]	396,436	0.39
AI-based Gen.	FG	<2.2e-16	990,448	0.320	[0.28, 0.36]	2,450	2.02
	OP	<2.2e-16	2.3833e+10	0.246	[0.24, 0.25]	391,250	0.38

D. Discussion

The evaluation results reveal several noteworthy insights, both regarding automated obfuscation attacks and the effectiveness of our subsequence match merging approach.

1) AI-based Obfuscation Attacks: In our observation, the effectiveness of AI-based obfuscation attacks strongly varies depending on the dataset used. As illustrated in Figure 6, the plagiarized programs mostly stay above 65% for the PROGpedia-19 dataset, while most of them fall below 50% for the Homework-5 dataset. Median similarities vary between 17% and 76% for these programs. Yet, we use the same obfuscation prompts for both datasets. Moreover, compared to algorithmic obfuscation, the range of similarities per dataset varies more notably for AI-based obfuscation. For the former, all interquartile ranges of plagiarism pairs are below 30 percentage points. For the latter, we observe interquartile ranges that span more than 50 percentage points (PROGpedia-19). Finally, in some cases, GPT produces incomplete or invalid code. Sometimes, the programs did not compile, thus requiring re-generation. Despite over 50 attempts, we could not produce a valid result for three original programs, which all exceeded 300 LOC. Thus, algorithmic obfuscation *currently* exhibits more consistent results than AI-based obfuscation and *currently* seems more effective. However, AI-based obfuscation is more useful in avoiding detection during manual inspection, as it produces a diverse range of modifications.

While full AI-based generation works to a certain extent, *currently* its effectiveness is limited. The programs entirely generated with GPT did not precisely adhere to the assignment's requirements, which led to additional output or slightly altered behavior. These discrepancies suggest that full generation may only be suitable for very small assignments. In our case, the TicTacToe dataset is both in complexity and size (∼236 LOC) at the perceived threshold, at which fully generated solutions start to show such discrepancies. Interestingly, even without applying our approach, AI-generated programs display a higher degree of similarity to each other than human-generated solutions. This can be attributed to the inherent degree of determinism within Large Language Models. While they are not entirely deterministic, their level of determinism is sufficient for software plagiarism detection purposes.

2) Resilience via Subsequence Match Merging: The evaluation shows that our approach provides obfuscation resilience for all employed attacks and datasets. Thus, we demonstrate that our approach's effectiveness is not limited to a specific set of obfuscation attacks. It has exhibited effectiveness against a diverse range of obfuscation attacks, including both algorithmic and AI-based attacks, encompassing semantic-preserving and semantic-agnostic alterations. Furthermore, we evaluated it across datasets in different programming languages, in addition to diverse assignment types and sizes, thus demonstrating its adaptability. This underscores our approach's broad resilience and language independence, positioning it as a versatile defense mechanism against automated obfuscation attacks on software plagiarism detectors.

While attack-specific defense mechanisms may achieve even better results for their targeted obfuscation attacks than attack-independent approaches, they fall short for other attack types. For example, an approach based on dead code detection may outperform ours for obfuscation via dead statement insertion; however, it will have no effect on refactoring-based obfuscation, as GPT employs. Moreover, attack-specific defense mechanisms can only be designed for *known* obfuscation attacks and are thus not suitable for emerging threats. Attack-independent approaches, like ours, are crucial for emerging threats. As they operate only at the level of the actual attack surface, the linearized program representation, and do not make assumptions about incoming attacks, they allow resilience against unknown attacks. As our approach can be combined with any other defense mechanism, we ultimately argue that layering defense mechanisms is the safest strategy (*swiss cheese model* [44] or *defense in depth* [45–47]). Layering allows combining the broad resilience of attack-independent approaches with the targeted effectiveness of attack-specific ones.

E. Threats to Validity

We now discuss how we address threats to the validity of our work as outlined by Wohlin *et al.* [48], as well as Runeson and Höst [49]. *Internal Validity:* For internal validity, we evaluated JPlag both with and without our approach, ensuring that all other conditions remained constant. Furthermore, for the automated obfuscation, we randomly selected programs from the respective datasets. To address the impact of prompt choice on AI-based obfuscation, we performed systematic "prompt engineering" prior to the evaluation. We then evaluated with 15 different prompts. *External Validity:* To ensure external validity, we employ five real-world datasets. These datasets are diverse, encompassing two programming languages and varying in the number of programs, program sizes, assignment types, and complexity levels. This diversity underscores the generalizability of our approach across different contexts. Moreover, our approach is tool-independent, allowing it to be implemented in any token-based and most structure-based plagiarism detectors, further enhancing its applicability. *Construct Validity:* To ensure construct validity, we aligned our evaluation methodology with those from related works [15, 21, 34]. Furthermore, we have accurately labeled datasets; as for automated obfuscation, we know which programs are plagiarized. Finally, we employ an approach-independent ground truth, use established similarity metrics, and a GQM plan [38, 39]. *Reliability:* For reliability, we provide our full code and data (see section VIII). Moreover, our approach has been incorporated into the widely used plagiarism detector JPlag [29].

F. Limitations

While our approach addresses current generative AI threats, rapid advancements in this field may necessitate future re-evaluation. However, all emerging attacks must affect the same attack surface (see subsection III-B). Thus, we are confident our approach will remain effective for emerging attacks. Moreover, small assignments are a challenge in plagiarism detection, as minor similarities can lead to false positives, given their minimal solution space. This makes separating plagiarism from coincidental similarities difficult. However, this limitation is inherent to plagiarism detection [34].

VI. Related Work

A. Software Plagiarism Detection

Code plagiarism detectors focus on the structure of programs [9, 26]. They find matching fragments via hashing and tiling [16, 17]. Specifically, JPlag [14, 17] and Sherlock [19] use *greedy string tiling* [50, 51], whereas MOSS and Dolos [52] employ *winnowing* [53]. *Resilience via Graphs:* Liu *et al.* [54] introduce *GPlag*, a graph-based approach that compares graphs directly instead of linearizing the program. They provide resilience against *some* obfuscation attacks like dead code insertion. However, they are impractical due to their computational expense; determining subgraph isomorphism is NP-complete, and doing so for pairwise comparisons does not scale well. *Resilience via Intermediate Representation:* Devore-McDonald and Berger [12] propose using semantic checking of intermediate representation (assembly and Java byte code) to defend against MOSSAD. However, this language-dependent defense mechanism is conceptual and is yet to be realized. Similarly, Karnalim [10] introduces an approach based on Java byte code. They employ normalization, e.g., by linearizing method contents. This approach can handle some obfuscation attacks. Nevertheless, this excludes higher-level obfuscation, e.g., via refactoring. *Resilience via Preprocessing:* Krieg [55] provides a preprocessing approach that removes tokens to increase the similarity between programs. However, this token-level approach is limited in effectiveness and computationally expensive. Inspired by theirs, our approach gains effectiveness and scalability by operating on matched subsequences as a post-processing step. Novak [30] proposes preprocessing via *Common-Code-Deletion*. They remove meaningless statements, such as getters, setters, and empty methods. This improves insertion resilience in certain cases. Karnalim *et al.* [56] employ a similar strategy via multiple preprocessing techniques. While this is effective against tools like MOSSAD, it is ineffective for others. *Differentiation.* In contrast to all aforementioned approaches, our defense mechanism is both language-independent and attack-independent, as we operate on the abstract level of the matched subsequences. Thus, we achieve broad resilience. Our approach can be combined with others, for example, token sequence normalization [21], to increase obfuscation resilience. This combines broad resilience with narrow capabilities against specific attacks.

B. Software Clone Detection

Plagiarism detection relates strongly to clone detection; however, code clones typically arise inadvertently [57]. Consequentially, plagiarism detection approaches need to deal with an additional layer of complexity introduced by an adversary-defender-scenario [21]. Targeted obfuscation attacks thus make software plagiarism detection more challenging than clone detection [58]. Still, many clone detection approaches share similarities in their employed techniques [59–61].

C. Genome Sequencing

Indel detection in bioinformatics identifies altered genome sequence pairs. Similarly, plagiarism detection identifies program pairs despite minor changes. *TransIndel* by Yang *et al.* [62] compares genome sequences composed of *chimeric reads* and aligns them. It infers insertions or deletions by iterating through both sequences with a fixed window size, thus resembling SMM. The Needleman–Wunsch algorithm [63] computes the globally optimal alignment between sequences. While designed for gene sequencing, it could be applied to plagiarism detection. However, it comes with high algorithmic complexity.

VII. Conclusion

This paper addresses the challenge of emerging automated obfuscation attacks in software plagiarism detection. To this end, we introduce a novel approach called subsequence match merging to enhance the obfuscation resilience of plagiarism detectors. Using only the abstract program representation, our approach is language-independent and obfuscation-attack-agnostic, thus proving broader resilience than the state-of-the-art. We evaluate our approach on five real-world datasets. We observe an increase of up to 44 percentage points for plagiarism pairs while maintaining minimal impact (avg. of 4pp) on unrelated programs. The results show that our approach provides significant resilience against both algorithmic and AI-based obfuscation attacks. Moreover, our approach improves the detection of AI-generated programs.

VIII. Data Availability

Our approach has been incorporated into the widely used open-source plagiarism detector JPlag [29]. We also provide our code and evaluation data as supplementary material [37].

Acknowledgment

This work is supported by the pilot program Core Informatics of the Helmholtz Association (HGF), by the Topic Engineering Secure Systems of the HGF, the German Federal Ministry of Education and Research (BMBF) via grant number 16KISA086 (ANYMOS), and by KASTEL Security Research Labs, Karlsruhe.

REFERENCES

[1] G. Cosma and M. Joy, “Towards a definition of source-code plagiarism,” *IEEE Transactions on Education*, vol. 51, no. 2, pp. 195–200, May 2008. DOI: 10.1109/te.2007.906776.

[2] W. Murray, “Cheating in computer science,” *Ubiquity*, vol. 2010, p. 2, Jun. 2010. DOI: 10.1145/1865907.1865908.

[3] T. Le, A. Carbone, J. Sheard, M. Schuhmacher, M. de Raath, and C. Johnson, “Educating computer programming students about plagiarism through use of a code similarity detection tool,” in *2013 Learning and Teaching in Computing and Engineering*, IEEE, Mar. 2013, pp. 98–105. DOI: 10.1109/LaTiCE.2013.37.

[4] A. Sutton, D. Taylor, and C. Johnston, “A model for exploring student understandings of plagiarism,” *Journal of Further and Higher Education*, vol. 38, no. 1, pp. 129–146, Jan. 2014. DOI: 10.1080/0309877X.2012.706807.

[5] C. Park, “In other (people’s) words: Plagiarism by university students–literature and lessons,” *Assessment & Evaluation in Higher Education*, vol. 28, no. 5, pp. 471–488, Oct. 2003. DOI: 10.1080/02602930301677.

[6] T. Camp *et al.*, “Generation cs: The growth of computer science,” *ACM Inroads*, vol. 8, no. 2, pp. 44–50, May 2017. DOI: 10.1145/3084362.

[7] C. Kustanto and I. Liem, “Automatic source code plagiarism detection,” in *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, IEEE, May 2009, pp. 481–486. DOI: 10.1109/SNPD.2009.62.

[8] L. Yan, N. McKeown, M. Sahami, and C. Piech, “Tmoss: Using intermediate assignment work to understand excessive collaboration in large classes,” in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE ’18, ACM, Feb. 2018, pp. 110–115. DOI: 10.1145/3159450.3159490.

[9] M. Novak, M. Joy, and D. Kermek, “Source-code similarity detection and detection tools used in academia: A systematic review,” *ACM Transactions on Computing Education*, vol. 19, no. 3, pp. 1–37, Sep. 2019. DOI: 10.1145/3313290.

[10] O. Karnalim, “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, Oct. 2016, pp. 63–68. DOI: 10.1109/icts.2016.7910274.

[11] T. Sağlam, L. Schmid, S. Hahner, and E. Burger, “How students plagiarize modeling assignments,” in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, ser. MODELS ’23, IEEE, Oct. 2023, pp. 98–101. DOI: 10.1109/MODELS-C59198.2023.00032.

[12] B. Devore-McDonald and E. D. Berger, “Mossad: Defeating software plagiarism detection,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–28, Nov. 2020. DOI: 10.1145/3428206.

[13] K. J. Ottenstein, “An algorithmic approach to the detection and prevention of plagiarism,” *ACM SIGCSE Bulletin*, vol. 8, no. 4, pp. 30–41, Dec. 1976. DOI: 10.1145/382222.382462.

[14] L. Prechelt, G. Malpohl, and M. Philippsen, *JPlag: Finding plagiarisms among a set of programs*. 2000, Technical Report. DOI: 10.5445/ir/542000.

[15] T. Sağlam, S. Hahner, L. Schmid, and E. Burger, “Obfuscation-resilient software plagiarism detection with jplag,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, (Lisbon, Portugal, Apr. 14–20, 2024), ser. ICSE-Companion, Institute of Electrical and Electronics Engineers (IEEE), Apr. 2024, pp. 264–265. DOI: 10.1145/3639478.3643074.

[16] A. Aiken. “Moss software plagiarism detector website,” Stanford University. (Jul. 2022), (visited on 2022).

[17] L. Prechelt, G. Malpohl, and M. Philippsen, “Finding plagiarisms among a set of programs with jplag.,” *Journal of Universal Computer Science*, vol. 8, no. 11, p. 1016, 2002. DOI: 10.3217/jucs-008-11-1016.

[18] R. C. Aniceto, M. Holanda, C. Castanho, and D. Da Silva, “Source code plagiarism detection in an educational context: A literature mapping,” in *2021 IEEE Frontiers in Education Conference (FIE)*, IEEE, Oct. 2021, pp. 1–9. DOI: 10.1109/FIE49875.2021.9637155.

[19] M. Joy and M. Luck, “Plagiarism in programming assignments,” *IEEE Transactions on Education*, vol. 42, no. 2, pp. 129–133, May 1999. DOI: 10.1109/13.762946.

[20] T. Foltýnek *et al.*, “Detecting machine-obfuscated plagiarism,” in *Sustainable Digital Communities*, vol. 12051, Springer International Publishing, Mar. 2020, pp. 816–827. DOI: 10.1007/978-3-030-43687-2_68.

[21] T. Sağlam, M. Brödel, L. Schmid, and S. Hahner, “Detecting automatic software plagiarism via token sequence normalization,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24, Association for Computing Machinery, Apr. 2024, 113:1–113:13. DOI: 10.1145/3597503.3639192.

[22] H. Gimpel *et al.*, “Unlocking the power of generative ai models and systems such as gpt-4 and chatgpt for higher education,” Mar. 2023.

[23] M. Daun and J. Brings, “How chatgpt will change software engineering education,” in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ser. ITiCSE 2023, Association for Computing Machinery, Jun. 2023, pp. 110–116. DOI: 10.1145/3587102.3588815.

[24] M. Khalil and E. Er, “Will chatgpt get you caught? rethinking of plagiarism detection,” *Interacción*, vol. 14040, pp. 475–487, Feb. 2023, 10.48550/arXiv.2302.04335. DOI: 10.48550/arXiv.2302.04335.

[25] T. Sağlam, S. Hahner, L. Schmid, and E. Burger, “Automated detection of ai-obfuscated plagiarism in modeling assignments,” in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ser. ICSE-SEET ’24, Association for Computing Machinery, Apr. 2024, pp. 297–308. DOI: 10.1145/3639474.3640084.

[26] L. Nichols, K. Dewey, M. Emre, S. Chen, and B. Hardekopf, “Syntax-based improvements to plagiarism detectors and their evaluations,” in *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE ’19, ACM, Jul. 2019, pp. 555–561. DOI: 10.1145/3304221.3319789.

[27] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, “Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection,” *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1157–1177, Dec. 2017. DOI: 10.1109/TSE.2017.2655046.

[28] OpenAI. “Introducing chatgpt.” Accessed: 2023-04-12. ().

[29] JPlag. “Jplag github repository,” GitHub. (2023), (visited on 2023).

[30] M. Novak, “Effect of source-code preprocessing techniques on plagiarism detection accuracy in student programming assignments,” Ph.D. dissertation, University of Zagreb. Faculty of Organization and Informatics, 2020.

[31] D. Pawelczak, “Benefits and drawbacks of source code plagiarism detection in engineering education,” in *2018 IEEE Global Engineering Education Conference (EDUCON)*, IEEE, Apr. 2018, pp. 1048–1056. DOI: 10.1109/EDUCON.2018.8363346.

[32] R. M. Maisch, “Preventing refactoring attacks on software plagiarism detection through graph-based structural normaliza-

tion," M.S. thesis, Karlsruher Institut für Technologie (KIT), 2024, 77 pp. DOI: 10.5445/IR/1000172813.

[33] J. Faidhi and S. Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment," *Computers & Education*, vol. 11, no. 1, pp. 11–19, 1987. DOI: 10.1016/0360-1315(87)90042-x.

[34] T. Sağlam, S. Hahner, J. W. Wittler, and T. Kühn, "Token-based plagiarism detection for metamodels," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '22, ACM, Oct. 2022, pp. 138–141. DOI: 10.1145/3550356.3556508.

[35] L. Schmid *et al.*, "Performance-detective: Automatic deduction of cheap and accurate performance models," in *Proceedings of the 36th ACM International Conference on Supercomputing*, ser. ICS '22, Association for Computing Machinery, 2022. DOI: 10.1145/3524059.3532391.

[36] L. Schmid, T. Sağlam, M. Selzer, and A. Koziolek, "Cost-efficient construction of performance models," in *Proceedings of the 4th Workshop on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn STrategy*, ser. PERMAVOST '24, Association for Computing Machinery, Jun. 2024, p. 1. DOI: 10.1145/3660317.3660322.

[37] N. Sağlam Timur Niehues, S. Hahner, and L. Schmid, *Supplementary material for "mitigating obfuscation attacks on software plagiarism detectors via subsequence merging"*, Karlsruhe Institute of Technology, Zenodo, Jan. 2025. DOI: 10.5281/zenodo.14679485.

[38] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 728–738, Nov. 1984. DOI: 10.1109/TSE.1984.5010301.

[39] V. R. Basili, "Software modeling and measurement: The goal/question/metric paradigm," Tech. Rep., Sep. 1992.

[40] J. C. Paiva, J. P. Leal, and Á. Figueira, "Progpedia: Collection of source-code submitted to introductory programming assignments," *Data in Brief*, vol. 46, p. 108 887, Feb. 2023. DOI: https://doi.org/10.1016/j.dib.2023.108887.

[41] V. Ljubovic, *Programming homework dataset for plagiarism detection*, 2020. DOI: 10.21227/71fw-ss32. [Online]. Available: https://dx.doi.org/10.21227/71fw-ss32.

[42] V. Ljubovic and E. Pajic, "Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories," *IEEE Access*, vol. 8, pp. 96 505–96 514, 2020. DOI: 10.1109/ACCESS.2020.2996146.

[43] M. Brödel, "Preventing automatic code plagiarism generation through token string normalization," bachelor's thesis, Karlsruher Institut für Technologie (KIT), 2023. DOI: 10.5445/IR/1000165371.

[44] J. Reason, "The contribution of latent human failures to the breakdown of complex systems," *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, vol. 327, no. 1241, pp. 475–484, Jul. 1990. DOI: 10.4324/9781315092898-2. (visited on 2024).

[45] M. Stytz, "Considering defense in depth for software applications," *IEEE Security & Privacy*, vol. 2, no. 1, pp. 72–75, Jan. 2004. DOI: 10.1109/MSECP.2004.1264860.

[46] R. Lippmann *et al.*, "Validating and restoring defense in depth using attack graphs," in *MILCOM 2006 - 2006 IEEE Military Communications conference*, IEEE, Oct. 2006, pp. 1–10. DOI: 10.1109/MILCOM.2006.302434.

[47] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems, Third Edition*, 3rd ed. John Wiley & Sons Inc., Dec. 2020. DOI: 10.1002/9781119644682.

[48] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Jun. 2012, pp. I–XXIII, 1–236. DOI: 10.1007/978-3-642-29044-2.

[49] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, Dec. 2008. DOI: 10.1007/s10664-008-9102-8.

[50] M. Wise, "String similarity via greedy string tiling and running karp-rabin matching," *Unpublished Basser Department of Computer Science Report*, Jan. 1993.

[51] M. J. Wise, "Neweyes: A system for comparing biological sequences using the running karp-rabin greedy string-tiling algorithm," *Proc Int Conf Intell Syst Mol Biol*, vol. 3, pp. 393–401, 1995.

[52] R. Maertens *et al.*, "Dolos: Language-agnostic plagiarism detection in source code," *Journal of Computer Assisted Learning*, vol. 38, no. 4, pp. 1046–1061, Aug. 2022. DOI: https://doi.org/10.1111/jcal.12662.

[53] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03, ACM, Jun. 2003, pp. 76–85. DOI: 10.1145/872757.872770.

[54] C. Liu, C. Chen, J. Han, and P. S. Yu, "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*, ser. KDD '06, Association for Computing Machinery, Aug. 2006, pp. 872–881. DOI: 10.1145/1150402.1150522.

[55] P. Krieg, "Preventing code insertion attacks on token-based software plagiarism detectors," Bachelor's Thesis, Karlsruhe Institute of Technology, Sep. 2022, 51 pp. DOI: 10.5445/IR/1000154301.

[56] O. Karnalim, Simon, and W. Chivers, "Preprocessing for source code similarity detection in introductory programming," in *Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '20, Association for Computing Machinery, Nov. 2020. DOI: 10.1145/3428029.3428065.

[57] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" In *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, IEEE Computer Society, Jan. 2009, pp. 485–495. DOI: 10.1109/ICSE.2009.5070547.

[58] L. Mariani and D. Micucci, "Audentes: Automatic detection of tentative plagiarism according to a reference solution," *ACM Trans. Comput. Educ.*, vol. 12, no. 1, pp. 1–26, Mar. 2012. DOI: 10.1145/2133797.2133799.

[59] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: A token based large-gap clone detector," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, ACM, May 2018, pp. 1066–1077. DOI: 10.1145/3180155.3180179.

[60] K. Ly, "Normalizer: Augmenting code clone detectors using source code normalization," M.S. thesis, California Polytechnic State University, Mar. 2017. DOI: 10.15368/theses.2017.21.

[61] S. Dou, Y. Wu, H. Jia, Y. Zhou, Y. Liu, and Y. Liu, *Cc2vec: Combining typed tokens with contrastive learning for effective code clone detection*, May 2024.

[62] R. Yang, J. L. Van Etten, and S. M. Dehm, "Indel detection from dna and rna sequencing data with transindel," *BMC Genomics*, vol. 19, no. 1, p. 270, Apr. 2018. DOI: 10.1186/s12864-018-4671-4.

[63] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970. DOI: https://doi.org/10.1016/0022-2836(70)90057-4.