Evaluating Software Plagiarism Detection in the Age of AI

Automated Obfuscation and Lessons for Academic Integrity

Timur Sağlam · Larissa Schmid

Abstract Plagiarism in programming assignments is a persistent issue in computer science education, increasingly complicated by the emergence of automated obfuscation attacks. While software plagiarism detectors are widely used to identify suspicious similarities at scale and are resilient to simple obfuscation techniques, they are vulnerable to advanced obfuscation based on structural modification of program code that preserves the original program behavior. While different defense mechanisms have been proposed to increase resilience against these attacks, their current evaluation is limited to the scope of attacks used and lacks a comprehensive investigation regarding AI-based obfuscation. In this paper, we investigate the resilience of these defense mechanisms against a broad range of automated obfuscation attacks, including both algorithmic and AI-generated methods, and for a wide variety of real-world datasets. We evaluate the improvements of two defense mechanisms over the plagiarism detector JPlag across over four million pairwise program comparisons. Our results show significant improvements in detecting obfuscated plagiarism instances, and we observe an improved detection of AI-generated programs, even though the defense mechanisms are not designed for this use case. Based on our findings, we provide an in-depth discussion of their broader implications for academic integrity and the role of AI in education.

Keywords Source Code Plagiarism · Plagiarism Detection · Plagiarism Obfuscation · Obfuscation Attack · CS Education · Generative AI

Timur Sağlam
KASTEL, Karlsruhe Institute of Technology (KIT), Germany
E-mail: saglam@kit.edu

Larissa Schmid
TCS, KTH Royal Institute of Technology, Sweden
E-mail: lgschmid@kth.se

1 Introduction

Plagiarism is a prevalent challenge in computer science education, facilitated by the ease of duplicating and modifying digital assignments [15, 42, 35]. Although students generally acknowledge plagiarism as academic misconduct, some will engage in it despite the threat of consequences [68]. Therefore, students are creative in *obfuscating* their plagiarism to conceal the relation to its source [51]. In the case of programming assignments, students commonly utilize techniques such as renaming, reordering, or restructuring [45, 27]. Plagiarism in programming assignments is particularly pronounced in beginner-level and mandatory courses, such as introductory programming courses [50].

While checking submissions for plagiarism manually is feasible for small course sizes, this quickly becomes infeasible for larger course sizes [9, 33] as the number of required pairwise comparisons grows quadratically – reaching 1,225 comparisons for just 50 submissions. This results in the individual risk of detection decreasing with rising course sizes [74]. In light of these issues, it is common for educators to use software plagiarism detection systems to uphold academic integrity for programming assignments [18]. These systems automate parts of the detection process and thus allow tackling the problem of plagiarism detection at scale. Thus, educators strongly rely on software plagiarism detectors to guide them in inspecting suspicious candidates. Plagiarism detectors analyze sets of programs to detect pairs with a suspiciously high degree of similarity [53]. However, assessing which suspicious candidates qualify as plagiarism is ultimately a human decision, given the underlying ethical considerations [16, 70]. Overall, plagiarism detection systems help identify plagiarism instances and, when using such systems is communicated [30], deter students from plagiarizing firsthand [7].

Crucially, plagiarism detectors are only effective when defeating them takes more effort than completing the actual assignment [18]. Yet, manually obfuscating a program successfully is tedious and requires understanding the underlying program, therefore requiring time and programming proficiency. Thus, a widespread assumption was that evading detection is not feasible for novice programmers as obfuscating the program requires more time than it takes to complete the actual assignments and requires a profound understanding of programming languages [25]. However, this assumption has been broken with the recent rise of automated *obfuscation attacks* [18, 21, 6, 51] which require neither time nor programming proficiency to employ successfully. These *obfuscation attacks* aim to avoid detection by strategically altering a plagiarized program, thus obscuring the relation to its original [60]: State-of-the-art detection approaches compare the structure of programs by identifying similarities between code fragments [43]. Thus, most obfuscation attacks alter the structural properties of the program, ideally without affecting its behavior. Early automated attacks relied purely on algorithmic approaches, for example, via repeated statement insertion [18]. However, the challenge intensifies with the rise of generative artificial intelligence, especially Large Language Models (LLMs) [17], making the obfuscation of plagiarism even more accessible with less effort than ever before [32, 61]. While state-of-the-art detectors exhibit some obfuscation resilience to changes like retyping and lexical changes, this does not apply to all types of obfuscation attacks [18, 38]. Thus, automated obfuscation attacks present a significant challenge for today's plagiarism detection systems, as they must now contend with increasingly sophisticated obfuscation techniques that can evade detection while maintaining the original program's functionality.

In recent work [60, 63], we proposed defense mechanisms tailored towards different obfuscation attacks. However, it is unclear how well they work against a broader range of attacks. Token Sequence Normalization [60] explicitly targets dead code insertion and statement reordering. Subsequence Match Merging [63] employs a heuristic to counteract any obfuscation attack that aims at interrupting the match found between two program codes. While we show that these approaches are effective against the obfuscation attacks they are targeting individually, educators typically do not know which obfuscation attacks students employed, thus making it hard to select a specific appropriate defense mechanism. Ideally, a combination of these defense mechanisms can be used to provide greater resilience. However, it is currently unclear whether the different defense mechanisms can be combined to achieve this while not producing false-positive results due to the overapproximation of similarities. Moreover, with the steady improvements of LLMs, AI-based obfuscation attacks become more and more feasible. However, the defense mechanisms have not been comprehensively tested against a broad spectrum of automated attacks, including both algorithmic and AI-based obfuscation techniques. Finally, their applicability to detecting AI-generated programs is yet to be assessed.

1.1 Research Contributions

In this paper, we investigate the resilience of software plagiarism detectors to different automated obfuscation attacks. As a first contribution (C1), we present a comprehensive evaluation of various automated obfuscation attacks, including both algorithmic and AI-based methods, also exploring the feasibility of using AI to generate programs. In addition to examining defense mechanisms on their own, we also explore their combined use. As a second contribution (C2), we complement our technical findings with a detailed discussion of their broader implications – not only for improving software plagiarism detection, but also for issues related to academic integrity and the role of AI in education.

1.2 Evaluation and Results

We conducted a comprehensive empirical evaluation to demonstrate the effectiveness of defense mechanisms against obfuscation attacks. Over the entirety of this evaluation, we analyze over *4 million data points*, each representing a pairwise comparison of two programs. Our datasets comprise over 14,000 files with over a million lines of code.

We evaluate the defense mechanisms with a wide range of real-world datasets [48, 37, 60] from different university courses. These courses range from mandatory undergraduate courses to master's-level elective courses. Furthermore, they contain different-sized programs, thus representing typical use cases for software plagiarism detection. In our evaluation, we employ a total of *five* different obfuscation techniques for the plagiarism instance. We use both algorithmic and AI-based obfuscation and use existing obfuscation tools [1, 18].

We demonstrate that the defense mechanisms offer broad obfuscation resilience across diverse datasets and attack types, thus significantly advancing resilience against automated obfuscation attacks for programming assignments. Notably, we achieved a median similarity difference increase of up to 99.65 percentage points against semantic-preserving insertion-based obfuscation. We also show substantial improvements against refactoring-based attacks (up to 22 percentage points). While resilience against AI-based obfuscation was comparatively lower (up to 19 percentage points), we still observe improved detection rates, including a notable 8.92 percentage point increase in identifying AI-generated programs, even though the defense mechanisms are not designed for this use case. These findings underscore the effectiveness of current defense mechanisms in defending against a wide range of obfuscation attacks, allowing for resilient source code plagiarism detection.

1.3 Outline

The remainder of this paper is structured as follows. First, section 2 introduces the foundations of automated obfuscation attacks and their impact on token-based plagiarism detection. In section 3, we present the defense mechanisms designed to counter these attacks, which we evaluate in this paper. Next, section 4 outlines our evaluation methodology, followed by section 5, which reports the results across various datasets and obfuscation strategies. We discuss threats to validity in section 6, and provide a broader discussion of implications and insights in section 7. Finally, section 8 reviews related work, and section 9 concludes.

2 Automated Obfuscation Attacks

Students often attempt to conceal plagiarism by obfuscating its origin [25, 44, 27, 51]. Since cosmetic changes alone (e.g., lexical edits) are insufficient against structural comparison [43], they increasingly alter program structure while preserving its behavior. Common strategies include inserting statements, refactoring control structures [27], or simplifying, combining, and splitting code fragments [45]. These techniques, however, are neither new nor especially worrying, as manual obfuscation is tedious, error-prone, and requires understanding the original program to be plagiarized [25]. *Automated* obfuscation attacks, however, introduce a paradigm shift. Automated obfuscation is both faster and more effective than manual obfuscation.

All automated obfuscation attacks targeting software plagiarism detectors – whether manual, algorithmic, or AI-based – are based on a single underlying principle: avoiding detection by strategically altering a plagiarized program, thus obscuring the relation to its original [60]. As state-of-the-art detection approaches compare the structure of programs by identifying similarities between code fragments [43], obfuscation attacks try to alter the structural properties of the program, ideally without affecting its behavior [44, 27, 51]. Their intended outcome is to disrupt the matching of fragments between programs, thus leading to a reduced similarity score [18]. Specifically, the goal is to prevent the detector from matching fragments above the specified match length cut-off threshold. This can be achieved by breaking up matching code fragments into shorter sub-fragments. However, to impact the detection quality of a software plagiarism detector, the obfuscation must affect the linearized program representation of the detector, which in the case of token-based approaches is the token sequence [60]. Consequently, modifications to the program code that do not affect the internal program representation are inherently ineffective. For example, renaming program elements does not affect token-based approaches, as names are omitted during the tokenization [53, 61] (see Figure 1).

Devore-McDonald and Berger [18] present an automated attack based on repeated insertion of dead statements into an existing program. This approach effectively deceives both JPlag [54] and MOSS [2], reducing the calculated similarity between a plagiarism instance and its source below the average similarity of unrelated student solutions. Similar attacks can be designed based on the automated application of refactoring operations [40]. The rapid improvements in the field of generative artificial intelligence significantly exacerbate this problem [34]. AI-powered tools can generate or alter source code [17] while requiring little manual effort and technical knowledge, making automated obfuscation more accessible than ever before [6, 32]. Tools like ChatGPT combine the capabilities of generative artificial intelligence with the approachable interface of a chatbot [61], thus further reducing the entry barrier to using generative AI. Essentially, automated obfuscation attacks make successfully evading plagiarism detection systems easier than ever.

3 Defense Mechanisms

In the following, we present defense mechanisms against automated obfuscation attacks from our prior work [60, 63]. These defense mechanisms are designed to provide broad resilience against automated obfuscation attacks by being largely language-independent, applicable across multiple programming languages, and agnostic to the underlying detection system, making them suitable for integration into any state-of-the-art, token-based detector such as MOSS, JPlag, or Dolos. Token-based plagiarism detectors are inherently immune to lexical obfuscation, and usually also to data-based obfuscation (see Figure 1). Our defense mechanisms additionally provide resilience to structural and complex attacks.

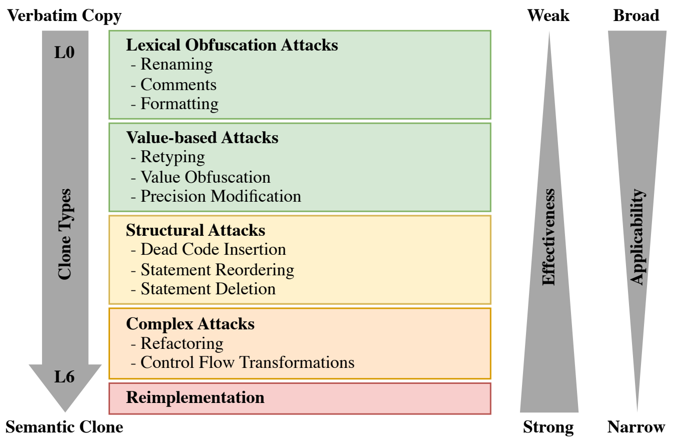

Fig. 1 Categorization of obfuscation attacks corresponding to the clone types by Karnalim [27] based on [20], illustrating the effectiveness and applicability corresponding to the obfuscation complexity. Based on [63].

3.1 Token Sequence Normalization

Token Sequence Normalization (TSN) [60] is a normalization-based defense mechanism designed to counter obfuscation attacks based on dead code insertion or statement reordering (*structural attacks* in Figure 1). It uses a *Token Normalization Graph* (TNG), a graph-based abstraction similar to program dependence graphs that captures semantic interdependencies between tokens. The normalization process begins by enriching the token sequence with language-independent semantic information. From this enriched sequence, a TNG is constructed to represent a partial ordering over tokens, abstracting away from the original code structure. The normalized token sequence is then generated from this graph by removing dead nodes and reverting reordered code via topological sorting. This normalization is performed before the pairwise comparison step in the plagiarism detection pipeline, allowing the detector to virtually de-obfuscate plagiarized code while preserving the scalability of token-based methods. The only language-specific component of this approach is the extraction of semantic information required for enrichment, which is consistent with the language-dependent nature of tokenization itself and does not introduce additional constraints on the detection system.

3.2 Subsequence Match Merging

Subsequence Match Merging (SMM) [63] is a defense mechanism designed to counter obfuscation in an attack-independent and language-independent manner. Thus, it covers any of the categories presented in Figure 1. It is based on the observation that all effective obfuscation attacks must disrupt the matching of code fragments by breaking up the internal linearized program representation of detectors. SMM operates on these internal representations by heuristically merging neighboring fragment matches in pairs of programs. This process is applied iteratively, subsuming gaps caused by obfuscation until no more neighboring matches can be merged. SMM thus restores the continuity of matches, which reverts the effects of the obfuscation attack without significantly increasing the false positive rate. Crucially, the approach is entirely language-independent and agnostic to the type of obfuscation, as it does not rely on the semantics of the internal representation.

3.3 Combination of Both

As TSN and SMM operate during different steps of the detection pipeline, they are complementary and can be combined. In our evaluation, we explore a hybrid defense strategy that applies TSN as a pre-processing step after parsing the input programs, followed by SMM as a post-processing step after computing matching subsequences. The rationale behind this combination is twofold. TSN provides strong resilience to insertion-based obfuscation, which is one of the easiest and most effective obfuscation attacks. SMM provides broad resilience against a range of obfuscation attacks, as its heuristic nature avoids making assumptions on the specifics of an obfuscation attack. This layered approach is expected to offer broad resilience, combining the benefits of both approaches.

4 Evaluation Methodology

This section outlines the methodology used to evaluate the effectiveness of the proposed defense mechanisms regarding obfuscation resilience and detection quality. We evaluate the aforementioned defense mechanisms regarding obfuscation resilience with the plagiarism detector JPlag as the baseline, as it is not only considered state-of-the-art [5] but also the most referenced and compared approach [45]. We use real-world datasets from different university courses. We employ a total of four different obfuscation techniques for the plagiarism instances: Dead code insertion, automated refactoring, AI-based obfuscation, and AI-based generation. Over the entirety of this evaluation, we analyze over *4.1 million data points*, each representing a similarity value of a pairwise comparison of two programs. The datasets sum up to over 14,000 files with around a million source lines of code.

In our evaluation, we analyze software plagiarism detectors for the purpose of evaluating their resilience with respect to automated obfuscation techniques in the context of computer science education. In this context, we investigate the following evaluation questions:

Q1 To what degree do defense mechanisms affect the similarity scores of unrelated programs?

Q2 What degree of resilience do defense mechanisms achieve against insertion-based obfuscation?
Q3 What degree of resilience do defense mechanisms achieve against refactoring-based obfuscation?
Q4 What degree of resilience do defense mechanisms achieve against AI-based obfuscation?
Q5 How well can we distinguish AI-generated from human programs?
Q6 What impact do defense mechanisms have on threshold-based plagiarism generators?

For *Q1*, we examine the similarity values for pairs of original programs as a metric. For *Q2* to *Q5*, we look at the similarity value differences between plagiarized and original programs and conduct comprehensive statistical tests by computing the statistical significance (p-values) as well as the practical significance (effect size) of these differences. To answer *Q6*, we measure the difference in the runtime of the plagiarism generator and the difference in the number of inserted lines in the plagiarism instance.

In the following, we outline the similarity metrics and statistical measures in detail. Next, we discuss our choice of baseline. We then describe the datasets we used. Finally, we explain obfuscation attacks used for obfuscation purposes.

4.1 Similarity Metrics

As plagiarism detection systems compute similarity scores between program pairs, these scores serve as the primary basis for identifying suspicious cases. In practice, similarity scores guide which candidates are reviewed first, as no objective indicator alone can confirm plagiarism. Detection tools typically provide similarity distributions and ranked lists of pairs – both derived from similarity scores – to support human inspection. The detailed visualization of matched code fragments is usually consulted only after identifying high-similarity candidates. Evaluating detection quality requires distinguishing between different types of program pairs, each requiring separate analysis of the detector's similarity scores:

1. *Original Pair*: Two original programs developed independently of each other, without shared origin.
2. *Plagiarism-To-Source Pair*: A plagiarism instance and its source program.

To clearly distinguish plagiarism from unrelated programs during human inspection, plagiarism pairs must have high similarity scores [60], while unrelated pairs must have low scores. Ideally, there is no overlap, with plagiarism pairs always showing higher similarity. However, in practice, overlap occurs as changes, especially obfuscation techniques, can reduce the similarity between a plagiarized program and its source. Thus, the *difference* in similarity between plagiarism and unrelated pairs is crucial to measure detection quality.

A common anti-pattern in existing works is evaluating plagiarism detectors using a fixed similarity threshold where scores above count as successful detections, and those below do not. While this simplifies deriving precision, recall, and F1 scores, this approach is *fundamentally flawed*, as the threshold can arbitrarily influence results and it can be tuned to favor one approach. Since no universal threshold fits all datasets, thresholds are chosen arbitrarily. Due to varying similarity distributions for different datasets, they can only be set after seeing the results, thus introducing a strong bias. A threshold-based evaluation reduces plagiarism detection to a binary classification problem, which is insufficient due to the mentioned problem of overlap. Fundamentally, it measures only whether plagiarism is detected (and only according to some arbitrary criterion), not how well it is detected.

For these reasons, we focus on the difference in similarity between plagiarism and non-plagiarism pairs. The larger this difference, the easier it is to detect plagiarism effectively. Thus, we measure to *what extent* a detection approach can produce such a difference between these pairs. This avoids overabstracting the problem into a binary classification. Varying statistical measures can be used to calculate the differences between these types of pairs. We use measures of central tendency like the mean (μ) or median (Q_2) and measures of spread like the difference between the interquartile ranges (Q_1 to Q_3). Figure 2 shows an **example** of the median difference ($\Delta median$) and interquartile range distance (ΔIQR). Each metric highlights different data properties: $\Delta median$ resists outliers more than $\Delta mean$, while ΔIQR indicates how well the main value ranges are separated. A ΔIQR above zero signifies that at least 75% of values in both sets do not overlap.

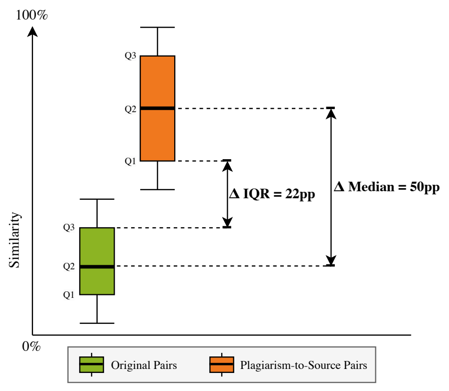

Fig. 2 Difference metrics comparing the separation between plagiarism instances and original student solutions based on the median and interquartile range (IQR) distances measured in percentage points.

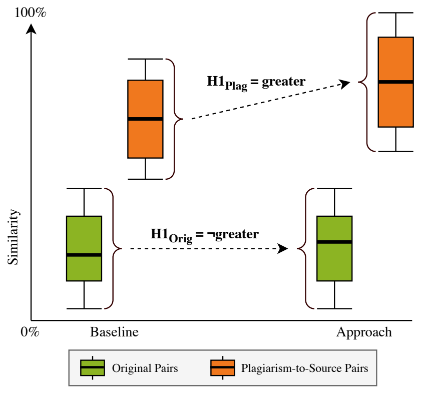

Fig. 3 Visualization of alternative hypotheses ($H1$) for one-sided significance tests between plagiarism and original pairs. For plagiarism, the shift should be significantly greater ($H1_{Plag}$), while for originals, it should not ($H1_{Orig}$).

We also employ statistical tests to assess statistical and practical significance. To that end we conduct one-sided Wilcoxon signed-rank tests to compare improvements in detection quality between approaches. Different hypotheses apply depending on the pair type, as shown in Figure 3. For plagiarism pairs, we test if they show a significant location shift ($H1_{Plag}$), i.e., higher scores. For original pairs, we test that no significant shift occurs ($H1_{Orig}$). For the practical significance, we use *Cliff's delta* δ [13] as an effect size measure, as we deal with non-normal distributions and paired data. Although a paired version of *Cohen's d* [14] exists, it is sensitive to outliers and only mildly robust to non-normality, making it unsuitable here. In contrast, while not ideal for paired data, *Cliff's delta* δ remains useful for rank-based comparisons, offering robustness to non-normality and variance differences. There are no established categories to interpret the resulting δ values. Thus, we base our interpretation on the derived categories by Romano et al. [56] based on *Cohens d*:

$$\delta\ \mathit{Interpretation} = \begin{cases} \text{Negligible} & \text{if } 0 \leq |\delta| < 0.147 \\ \text{Small} & \text{if } 0.147 \leq |\delta| < 0.33 \\ \text{Medium} & \text{if } 0.33 \leq |\delta| < 0.474 \\ \text{Large} & \text{if } 0.474 \leq |\delta| < 0.7 \\ \text{Very Large} & \text{if } 0.7 \leq |\delta| \leq 1 \end{cases} \tag{1}$$

Note that a negative effect size suggests an adverse interpretation, e.g., that the comparison group is greater than the target group.

4.2 Choice of Baselines

For the evaluation of programming assignments, we utilize JPlag as our baseline. JPlag is not only regarded as a state-of-the-art tool [5, 45] but also stands out as one of the most frequently referenced approaches and the most compared approach in the literature [45]. Its widespread use in practice and scientific literature makes it an ideal standard for assessing programming-based plagiarism.

While MOSS is widely used [45], we excluded it as a baseline for four reasons: (1) it only returns a subset of similarity values, skewing comparisons; (2) it is closed-source and cannot be extended with our defense mechanisms; (3) it requires sending data to U.S.-based servers, conflicting with General Data Protection Regulation of the European Union (GDPR); and (4) it imposes strict usage limits, making large-scale evaluation infeasible. We also excluded Dolos due to its limited adoption, and lack of support for multi-file programs. Multi-file programs are common in programming assignments, making it essential to evaluate plagiarism detection systems on such datasets. Nonetheless, both MOSS and Dolos are token-based and equally vulnerable to obfuscation attacks [18, 62], underscoring the need for improved defenses.

As an example, Figure 4 shows the results of JPlag, MOSS, and Dolos on a simple dataset using insertion-based obfuscation. All three tools yield low similarity scores for the plagiarism instances, causing overlap with unrelated programs. Note that MOSS omits many lower similarity values by design, and the dataset includes only small, single-file programs compatible with Dolos.

4.3 Datasets

We used a total of six real-world datasets; four are publicly available, and two are internal. Since public datasets are limited in both size and number, we supplement them with internal datasets. All datasets come from an educational setting but stem from different courses and assignment types. First, we used two tasks from the publicly available collection *PROGpedia* [48]. Here, Task 19 covers the design of a graph data structure and a depth-first search to analyze a social network. Task 56 concerns minimum spanning trees using Prim's algorithm. Both datasets contain small Java programs. For both datasets, we used only syntactically and semantically correct solutions and the latest version of each program.

Next, we used the *TicTacToe* dataset [60], which contains command-line-based Java implementations of the paper-and-pencil game TicTacToe. This dataset is from an introductory programming class at KIT, specifically from a weekly assignment. This dataset contains many programs, each of which is medium-sized. We also used the *BoardGame* dataset [60]. This assignment is from the same course as the *TicTacToe* dataset. However, it is the final project of the course. Here, the task is also a command-line-based game; however, this time, it is a comprehensive board game. Thus, it contains very large programs.

Finally, we used two tasks from the publicly available homework dataset by Ljubovic and Pajic [37]. 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 six datasets listed in Table 1.

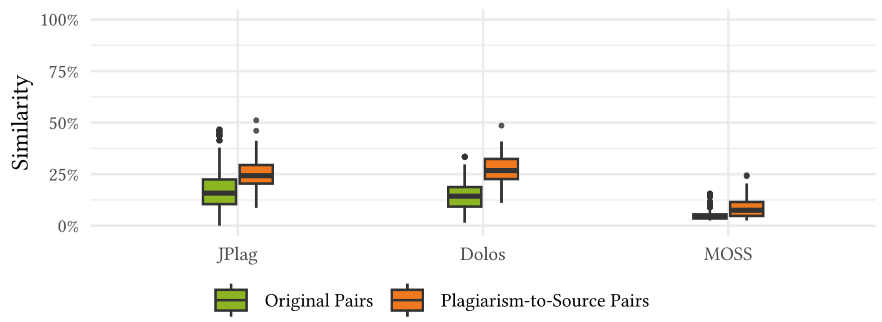

Fig. 4 Obfuscation vulnerability illustrated for the three token-based approaches JPlag, Dolos, and MOSS with the dataset PROGpedia-19 [48] and plagiarism instances automatically obfuscated via statement insertion.

4.4 Obfuscation Attacks

We evaluate four automated obfuscation attacks, two algorithmic and two AI-based. For ethical reasons, we briefly discuss these attacks without revealing details to avoid encouraging their use.

The first algorithmic obfuscation attack is the insertion of dead statements. For this, we employ two different tools. The first one is MOSSad [18]. As previously discussed, it is indeterministic and operates threshold-based. The second is *PlagGen* [8], which is similar to MOSSad but is deterministic and exhaustive. In both cases, the statement insertion uses statements from the original program and a pool of pre-defined statements. Furthermore, both ensure that the inserted statements do not change the behavior of the programs. Thus, this obfuscation attack is semantic-preserving. We use PlagGen for Java and MOSSad for C++, as these are the languages supported by each tool. Second, we employ the refactoring-based obfuscation attack by Maisch [40], which leverages *Spoon* [52] and automatically applies semantic-preserving refactoring operations at random positions at the AST level to obfuscate a program. In detail, the refactoring operations include optional wrapping, extracting expressions as new variables, introducing constant container classes and extraction of constants, swapping if-else-statements and inverting the corresponding conditions, inserting methods and constructors, and introducing access methods for existing fields. As the behavior of the programs is not changed, this obfuscation attack is also semantic preserving. This implementation of the obfuscation attack only supports Java programs, so we only use four of the six datasets with this obfuscation attack.

For the AI-based obfuscation attacks, we exploit OpenAI's GPT-4 for automated plagiarism, which is currently the state-of-the-art LLM. There are generally two ways of using generative AI to *cheat* for programming assignments: *AI-based 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 fifteen different prompts, mimicking how students would ask GPT to obfuscate their

Dataset Name	# Programs	Mean LOC	Total LOC	Language	Source
PROGpedia Task 19	27	131	3.5K	Java	[48]
PROGpedia Task 56	28	85	2.4K	Java	[48]
TicTacToe	626	236	148K	Java	internal
BoardGame	434	1529	663K	Java	internal
Homework Task 1	59	282	17K	C/C++	[37]
Homework Task 5	18	123	2.2K	C/C++	[37]

Table 1 Programming assignment datasets used for the evaluation with the number of included programs, mean size in lines of code (LOC) excluding comments, the programming language, and source of the dataset.

plagiarism. The prompts range from requesting minor structural changes to requesting a reimplemented version of the original program. As for this attack, the programs need to be sent to the OpenAI GPT server; we did not use it for the BoardGame dataset due to its sensitive nature. Finally, we use full generation as the final obfuscation. However, we can only employ it for the TicTacToe dataset, as we require the full assignment description and test cases to test for the expected behavior. AI-based obfuscation is a semantic agnostic attack. While the prompts contain instructions to preserve the program behavior, there are generally no guarantees that the changes proposed by GPT-4 conform to these instructions. Similarly, for AI-based generation, there is no guarantee that the programs fully implement all details requested by the task. In sum, we use the following four techniques to create 787 plagiarized programs (see Table 2 for details):

1. **Insertion-based Obfuscation** (semantic-preserving): Inserting dead statements into the program (PlagGen [8] for Java and MOSSad [18] for C/C++).
2. **Refactoring-based Obfuscation** (semantic-preserving): Applying a variety of semantic-preserving refactoring operations, for example, transformations of control structures, field access, and method granularity [40].
3. **AI-based Obfuscation** (sematic-agnostic): We obfuscate human solutions with GPT-4 [1] based on 15 varying prompts requesting structural changes.
4. **AI-based Generation** (sematic-agnostic): We fully generate AI-based solutions with GPT-4 [1] based on only the textual task description of the assignment.

5 Evaluation Results

In the following, we provide the results of our evaluation, which demonstrate that the defense mechanisms offer broad obfuscation resilience across diverse datasets and attack types. We compare them to JPlag without any defense mechanisms as the baseline. We provide a replication package for this evaluation [59]. The key findings from our comprehensive evaluation, which offer new insights beyond prior evaluations, can be summarized as follows: *Insertion-based Obfuscation:* Combining both defense mechanisms provides improved

Obfuscation Attack Type	PROGp.-19	PROGp.-56	Homew.-1	Homew.-5	TTT	BoardGame
Insertion-based Obf.	27	28	59	17	50	20
Alteration-based Obf.	27	28	59	17	50	20
Refactoring-based Obf.	27	28	-	-	50	20
AI-based Obf.	75	75	75	75	75	-
AI-based Generation	50	50	-	-	50	-

Table 2 Overview on the number of plagiarized programs per dataset and obfuscation attack type (851 in total). Each of the 15 prompts is applied to 5 originals for the AI-based obfuscation.

ds	Variant	Pairs	p	W	δ	δ *Int.*	δ 95% CI	n
Pp.-19	TSN	OP	8.3e-05	13,137	0.005	Negligible	[-0.08, 0.09]	351
	SMM	OP	$< 1e-10$	6,441	0.178	Small	[0.09, 0.26]	351
	Both	OP	$< 1e-10$	26,133	0.185	Small	[0.10, 0.27]	351
Pp.-56	TSN	OP	0.25	3,580	-0.034	Negligible	[-0.11, 0.04]	378
	SMM	OP	$< 1e-10$	4,753	0.154	Small	[0.08, 0.23]	378
	Both	OP	$< 1e-10$	14,018	0.148	Negligible	[0.07, 0.22]	378
TTT	TSN	OP	$< 1e-10$	1,256,914,634	0.009	Negligible	[0.01, 0.01]	188,805
	SMM	OP	$< 1e-10$	3,346,969,836	0.177	Small	[0.17, 0.18]	188,805
	Both	OP	$< 1e-10$	6,099,071,135	0.186	Small	[0.18, 0.19]	188,805
BG	TSN	OP	$< 1e-10$	1,379,905,112	0.021	Negligible	[0.01, 0.03]	67,161
	SMM	OP	$< 1e-10$	1,637,665,065	0.104	Negligible	[0.10, 0.11]	67,161
	Both	OP	$< 1e-10$	2,069,412,722	0.124	Negligible	[0.12, 0.13]	67,161
Hw.-1	TSN	OP	1	378,825	-0.069	Negligible	[-0.11, -0.03]	1,711
	SMM	OP	$< 1e-10$	557,040	0.161	Small	[0.12, 0.20]	1,711
	Both	OP	$< 1e-10$	656,971	0.106	Negligible	[0.07, 0.14]	1,711
Hw.-5	TSN	OP	0.58	2,229	-0.058	Negligible	[-0.19, 0.07]	153
	SMM	OP	3.9e-10	1,275	0.111	Negligible	[-0.02, 0.24]	153
	Both	OP	0.015	3,153	0.062	Negligible	[-0.07, 0.19]	153

Table 3 One-sided Wilcoxon signed-rank test results for **unrelated, student-made programs** regarding the potential adverse effects of our defense mechanisms compared to the baseline (sig. level of $\alpha = 0.01$, alternative hypothesis $H1 = greater$, test statistic W, effect size via Cliff's delta δ, its interpretation $\delta\, Int.$, its confidence interval CI, and the sample size n). Note that high p and low δ are desirable, as original pairs should *not* be *greater*.

resilience. *Refactoring-based Obfuscation:* Token sequence normalization minimally impacts refactoring-based attacks, as expected. However, subsequence match merging significantly improves detection, and combining both mechanisms achieves enhanced separation of plagiarized and original programs. *GPT-4-based Obfuscation:* Token sequence normalization has no positive or negative impact on AI-based obfuscation. Combining token sequence normalization and subsequence match merging shows significant improvements without drawbacks. *GPT-4-generated Programs:* Despite the defense mechanisms not being tailored for AI-generated program detection, we observe significantly improved detection across datasets for subsequence match merging. *Threshold-based Plagiarism:* The defense mechanisms substantially increase the computational cost of threshold-based obfuscation, making such an obfuscation method more tedious and also easily detectable via metrics such as program size or number of tokens.

In summary, our evaluation demonstrates that the proposed defense mechanisms are highly effective across a range of automated obfuscation attacks. The proposed defense mechanisms provide significantly (statistical and practical significance) improved obfuscation resilience without any practically significant change in false-positive rates. In the following, we present detailed results for each attack type as outlined in subsection 4.4 individually. The original, human-made programs of each dataset remain the same for all evaluation stages. Thus, we first discuss the effect of the defense mechanism on these unrelated programs.

5.1 Effect on Unrelated Programs

Effective plagiarism detection requires not only high similarity for plagiarism pairs but also minimal impact on unrelated, original programs. Our results show that the effect on such unrelated programs is negligible, indicating no meaningful increase in false positives.

Table 3 presents statistical test results for the original pairs, comparing defense mechanisms to the baseline. Token sequence normalization has virtually no impact on unrelated programs: median similarity changes range from −1.23 to +0.28 percentage points across datasets, with only one statistically significant change (PROGpedia-19), which remains insignificant due to the negligible effect size. Subsequence match merging yields small median increases (+0.78 to +6.59), statistically significant in four datasets, yet with negligible to small effect sizes and thus little to no practical significance.

When both mechanisms are combined, results are comparable to subsequence match merging alone, with median increases from +0.91 to +6.75. While statistically significant in three datasets, effect sizes remain negligible to small, confirming little to no practical significance for the impact on unrelated programs.

Answer to Q1: *The defense mechanisms have a negligible effect on unrelated programs, meaning their impact on the false positive rate is both practically and, in some cases, statistically insignificant.*

5.2 Insertion-based Obfuscation

Figure 5 presents results for insertion-based obfuscation attacks, with corresponding statistical measures in Table 4. As described earlier, we use MOSSad [18] for C++ datasets (Homework-1 and Homework-5) and PlagGen [8] for the others. The key difference lies in their termination strategies: MOSSad uses a threshold-based approach, while PlagGen exhaustively inserts code in all possible positions.

Baseline

Figure 5 shows the severe impact of insertion-based obfuscation on the baseline. Median similarity values for plagiarism pairs drop to between 5.29% (TicTacToe) and 19.59% (PROGpedia-56), leading to near-complete overlap with original pairs in all datasets except PROGpedia-56, which still shows substantial overlap. In the TicTacToe dataset, median similarity for plagiarism pairs is even *lower* than for original pairs. As detailed in Table 4, the median similarity difference between plagiarism and original pairs ranges from −0.78 percentage points (TicTacToe) to +19.56 (PROGpedia-56), with most datasets showing differences below ten points – indicating little to no separation. These results

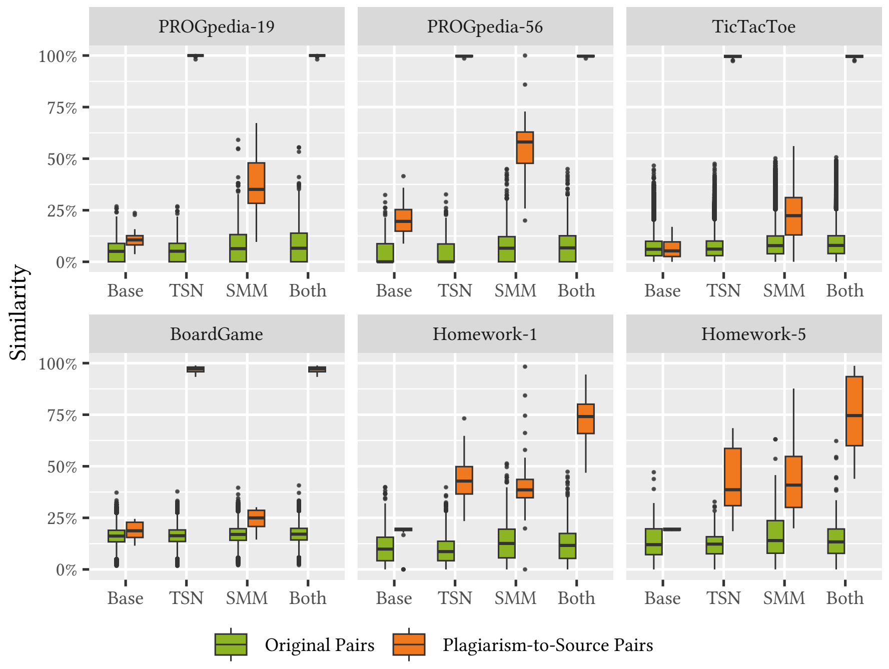

Fig. 5 Similarity scores for original program pairs and **insertion-based plagiarism** pairs. Ideally, plagiarism pairs exhibit high similarity, while original pairs should exhibit low similarity.

confirm that insertion-based obfuscation is highly effective against JPlag and similar token-based detectors, significantly hindering plagiarism detection in the absence of additional defenses.

Token Sequence Normalization

Token sequence normalization is designed to counter insertion-based obfuscation, and the results in Figure 5 confirm its strong effectiveness. For the Java datasets, it renders JPlag effectively immune to such attacks. Similarity values for plagiarism pairs increase significantly across all datasets. In the Java datasets, median similarities reach between 97.23% (BoardGame) and 100.00% (PROGpedia-19), eliminating overlap with original pairs and achieving full separation. In the C++ datasets, values rise to 38.63% (Homework-5) and 42.83% (Homework-1), with the remaining overlap confined to edge quartiles. This improvement is also evident in the similarity differences between plagiarism and original pairs (Table 4): 80.09 to 99.65 percentage points for Java, and 26.36 to 34.21 for C++ datasets. Statistical tests confirm both statistical and practical significance across all datasets (Table 5), with low p-values and very large effect sizes. Overall, token sequence normalization substantially im-

Dataset	Variant	Median	Mean	Q_1	Q_3	Δ Mean	Δ Median	Δ IQR
PROGpedia-19	Base	10.61	11.10	8.17	12.67	5.37	5.55	-0.74
	TSN	**100.00**	**99.87**	**100.00**	**100.00**	**94.12**	**94.87**	**91.06**
	SMM	35.10	37.12	28.36	47.94	27.97	28.74	15.16
	Both	**100.00**	**99.87**	**100.00**	**100.00**	90.66	93.43	86.12
PROGpedia-56	Base	19.59	21.36	14.86	25.33	16.47	19.59	6.09
	TSN	**99.65**	**99.66**	**99.47**	**100.00**	**95.14**	**99.65**	**90.82**
	SMM	58.08	56.70	47.75	62.91	48.44	51.49	35.56
	Both	**99.65**	**99.66**	**99.47**	**100.00**	91.54	92.91	86.85
TicTacToe	Base	5.29	6.36	2.57	9.66	-0.49	-0.78	-7.39
	TSN	**99.52**	**99.42**	**99.18**	**100.00**	**92.48**	**93.37**	**89.11**
	SMM	22.36	23.39	13.01	31.14	14.57	14.52	0.49
	Both	**99.52**	**99.42**	**99.18**	**100.00**	90.49	91.58	86.54
BoardGame	Base	18.72	18.66	15.49	22.85	2.48	2.53	-3.49
	TSN	**97.23**	**96.82**	**95.82**	**98.07**	**80.49**	**80.90**	**76.69**
	SMM	24.97	24.12	20.81	28.65	7.18	8.01	1.06
	Both	**97.23**	**96.82**	**95.82**	**98.07**	79.72	80.13	75.92
Homework-1	Base	19.45	18.36	18.86	19.90	7.97	9.57	3.27
	TSN	42.83	43.58	36.57	49.84	34.21	34.19	22.92
	SMM	38.57	40.63	34.75	43.63	27.58	25.99	15.25
	Both	**74.10**	**73.32**	**65.88**	**80.09**	**61.32**	**62.48**	**48.45**
Homework-5	Base	19.24	19.38	19.03	19.91	5.99	7.25	-0.65
	TSN	38.63	42.05	30.90	58.65	30.39	26.36	15.06
	SMM	40.89	43.36	30.05	54.77	27.44	26.91	6.40
	Both	**74.57**	**75.21**	**59.98**	**93.46**	**60.62**	**61.22**	**40.40**

Table 4 Statistical measures for plagiarism pairs and their differences (Δ) from original pairs for **insertion-based obfuscation** (corresponds to Figure 5). Higher values indicate better performance. Note that measures are expressed as percentages and their differences as percentage points. Highest values by a margin of 0.25 are marked in bold.

proves obfuscation resilience and demonstrates the value of targeted defenses against insertion-based obfuscation.

Subsequence Match Merging

As an attack-independent defense, subsequence match merging is not specifically tailored to insertion-based obfuscation, but still yields notable improvements over the baseline (Figure 5). Median similarity values for plagiarism pairs increase across all datasets, ranging from 22.36% (TicTacToe) to 58.08% (PROGpedia-56), with overlap mostly confined to quartile extremes. Corresponding median similarity *differences* between plagiarism and original pairs (Table 4) range from 8.01 (BoardGame) to 51.49 percentage points (PROGpedia-56), indicating strong separation. Statistical tests (Table 5) confirm that improvements are both statistically and practically significant. P-values are low across all datasets, and effect sizes are very large, except for BoardGame, which shows a large effect. While less effective than token sequence normalization, subsequence match merging still offers strong resilience against insertion-based obfuscation – especially notable given its general-purpose nature.

ds	Variant	Pairs	p	W	δ	δ *Int.*	δ 95% CI	n
PROGpedia-19	TSN	P2S	3e-06	378	1.000	Very Large	[1.00, 1.00]	27
	SMM	P2S	3e-06	378	0.904	Very Large	[0.70, 0.97]	27
	Both	P2S	3e-06	378	1.000	Very Large	[1.00, 1.00]	27
PROGpedia-56	TSN	P2S	2e-06	406	1.000	Very Large	[1.00, 1.00]	28
	SMM	P2S	2e-06	406	0.939	Very Large	[0.79, 0.98]	28
	Both	P2S	2e-06	406	1.000	Very Large	[1.00, 1.00]	28
TicTacToe	TSN	P2S	3.9e-10	1,275	1.000	Very Large	[1.00, 1.00]	50
	SMM	P2S	8.4e-10	1,176	0.766	Very Large	[0.60, 0.87]	50
	Both	P2S	3.9e-10	1,275	1.000	Very Large	[1.00, 1.00]	50
BoardGame	TSN	P2S	4.8e-05	210	1.000	Very Large	[1.00, 1.00]	20
	SMM	P2S	4.8e-05	210	0.545	Large	[0.20, 0.77]	20
	Both	P2S	4.8e-05	210	1.000	Very Large	[1.00, 1.00]	20
Homework-1	TSN	P2S	$<$ 1e-10	1,770	1.000	Very Large	[1.00, 1.00]	59
	SMM	P2S	$<$ 1e-10	1,653	0.955	Very Large	[0.82, 0.99]	59
	Both	P2S	$<$ 1e-10	1,770	1.000	Very Large	[1.00, 1.00]	59
Homework-5	TSN	P2S	1.3e-4	170	0.889	Very Large	[0.45, 0.98]	18
	SMM	P2S	1.6e-4	153	0.969	Very Large	[0.83, 0.99]	18
	Both	P2S	1.1e-4	171	1.000	Very Large	[0.99, 1.00]	18

Table 5 One-sided Wilcoxon signed-rank test results for **insertion-based obfuscation** regarding the improvement by our defense mechanism compared to baseline (sig. level of $\alpha = 0.01$, alternative hypothesis $H1 = greater$, test statistic W, effect size via Cliff's delta δ, its interpretation $\delta\,Int.$, its 95 percent confidence interval CI, and the sample size n). For plagiarism-to-source pairs (P2S), low p and high δ are desirable.

Combination of Both

Combining both defense mechanisms yields strong improvements over the baseline, rendering JPlag effectively immune to insertion-based attacks. For Java datasets, results are comparable to subsequence match merging alone, while C++ datasets show clear gains over either individual method. We no longer observe any significant overlap between the plagiarism and original pairs, resulting in a clear separation between both types of pairs. Median similarity *differences* (Table 4) range from 61.22 (Homework-5) to 93.42 percentage points (PROGpedia-19), indicating substantial improvements across all datasets. Statistical tests (Table 5) confirm that these gains are both statistically and practically significant, with low p-values and very large effect sizes for all datasets. In summary, the combination of both mechanisms provides robust resilience against insertion-based obfuscation, especially strengthening detection for C++ programs.

Answer to Q2: *The defense mechanisms significantly increase the resilience against semantic preserving insertion-based obfuscation attacks. The median similarity differences increase, depending on the dataset, up to 99.65 percentage points, thus producing a complete separation of plagiarized and original programs. Thus, the degree of resilience effectively reflects near-immunity to insertion-based attacks.*

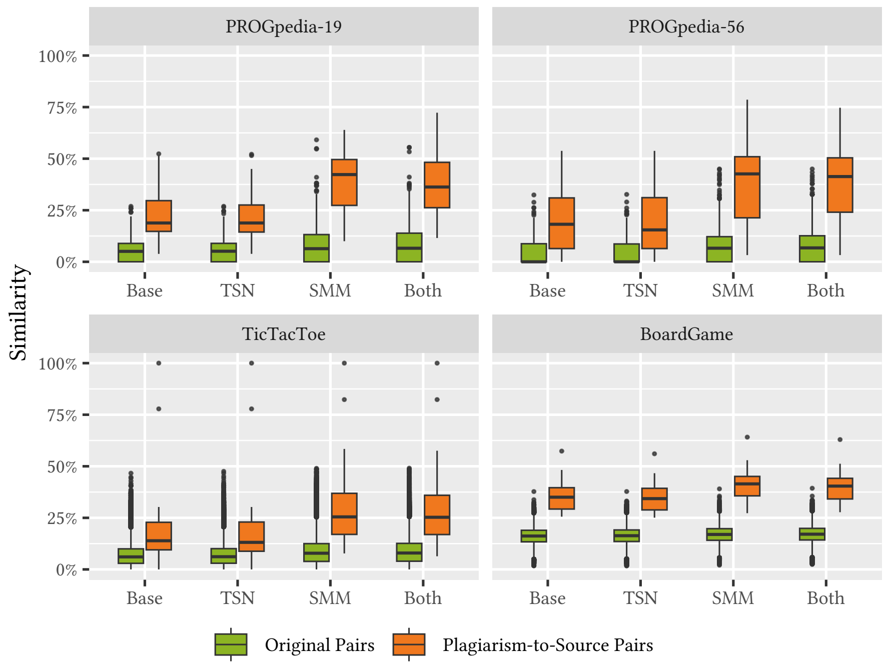

Fig. 6 Similarity scores for original program pairs and **refactoring-based plagiarism** pairs. Ideally, plagiarism pairs exhibit high similarity, while original pairs should exhibit low similarity.

5.3 Refactoring-based Obfuscation

Figure 6 presents results for refactoring-based obfuscation, which applies a mix of semantic-preserving refactorings at random positions in the parse tree of programs. Corresponding statistical measures are shown in Table 6. As the obfuscation tool [40] supports only Java, this evaluation stage includes four of the six datasets.

Baseline

Figure 6 shows the substantial impact of refactoring-based obfuscation on the baseline. Median similarity values for plagiarism pairs drop to between 13.90% (TicTacToe) and 35.02% (BoardGame), leading to clear overlap with original pairs in all datasets except BoardGame, where overlap is limited to outliers. The reduced effect on BoardGame likely stems from the fact that complex obfuscation techniques are harder to apply broadly, reducing the effectiveness for large programs. Median similarity *differences* between plagiarism and original pairs (Table 6) range from 7.83 (TicTacToe) to 18.82 percentage points (BoardGame), indicating only limited separation. Overall, refactoring proves to be an effective obfuscation method against baseline JPlag.

Dataset	Variant	Median	Mean	Q_1	Q_3	$\varDelta$ Mean	$\varDelta$ Median	$\varDelta$ IQR
PROGpedia-19	Base	18.82	23.21	14.74	29.64	17.48	13.76	5.83
	TSN	18.82	22.48	14.43	27.53	16.73	13.70	5.49
	SMM	**42.29**	**38.63**	**27.35**	**49.59**	**29.48**	**35.93**	**14.15**
	Both	36.26	37.15	26.23	48.21	27.93	29.69	12.35
PROGpedia-56	Base	18.20	21.55	6.42	30.97	16.66	18.20	-2.34
	TSN	15.47	20.85	6.43	31.10	16.34	15.47	-2.22
	SMM	**42.62**	**37.02**	21.33	**50.94**	28.76	**36.02**	9.13
	Both	41.32	**37.21**	**24.05**	50.40	**29.09**	34.58	**11.44**
TicTacToe	Base	13.90	17.86	9.49	22.84	11.01	7.83	-0.47
	TSN	13.12	17.52	8.81	22.97	10.58	6.96	-1.26
	SMM	**25.47**	**28.84**	**16.98**	**36.89**	**20.02**	**17.62**	**4.48**
	Both	**25.28**	28.00	**16.91**	35.93	19.08	17.32	**4.28**
BoardGame	Base	35.02	35.76	29.27	39.61	19.58	18.82	10.30
	TSN	34.32	35.20	28.85	39.32	18.87	17.98	9.72
	SMM	**41.47**	**41.37**	**35.67**	**45.09**	**24.42**	**24.52**	**15.92**
	Both	40.40	40.54	34.16	44.16	23.45	23.32	14.26

Table 6 Statistical measures for plagiarism pairs and their differences ($\varDelta$) from original pairs for **refactoring-based obfuscation** (corresponds to Figure 6). Higher values indicate better performance. Note that measures are expressed as percentages and their differences as percentage points. Highest values by a margin of 0.25 are marked in bold.

Dataset	Variant	Pairs	p	W	δ	δ *Int.*	δ 95% CI	n
PROGpedia-19	TSN	P2S	1	11	-0.036	Negligible	[-0.33, 0.26]	27
	SMM	P2S	6.5e-06	325	0.567	Large	[0.28, 0.76]	27
	Both	P2S	5e-06	350	0.510	Large	[0.21, 0.72]	27
PROGpedia-56	TSN	P2S	0.31	108	-0.015	Negligible	[-0.31, 0.28]	28
	SMM	P2S	2.2e-05	253	0.462	Medium	[0.16, 0.68]	28
	Both	P2S	6.5e-06	325	0.473	Medium	[0.17, 0.69]	28
TicTacToe	TSN	P2S	0.98	143	-0.020	Negligible	[-0.24, 0.21]	50
	SMM	P2S	5.7e-10	1,225	0.513	Large	[0.30, 0.68]	50
	Both	P2S	5.7e-10	1,225	0.484	Large	[0.27, 0.65]	50
BoardGame	TSN	P2S	1	14	-0.065	Negligible	[-0.40, 0.28]	20
	SMM	P2S	4.8e-05	210	0.430	Medium	[0.06, 0.70]	20
	Both	P2S	4.8e-05	210	0.380	Medium	[0.01, 0.66]	20

Table 7 One-sided Wilcoxon signed-rank test results for **refactoring-based obfuscation** regarding the improvement by our defense mechanism compared to baseline (sig. level of $\alpha = 0.01$, alternative hypothesis $H1 = greater$, test statistic W, effect size via Cliff's delta δ, its interpretation $\delta\, Int.$, its 95 percent confidence interval CI, and the sample size n). For plagiarism-to-source pairs (P2S), low p and high δ are desirable.

Token Sequence Normalization

Since refactoring-based obfuscation does not involve statement insertion or reordering, token sequence normalization has little to no effect effect – an expected outcome, given its design focus. As shown in Figure 6, the results closely resemble the ones for the baseline. In three of four datasets, the median similarity for plagiarism pairs is slightly lower, with reductions ranging from 0.78 (TicTacToe) to 2.73 percentage points (PROGpedia-56) – a marginal difference. This trend is also reflected in the similarity *differences* between plagiarism and original pairs (Table 6), which range from 6.96 (TicTacToe) to 18.87 percentage points (BoardGame). Statistical tests (Table 7) confirm that these changes are neither statistically nor practically significant. High p-values and near-zero (even negative) effect sizes indicate negligible impact. In summary, token sequence normalization provides no measurable resilience against refactoring-based obfuscation, which is consistent with its intended scope.

Subsequence Match Merging

Subsequence match merging leads to clear improvements over the baseline. As shown in Figure 7, median similarity values for plagiarism pairs increase across all datasets, ranging from 25.47% (TicTacToe) to 42.62% (PROGpedia-56). Overlap with original pairs is reduced, mostly limited to quartile extremes. Median similarity *differences* between plagiarism and original pairs (Table 6) range from 17.62 (TicTacToe) to 36.02 percentage points (PROGpedia-56), indicating stronger separation. Statistical tests (Table 7) confirm statistical and practical significance: p-values are low across all datasets, and effect sizes are medium (BoardGame, PROGpedia-56) to large (TicTacToe, PROGpedia-19). In summary, subsequence match merging provides meaningful resilience against refactoring-based obfuscation, enabling effective detection even when structural changes are introduced through extensive refactorings.

Combination of Both

Combining both defense mechanisms yields strong improvements over the baseline, closely mirroring the results of subsequence match merging alone. As shown in Figure 7, median similarity values for plagiarism pairs range from 25.28% (TicTacToe) to 41.32% (PROGpedia-56), with limited overlap confined to quartile extremes. Median similarity *differences* (Table 6) span from 17.32 (TicTacToe) to 34.58 percentage points (PROGpedia-56), showing solid improvement. Results are slightly lower than those of subsequence match merging alone, except for PROGpedia-56, where the combination performs marginally better. Statistical tests (Table 7) confirm both statistical and practical significance: all p-values are low, with effect sizes ranging from medium (BoardGame, PROGpedia-56) to large (TicTacToe, PROGpedia-19). In summary, the combined defenses offer significant resilience against refactoring-based obfuscation across all datasets, though the added benefit over subsequence match merging alone is limited.

Answer to Q3: *The defense mechanisms significantly increase the resilience against semantic preserving refactoring-based obfuscation attacks. The median similarity differences increase, depending on the dataset, up to 22 percentage points, thus strongly improving the separation between plagiarized and original programs.*

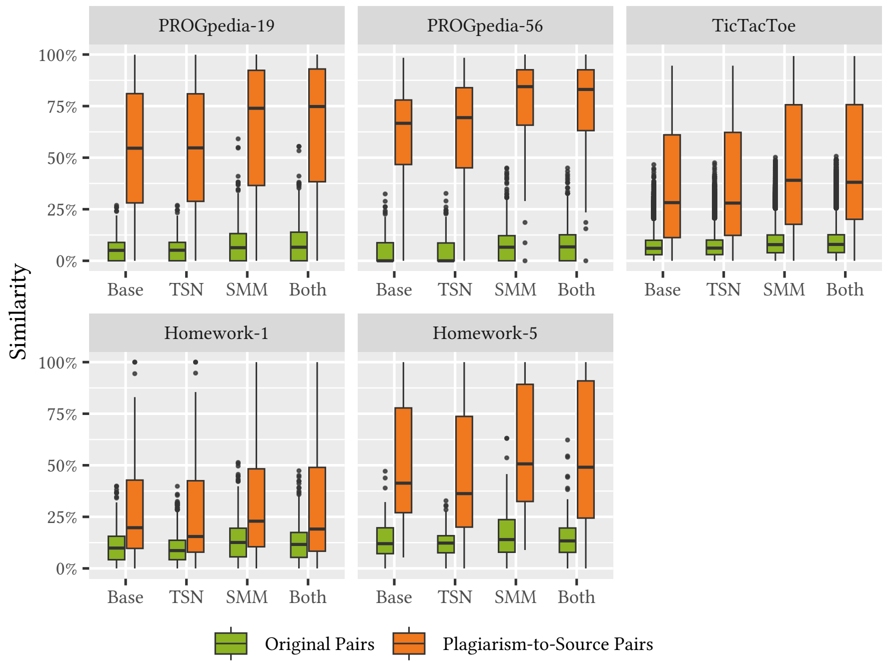

Fig. 7 Similarity scores for original program pairs and **AI-based plagiarism** pairs (obfuscation with 15 varying GPT-4 prompts). Ideally, plagiarism pairs exhibit high similarity, while original pairs should exhibit low similarity.

5.4 GPT-4-based Obfuscation

Figure 7 presents the results for GPT-4-based obfuscation, with corresponding statistical measures in Table 8. We used 15 distinct prompts instructing GPT-4 to alter program code while preserving its functionality. However, as this process lacks formal guarantees, the resulting obfuscation is considered semantic-agnostic. The BoardGame dataset was excluded due to its use in a final exam and the associated privacy concerns with sending data to OpenAI servers.

Baseline

Figure 7 shows that GPT-4-based obfuscation, using 15 behavior-preserving prompts, can be effective against JPlag. Median similarity values for plagiarism pairs drop to between 19.74% (Homework-5) and 66.67% (PROGpedia-56), which is notably a higher range than seen with other obfuscation methods. Overlap with original pairs varies across datasets. Homework-1 shows substantial overlap, including interquartile ranges, while overlap is less pronounced for PROGpedia datasets. As shown in Table 8, median similarity *differences* range from 9.86 (Homework-1) to 66.67 percentage points (PROGpedia-56),

Dataset	Variant	Median	Mean	Q_1	Q_3	Δ Mean	Δ Median	Δ IQR
PROGpedia-19	Base	54.59	54.12	28.06	81.04	48.39	49.53	19.15
	TSN	54.74	54.84	28.82	80.95	49.08	49.61	19.88
	SMM	73.95	**63.46**	36.51	92.34	**54.31**	67.59	23.31
	Both	**74.79**	**63.60**	**38.32**	**92.98**	**54.38**	**68.22**	**24.44**
PROGpedia-56	Base	66.67	61.76	46.67	77.96	56.87	66.67	37.91
	TSN	69.40	63.92	45.05	83.92	59.40	69.40	36.41
	SMM	**84.43**	**75.37**	**65.76**	**92.60**	**67.11**	**77.84**	**53.56**
	Both	83.07	75.10	63.11	**92.59**	**66.98**	76.32	50.50
TicTacToe	Base	28.20	35.60	11.25	61.05	28.75	22.13	1.28
	TSN	27.98	37.50	12.32	62.26	30.56	21.83	2.25
	SMM	**39.02**	44.96	17.68	**75.65**	36.13	**31.18**	5.16
	Both	38.07	**45.53**	**20.11**	**75.70**	**36.61**	30.13	**7.48**
Homework-1	Base	19.74	27.71	9.70	42.81	17.32	9.86	**-5.90**
	TSN	15.47	26.44	7.88	42.50	17.07	6.83	**-5.77**
	SMM	**22.90**	**32.69**	**10.49**	48.27	**19.65**	**10.32**	-9.01
	Both	19.09	31.42	8.35	**48.96**	**19.42**	7.47	-9.08
Homework-5	Base	41.33	50.37	27.02	77.79	36.97	29.34	7.34
	TSN	36.26	46.13	20.03	73.68	34.48	23.98	4.19
	SMM	**50.68**	**57.54**	**32.45**	89.24	**41.62**	**36.70**	**8.79**
	Both	49.06	53.90	24.40	**90.91**	39.31	35.71	4.82

Table 8 Statistical measures for plagiarism pairs and their differences (Δ) from original pairs for **AI-based obfuscation** (corresponds to Figure 7). Higher values indicate better performance. Note that measures are expressed as percentages and their differences as percentage points. Highest values by a margin of 0.25 are marked in bold.

Dataset	Variant	Pairs	p	W	δ	δ *Int.*	δ 95% CI	n
PROGpedia-19	TSN	P2S	0.0027	893	0.019	Negligible	[-0.17, 0.20]	74
	SMM	P2S	< 1e-10	1,485	0.210	Small	[0.02, 0.38]	74
	Both	P2S	< 1e-10	2,216	0.209	Small	[0.02, 0.38]	74
PROGpedia-56	TSN	P2S	0.025	971	0.077	Negligible	[-0.11, 0.26]	75
	SMM	P2S	< 1e-10	1,540	0.371	Medium	[0.19, 0.53]	75
	Both	P2S	< 1e-10	2,136	0.360	Medium	[0.18, 0.52]	75
TicTacToe	TSN	P2S	5.2e-08	1,107	0.042	Negligible	[-0.14, 0.22]	75
	SMM	P2S	< 1e-10	1,770	0.194	Small	[0.01, 0.37]	75
	Both	P2S	< 1e-10	2,342	0.210	Small	[0.03, 0.38]	75
Homework-1	TSN	P2S	0.89	913	-0.046	Negligible	[-0.23, 0.14]	74
	SMM	P2S	2e-06	406	0.075	Negligible	[-0.11, 0.26]	74
	Both	P2S	0.003	1,536	0.024	Negligible	[-0.16, 0.21]	74
Homework-5	TSN	P2S	1	229	-0.115	Negligible	[-0.29, 0.07]	75
	SMM	P2S	2.7e-09	1,035	0.153	Small	[-0.03, 0.33]	75
	Both	P2S	0.11	1,260	0.047	Negligible	[-0.14, 0.23]	75

Table 9 One-sided Wilcoxon signed-rank test results for **AI-based obfuscation** regarding the improvement by of our defense mechanism compared to baseline (sig. level of $\alpha = 0.01$, alternative hypothesis $H1 = greater$, test statistic W, effect size via Cliff's delta δ, its interpretation δ *Int.*, its 95 percent confidence interval CI, and the sample size n). For plagiarism-to-source pairs (P2S), low p and high δ are desirable.

indicating limited separation – particularly for Homework and TicTacToe. Interestingly, the variability in attack effectiveness across prompts is similar to that across datasets. We observe that the dataset itself – likely due to the underlying assignment and domain – has a greater influence on obfuscation effectiveness. Thus, while GPT-based obfuscation is effective, its reliability is lower than that of algorithmic methods due to significant variation in performance across datasets and prompts.

Token Sequence Normalization

Since GPT-4-based obfuscation involves diverse modifications beyond statement insertion or reordering, token sequence normalization has a limited impact. As shown in Figure 7, results are similar to the baseline across all five datasets. Median similarity values for plagiarism pairs vary slightly, ranging from -5.07 (Homework-5) to $+2.37$ percentage points (PROGpedia-56). Corresponding median similarity *differences* with original pairs (Table 8) range from 6.83 (Homework-1) to 69.40 percentage points (PROGpedia-56), aligning closely with baseline values – slightly better for PROGpedia, slightly worse for the Homework datasets, possibly reflecting language-specific differences in GPT-4's output. Statistical tests (Table 9) show some statistical significance (PROGpedia-19, TicTacToe) but no practical significance. Effect sizes are negligible across all datasets, with negative values for the Homework sets. In summary, token sequence normalization offers no meaningful resilience against GPT-4-based obfuscation, though it also introduces no adverse effects.

Subsequence Match Merging

Subsequence match merging significantly improves results over the baseline. As shown in Figure 7, median similarity values for plagiarism pairs increase across all datasets, ranging from 22.90% (Homework-1) to 84.43% (PROGpedia-56). Overlap with original pairs is reduced, primarily limited to quartile extremes. Median similarity *differences* (Table 8) range from 10.32 (Homework-1) to 77.84 percentage points (PROGpedia-56), indicating substantial separation. This confirms that subsequence match merging provides a solid improvement over the baseline. Statistical tests (Table 9) show statistically significant improvements across all datasets. Practical significance is achieved in all but Homework-1, where the effect size remains negligible. Note that the high variance in plagiarism pair similarities affects the effect size measure [23]. In contrast, the remaining datasets show small to medium effect sizes, indicating practical significance. In sum, subsequence match merging offers robust resilience against GPT-4-based obfuscation despite its semantic-agnostic nature and variability across datasets and prompts.

Combination of Both

Combining both defense mechanisms results in strong improvements over the baseline, largely mirroring the effect of subsequence match merging alone. As shown in Figure 7, median similarity values for plagiarism pairs range from 19.09% (Homework-1) to 83.07% (PROGpedia-56), with reduced overlap mostly confined to quartile boundaries. Median similarity *differences* (Table 8) range from 7.47 (Homework-1) to 76.32 percentage points (PROGpedia-56), showing consistent improvement over the baseline. The effect is slightly weaker than with subsequence match merging alone, except for PROGpedia-19, where the combination performs marginally better. Statistical tests (Table 9) confirm statistical significance in all datasets except Homework-5 ($p = 0.11$). Practical significance is observed for all but the Homework datasets, which exhibit small effect sizes. These datasets appear more vulnerable to AI-based obfuscation, possibly due to the small size of these programs or due to the semantic-agnostic nature of the transformation, which may alter the behavior of programs. For the remaining four Java datasets, effect sizes are small to medium, indicating practical significance. As with other AI-based attacks, high variance in similarity scores due to prompt diversity reduces measured effect sizes [23]. In summary, the combined defenses offer significant resilience against AI-based obfuscation for Java datasets, though results are more limited for C++ programs. Yet, the defense mechanisms improve detection despite the potentially disruptive nature of AI-based obfuscation.

Answer to Q4: *The defense mechanisms significantly increase the resilience against semantic agnostic AI-based obfuscation attacks. The median similarity differences increase, depending on the dataset, up to 19 percentage points, thus improving the separation between plagiarized and original programs, albeit to a lesser degree than other attack types.*

5.5 GPT-4-generated Programs

Figure 8 presents results for programs generated by GPT-4 based on assignment descriptions, with corresponding statistical measures in Table 10. Unlike previous stages, this evaluation does not involve obfuscation, as the generated programs are not derived from human-written ones. Instead, we compare the similarity among GPT-4-generated programs to that of unrelated human submissions. While the defense mechanisms are not designed for this setting, they improve the distinction between AI-generated and unrelated human programs. Such a capability can help detect AI-generated submissions if multiple students use the same language model. As with the last stage, BoardGame was excluded for privacy reasons.

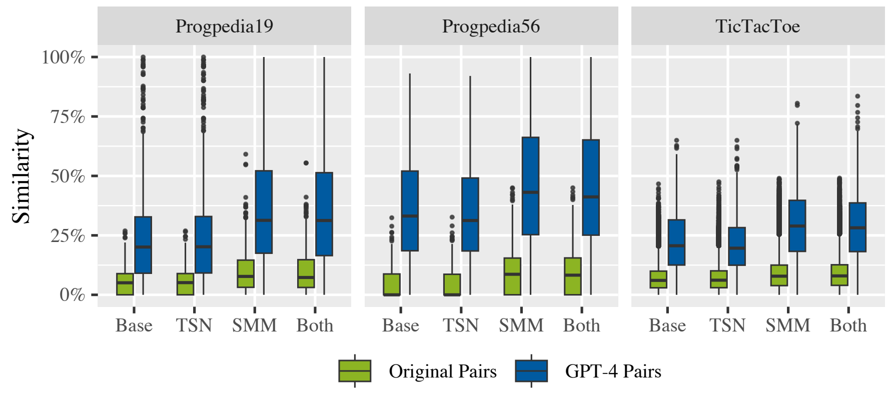

Fig. 8 Similarity scores for original (human) program pairs and pairs of **AI-generated programs** (based on GPT-4 and the assignment description). Ideally, generated pairs exhibit high similarity, while original pairs should exhibit low similarity.

Dataset	Variant	Median	Mean	Q_1	Q_3	Δ Mean	Δ Median	Δ IQR
PROGpedia-19	Base	20.10	22.90	9.09	32.77	17.17	15.04	0.18
	TSN	20.20	23.20	9.18	32.93	17.44	15.07	0.24
	SMM	**31.33**	**36.66**	**17.51**	**52.14**	**26.52**	23.58	**2.92**
	Both	**31.25**	36.33	16.49	51.35	**26.38**	**23.96**	1.73
PROGpedia-56	Base	33.13	35.81	18.53	52.04	30.92	33.13	9.77
	TSN	31.25	34.58	18.45	49.11	30.06	31.25	9.81
	SMM	**43.11**	**45.50**	**25.30**	**66.20**	**35.31**	**34.47**	**9.85**
	Both	41.18	44.99	**25.10**	65.13	**35.24**	32.91	9.59
TicTacToe	Base	20.63	22.53	12.53	31.51	15.67	14.57	2.57
	TSN	19.62	20.72	12.47	28.25	13.78	13.46	2.40
	SMM	**28.94**	**29.60**	**18.27**	**39.72**	**20.77**	**21.10**	**5.75**
	Both	28.18	28.98	**18.18**	38.65	20.06	20.24	**5.55**

Table 10 Statistical measures for plagiarism pairs and their differences (Δ) from original pairs for **AI-based generation** (corresponds to Figure 8). Higher values indicate better performance. Note that measures are expressed as percentages and their differences as percentage points. Highest values by a margin of 0.25 are marked in bold.

Baseline

Figure 8 shows that GPT-4-generated programs, though created from the same assignment prompt, exhibit relatively low similarity: the median similarities among generated pairs are between 20.10% (PROGpedia-19) and 31.33% (PROGpedia-56). This reflects the inherent indeterminism of generative AI. In contrast, however, unrelated human-made programs for the same task have even lower similarities, with median values between 0.00% (PROGpedia-56) and 6.06% (TicTacToe). Thus, even with the baseline, GPT-4-generated programs are significantly more similar to each other than human submissions.

However, some overlap remains. As shown in Table 10, the median similarity *differences* between AI-generated and human program pairs are 14.57

Dataset	Variant	Pairs	p	W	δ	$\delta\,Int.$	δ 95% CI	n
PROGpedia-19	TSN	FG	1.8e-11	4,937	0.007	Negligible	[-0.04, 0.05]	1,225
	SMM	FG	$<$ 1e-10	445,096	0.321	Small	[0.28, 0.36]	1,225
	Both	FG	$<$ 1e-10	407,953	0.310	Small	[0.27, 0.35]	1,225
PROGpedia-56	TSN	FG	1	175,838	-0.033	Negligible	[-0.08, 0.01]	1,225
	SMM	FG	$<$ 1e-10	475,800	0.219	Small	[0.17, 0.26]	1,225
	Both	FG	$<$ 1e-10	636,011	0.207	Small	[0.16, 0.25]	1,225
TicTacToe	TSN	FG	1	120,687	-0.072	Negligible	[-0.12, -0.03]	1,225
	SMM	FG	$<$ 1e-10	334,971	0.271	Small	[0.23, 0.31]	1,225
	Both	FG	$<$ 1e-10	416,672	0.256	Small	[0.21, 0.30]	1,225

Table 11 One-sided Wilcoxon signed-rank test results for **AI-based generation** regarding the improvement by our defense mechanism compared to baseline (sig. level of $\alpha = 0.05$, alternative hypothesis $H1 = greater$, test statistic W, effect size via Cliff's delta δ, its interpretation $\delta\,Int.$, its 95 percent confidence interval CI, and the sample size n). For Fully-Generated Pairs (FG), low p and high δ are desirable.

(TicTacToe), 15.04 (PROGpedia-19) and 33.13 (PROGpedia-56) percentage points, which is comparable to differences observed for refactoring- and alteration-based obfuscation, but notably higher than for insertion-based attacks, which showed a median difference slightly below zero (-0.78). In summary, while generated programs are more alike than human ones, the limited separation leaves room for evasion – highlighting the need for improved detection mechanisms.

Token Sequence Normalization

As token sequence normalization targets statement insertion and reordering, it is not expected to meaningfully affect AI-generated programs, which typically lack dead code and are less impacted by statement order. As shown in Figure 8, results align with expectations: the median similarities among generated pairs match the baseline. Similarly, the median similarity *difference* with unrelated human programs are at similar values as the baseline (Table 10). Statistical tests (Table 11) confirm that the variations are both statistically and practically insignificant. With a p-value of 1 and a near-zero effect size, no meaningful improvement is observed. In summary, token sequence normalization does not enhance the detection of AI-generated programs but also introduces no significant drawbacks.

Subsequence Match Merging

Subsequence match merging yields a clear improvement over the baseline. As shown in Figure 8, the median similarity among generated pairs increases by more than 8.31 (TicTacToe) to 11.23 (PROGpedia-19) percentage points, reducing overlap with human programs – now mostly limited to the upper quartile. This improvement is reflected in the median similarity *difference* between AI-generated and human program pairs, which rises to between 21.10 (TicTacToe) and 34.47 (PROGpedia-56) percentage points (Table 10). Statistical tests

(Table 11) confirm both statistical and practical significance: p-values are low, and the effect size, though small, is meaningful in practice. In summary, subsequence match merging significantly enhances the detection of AI-generated programs – despite not being designed for this purpose – highlighting its versatility as a defense mechanism.

Combination of Both

Combining both defense mechanisms results in a strong improvement over the baseline, closely mirroring the effect of subsequence match merging alone. As shown in Figure 8, the median similarity among generated pairs increases by between 7.55 (TicTacToe) and 11.15 (PROGpedia-19) percentage points, reducing overlap with human submissions – primarily in the upper quartile. The median similarity *difference* between AI-generated and human program pairs rises to between 20.24 (TicTacToe) and 34.47 (PROGpedia-56) percentage points (Table 10). Statistical tests (Table 11) confirm both statistical and practical significance, with low p-values and a small but meaningful effect size. In summary, the combination of both defenses significantly improves the detection of AI-generated programs, performing nearly identically to subsequence match merging alone and introducing no observable drawbacks.

Answer to Q5: *The defense mechanisms, while not designed for this purpose, significantly increase the detection rate of AI-generated programs. The median similarity difference to human programs increases by up to 8.92 percentage points, thus improving the separation between plagiarized and original programs moderately but yet significantly.*

5.6 Threshold-based Obfuscation

In the previous sections, we evaluated each defense variant using identical plagiarism instances, enabling direct comparison. However, threshold-based obfuscation (e.g., MOSSad [18]) dynamically adapts its transformations based on the output of a plagiarism detector, as it obfuscates until the similarity to the original falls below a target threshold. In subsection 5.2, we configured it to use baseline JPlag. This raises the question of how the defenses affect obfuscation when enabled during the threshold-based obfuscation process itself. To investigate this, we selected ten random programs from each Homework dataset and ran MOSSad with all defense variants (MOSSad only supports C/C++ programs). Note that these programs are relatively small (105 LOC on average in Homework-1 and 282 LOC in Homework-5). Experiments were run on a high-performance system (AMD Ryzen 7 7700, 16GB RAM, Arch Linux), representing a realistic upper bound for student hardware. For all programs and variants, MOSSad is configured to terminate when reaching the

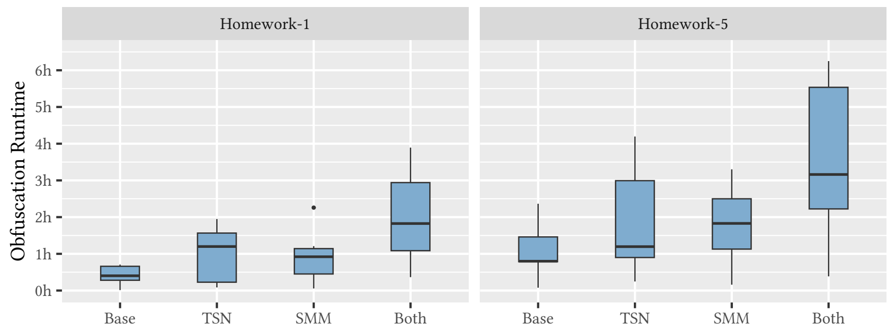

Fig. 9 Required obfuscation duration per program for MOSSad to reach an obfuscation threshold of 25 percent (for programs with original sizes of ˜105 LOC for Hw.-1 and ˜123 LOC for Hw.-5).

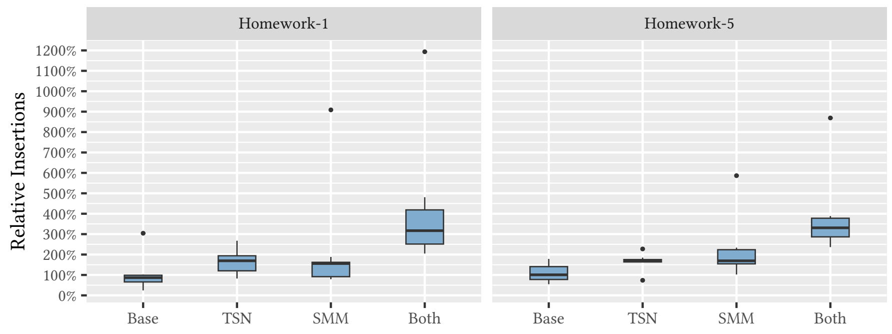

Fig. 10 Required relative insertion of statements for MOSSad to reach a 25 percent obfuscation threshold (relative insertions compared to the original program size to normalize for program size).

25 percent similarity to the original. We observed a notable increase in obfuscation time and program size when using MOSSad to evade the defense mechanisms.

Figure 9 shows a substantial increase in runtime when defenses are active. For the baseline, median obfuscation times were 24 minutes (Homework-1) and 48 minutes (Homework-5). With token sequence normalization (TSN), runtime increased to 72 minutes across both datasets. With subsequence match merging (SMM), runtimes rose to 55 minutes (Homework-1) and 110 minutes (Homework-5). Combining both defenses further increased runtimes to 110 minutes and 190 minutes, respectively, with observed maxima reaching 234 and 375 minutes. These results show that the defenses impose significant computational overhead even on small programs, raising the effort required to successfully obfuscate plagiarism.

Figure 10 shows the relative growth in program size. For the baseline, the median size increased by 87% (Homework-1) and 101% (Homework-5), nearly doubling the original code. With TSN, insertions rose to 169%, while SMM required 156% (Homework-1) and 169% (Homework-5). When both defenses were enabled, median increases reached 317% and 331%, respectively. Maximum observed increases were 1193% and 869%, and all obfuscated programs exceeded 200% of the original size. Such extreme growth makes obfuscated code highly conspicuous and easily identifiable through token count, manual inspection, or outlier detection. Consequently, it makes threshold-based obfuscation apparent, thus turning it into an ineffective obfuscation strategy.

Despite the high-performance system and small input programs, obfuscating just 20 programs across all variants took over 125 hours. Larger programs or slower systems would require days per attempt. Importantly, we configured MOSSad to stop at a 25% similarity threshold (to restrict then overall computation time), while unrelated program pairs typically fall around 10–15%. Achieving lower similarity – and thus avoiding detection entirely – would require even more aggressive obfuscation. Overall, our contributions substantially enhance obfuscation resilience, making threshold-based obfuscation highly time-consuming and resulting in plagiarized solutions that are exceptionally conspicuous due to their size. These factors collectively act as strong deterrents against obfuscation-based plagiarism, making the obfuscation efforts more tedious than completing the actual assignment.

Answer to Q5: *The defense mechanisms strongly increase the computational cost for threshold-based plagiarism, thus resulting in an obfuscation time of up to 6 hours per program and up to 1300 percent increase in program size, making threshold-based plagiarism more tedious and easily detectable.*

6 Threats to Validity

We now discuss how we address threats to the validity of our evaluation, following the guidelines outlined by Wohlin et al. [72] and Runeson and Höst [58].

Internal Validity

Baseline Consistency: For internal validity, we used JPlag as a baseline but also implemented the defense mechanism for JPlag, ensuring that all other conditions remained constant when comparing the defense mechanism with each other or with the baseline. *Handling of invalid programs:* Some public datasets contain invalid or incomplete programs (e.g., programs that do not compile), which could lead to inaccurate results if not properly handled. We addressed this by preprocessing the datasets and removing programs that do not compile. *Validity of the Labeling:* Public datasets often contain incomplete or biased plagiarism labels. This issue does not affect our evaluation, as all plagiarism instances are generated through controlled automated obfuscation.

As a preprocessing step, we carefully filtered out instances of human plagiarism based on the labels, analyzed them with JPlag, and performed human inspections.

External Validity

Generalizability across datasets: Our evaluation uses real-world student submissions from diverse university courses, covering two programming languages and varying assignment sizes. This reflects typical software plagiarism detection scenarios and supports a representative, generalizable assessment [45]. *Generalizability of obfuscation attacks:* Limiting the evaluation to only a few types of obfuscation attacks could hinder the applicability of our results to broader contexts. To enhance external validity and thus ensure that our findings are generalizable, we included a diverse set of real-world obfuscation techniques. *Influence of Prompt Quality:* To address the impact of prompt choice for AI-based obfuscation, we performed systematic "*prompt-engineering*" prior to the evaluation. We then evaluated with 15 suitable prompts. We generated multiple plagiarism instances for each prompt, which we repeated for multiple datasets. While the impact of the prompt varies, the variation is not strong enough to obscure the overall trend, supporting the generalizability of our results.

Construct Validity

Evaluation Methodology Alignment: To enhance construct validity, we aligned our evaluation methodology with those from established and related research works. Moreover, we employ an approach-independent ground truth, and use established similarity metrics. *Underlying Research Object:* Our measurements align directly with the research objective of evaluating detector resilience against automated obfuscation. We use similarity scores from the detectors as primary measurements and assess obfuscation using real-world tools like MOSSad and GPT-4. *Choice of Baseline:* The baseline selection might affect the comparison and outcomes. We selected JPlag as a baseline, as other widely used tools are either not applicable to all datasets, closed-source, or provide restricted results. JPlag is widely recognized as a state-of-the-art tool [5, 45], ensuring that the comparison is relevant and accurate. It operates similarly to other widely used tools by employing standard similarity metrics.

Reliability

To ensure reliability, we provide a comprehensive reproduction package for our evaluation [59]. *Use of Internal Datasets:* Using internal datasets can hinder reproducibility. To enhance reliability, we used both public and internal datasets, balancing generalizability with the need for open data where possible. We discussed any preprocessing steps and the employed obfuscation attacks for all datasets. For the internal datasets (TicTacToe and BoardGame), we provide raw results and metadata in our replication package. *Publishing of Obfuscation Attacks:* The obfuscation attacks utilized in our study can be considered malware, which restricts our ability to provide access to these tools. The exception is GPT-4 [1], which is publicly available; however, we do not provide a detailed, step-by-step guide on exploiting it for plagiarism detection. While omitting these artifacts or details may hinder reproducibility, balancing this limitation with ethical considerations and the responsibility regarding potential misuse.

7 Discussion

In the following, we discuss the interpretation of the evaluation results and highlight key takeaways for software plagiarism detection. Our evaluation highlights the effectiveness of automated obfuscation techniques against plagiarism detectors *without* defense mechanisms. Insertion-based obfuscation proves especially effective, fully concealing plagiarism by adding semantically irrelevant code. Refactoring-based obfuscation also poses a substantial challenge, as structural changes that preserve behavior significantly reduce similarity, limiting the detector's ability to identify plagiarism instances.

AI-based obfuscation introduces significant variability, with its effectiveness depending more on dataset characteristics than on prompt or language. While powerful, its reliability is lower than that of algorithmic methods. However, due to the rapid advancements of generative AI, AI-based poses a growing challenge to detection systems. Currently, AI-based program generation is only effective for smaller programs (below 300 to 400 LOC). Our results show that AI-generated programs exhibit increased similarity to each other compared to human-written programs, aiding detection when multiple students use the same model. Although not designed for this setting, subsequence match merging improves the separation of AI-generated from human-written programs.

7.1 On Providing Broad Obfuscation Resilience

Our evaluation shows that our approach improves obfuscation resilience for all employed attacks and datasets. As expected, the *degree* of those improvements depends on the type of obfuscation attack. Nevertheless, when employing the defense mechanisms, we demonstrate that the provided resilience is not limited to a specific obfuscation attack. We demonstrated effectiveness against a wide-range of obfuscation attacks, including both algorithmic and AI-based attacks, encompassing semantic-preserving and semantic-agnostic obfuscation. Furthermore, we evaluate datasets across different programming languages, in addition to diverse assignment types and sizes, thus demonstrating its adaptability. In total, we use six datasets in combination with five distinct obfuscation attack types. Moreover, each attack type involves various modifications. For example, refactoring-based obfuscation includes multiple transformation types, while AI-based obfuscation involves 15 varying prompts to generate diverse plagiarism instances.

Our evaluation showed that our contributions provide broad resilience against automated obfuscation attacks on programming assignments by systematically covering these different categories of obfuscation attacks. The smallest improvement was observed for AI-based obfuscation, which is expected, given that this is a semantic-agnostic attack using highly challenging prompts, including partial implementations. Detecting partial implementations is particularly difficult for plagiarism detectors as they must carefully balance between detecting re-implementation and avoiding false positives. On the other hand, the strongest improvement was observed for structural attacks, which is a significant result. Structural attacks are among the easiest to automate, even with traditional methods, and they tend to consistently affect plagiarism detectors. Thus, improving resilience in this area is crucial for the effectiveness of detection tools.

7.2 Outliers and Remaining Overlap

Except for insertion-based obfuscation (see Figure 5), where the defense mechanisms completely eliminate *any* overlap between plagiarism instances and unrelated programs, the evaluation results demonstrate minor overlap. This raises an important question regarding the expectations one should have concerning the quality of plagiarism detection. In practical terms, some overlap among outliers is not a significant concern. It is essential to recognize that no plagiarism detection tool is perfect. Thus, educators must accept that human inspection is always the final step in plagiarism detection and that no one should solely rely on the results of an automated tool without first verifying the flagged candidates themselves.

Furthermore, it is crucial to note that plagiarism detectors compare pairs of programs, and thus, a single program might be included in multiple comparisons. This means that detecting every plagiarism pair is not necessary to identify all students involved in plagiarism. In practice, educators would be presented with a ranked list of suspicious pairs, including unrelated and plagiarism pairs. For example, in our evaluation of GPT-4-generated programs, it is not necessary to identify all 1,225 pairs of AI-generated programs to detect each of the 50 generated programs at least once. Notably, when both of the defense mechanisms are enabled (*Both* in Figure 8), only the first 158 pairs (which is the top 0.07 percent out of all 220,780 analyzed pairs) need to be inspected to successfully identify 90 percent of the AI-generated programs at least once. To detect all 50 AI-generated programs, the first 711 pairs need to be checked, which is the top 0.3 percent of all pairs. This underscores that a slight overlap between the pairs of unrelated programs and the plagiarism pairs is not a cause for concern.

Ultimately, it is important to emphasize that no plagiarism detection tool can provide 100 percent certainty. Therefore, human inspection and informed decision-making are essential in ensuring fair and accurate misconduct investigations. Educators must *always* engage in thoughtful analysis of the results generated by these tools to discern genuine cases of plagiarism from false positives effectively.

7.3 AI-based Plagiarism

AI-based attacks [6], particularly those utilizing generative AI, present a growing concern for plagiarism detection. We discussed two possible scenarios when employing generative AI to cheat on programming assignments. *Automatic obfuscation* of an existing solution and *fully generating* solutions from the assignment description. Based on our evaluation results, automatic obfuscation is *currently* the more effective approach for medium and larger assignments, as fully generating only works well for smaller programs. Generated programs do not fulfill necessary functional requirements (not implementing the required behavior precisely) and even non-functional requirements like code style, thus requiring significant manual effort to improve them sufficiently. Automatic obfuscation resembles human obfuscation practices, as a pre-existing solution is altered while *trying* to preserve the program behavior. For both approaches, the defense mechanisms have shown improved resilience.

For AI-generated solutions, there's an ongoing debate on whether this form of cheating qualifies as plagiarism [45, 61]. Our approach improves the detection rate by helping to recognize the similarities among generated solutions that occur due to the semi-deterministic nature of large language models. This improvement is surprising, as the defense mechanisms are *not* designed to detect AI-generated programs.

7.3.1 On the Effectiveness of AI-based Obfuscation

Our evaluation results show that the effectiveness of the defense mechanisms for AI-based obfuscation is less pronounced compared to their performance against algorithmic attacks. This can be attributed to two key factors. First, the overall varying effectiveness of AI-based obfuscation plays a significant role. Our results indicate a strong variance in the similarity values achieved by AI-based obfuscation. While part of this variability can be explained by the different prompts used in our evaluation, this trend remains consistent even when examining the results for each prompt individually. For plagiarized programs that already exhibit a high degree of similarity to their original versions, there is limited potential and necessity for the defense mechanisms to increase that any further. Second, generative AI employs a much broader range of modifications compared to algorithmic obfuscation techniques. Algorithmic methods typically rely on a well-defined, limited set of changes during obfuscation. Even refactoring-based obfuscation, which involves multiple refactoring operations, operates within a constrained set of transformations. In contrast, AI-based obfuscation introduces a far more diverse range of modifications, even when using the same prompt. In our evaluation, we observed strong variations in the types of changes applied by AI depending on both the prompt used and the dataset involved. These diverse modifications alter token sequences extensively, posing a challenge to the defense mechanisms.

Nonetheless, it is important to note that our evaluation still shows a notable improvement in resilience against AI-based obfuscation, even in the presence of these complex and varied changes. This demonstrates that while AI obfuscation is an effective technique, the defense mechanisms mitigate its effects. Interestingly, the effectiveness of AI-based obfuscation attacks strongly varies depending on the dataset used. As illustrated in Figure 7, AI-based obfuscation performs well for Homework-1, while it does not perform well for both PROGpedia datasets. TicTacToe and Homework-5 achieve mixed results. The median similarity differences range, depending on the dataset, between around ten and around 78 percentage points (see Table 8). Although the evaluated plagiarism instances proved to be effective, the process of generating them was not straightforward. In some cases, GPT-4 produces incomplete or invalid code. 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* can be just as effective. However, AI-based obfuscation is more useful in avoiding detection during manual inspection, as it produces diverse modifications and can imitate human-made code.

7.3.2 On the Effectiveness of AI-based Generation

While AI-based generation works to a certain extent, its effectiveness is *currently* limited. The programs generated entirely by GPT-4 did not fully comply with the specific requirements of the programming assignments, often resulting in additional output or slightly altered behavior. These discrepancies suggest that fully AI-generated solutions may only be suitable for smaller, less complex assignments. In our case, the TicTacToe dataset, with an average size of 236 lines of code, appears to be near the threshold where fully generated solutions start to exhibit these inconsistencies.

A noteworthy observation is that AI-generated programs are typically shorter than those created by human developers, especially within the TicTacToe dataset. This reduction in length may contribute to the higher degree of similarity observed between AI-generated solutions. While large language models like GPT-4 are not entirely deterministic, they exhibit a level of determinism sufficient for software plagiarism detection purposes. This inherent determinism, coupled with the more concise code produced by AI, may explain why AI-generated programs tend to resemble each other more closely than human-generated ones. Finally, GPT-4 tends to produce placeholder comments instead of fully implementing certain methods, particularly when the task or method is not well-defined in the prompt. This behavior further limits the effectiveness of AI-based generation for complex assignments, as these incomplete implementations require additional manual intervention to complete.

7.3.3 Emerging Threats

While our results thus show that our contributions can effectively address *current* threats of artificial intelligence, rapid advancements in this field may necessitate future re-evaluation. In the future, AI-based obfuscation methods may exhibit less variance in their effectiveness, thus increasing their reliability. Similarly, new algorithmic attacks might emerge. However, as discussed, all emerging attacks must affect the same attack surface, namely, the internal, linear program representation. Thus, subsequence match merging will continue to provide resilience to emerging attacks. However, the degree of that resilience remains to be assessed.

The rapid development in the field of generative AI may lead to emerging threats that warrant close attention [34]. One area of particular concern is AI-generated programs. As generative AI advances, this might become feasible for larger programs and produce functionally correct programs for more complex assignments. To detect such fully generated programs, detection systems need capabilities to detect obfuscation via implementation, which can be considered semantic clones. Here, caution is warranted. While matching full re-implementations seems desirable, it risks introducing significant false positives by flagging unrelated programs created independently by students. Note that unrelated solutions to a single problem can also be seen as semantic clones. Thus, we see the danger of creating unreliable detection systems, which may lead to unfairly penalizing students. Addressing re-implementation or semantic clones, therefore, raises philosophical questions about the boundaries of what type of plagiarism we actually want a detection system to target.

For fully generated programs, for example, via generative AI, plagiarism detection methods may not be sufficient for emerging attacks. If traditional plagiarism detection methods, including the defense mechanisms evaluated in this paper, prove inadequate against more sophisticated AI-generated code, alternative techniques may need to be explored [31]. One research area is the development of AI-based detectors that act as countermeasures to generative AI. However, at present, such AI-based detectors have not demonstrated sufficient reliability or performance, and they remain an area of ongoing research [71, 49, 32]. Another possibility lies in signature- or watermark-based methods, where the artifacts generated by AI are always identifiable as such. This approach would involve recognizing specific patterns or characteristics inherent to AI-generated content, allowing for consistent identification, regardless of the obfuscation techniques applied. Again, this is ongoing research [76, 24]. It is important to note, however, that these potential future developments lie beyond the scope of this paper and even outside of the research area of software plagiarism detection.

7.4 Layering Defense Mechanisms

Attack-specific defense mechanisms are highly effective, as they can be tailored with strong assumptions about specific obfuscation techniques in mind. For their targeted obfuscation attacks, attack-specific mechanisms outperform attack-independent approaches. This is evident in the case of token sequence normalization in Figure 5, where the defense mechanism fully separates plagiarism pairs from original pairs, completely outperforming subsequence match merging. However, attack-specific mechanisms mostly focus solely on a single known obfuscation attack type. Multiple attack-specific mechanisms must be combined to achieve broad resilience. Additionally, attack-specific mechanisms can only be designed for known attacks and may not be equipped to handle emerging threats, as they rely on assumptions that may not hold true for unknown obfuscation techniques. Attack-independent mechanisms, such as subsequence match merging, make fewer assumptions about the obfuscation techniques in use. Thus, they provide less resilience for a given obfuscation attack. Their strength, however, lies in providing broad resilience. Throughout our evaluation, we observed that subsequence match merging consistently offered resilience across a variety of obfuscation attacks. Because of its heuristic nature and the fact that it operates at a high level of abstraction, it can provide *some* resilience against unknown and emerging obfuscation attacks. Attack-independent approaches are essential for defending against emerging threats. Since they make fewer assumptions about the nature of incoming obfuscation attacks, they offer a level of protection against unknown attacks that attack-specific mechanisms may not.

The ideal solution is to combine multiple defense mechanisms, leveraging both attack-specific and attack-independent defense mechanisms. This strategy provides targeted resilience against well-known or highly effective obfuscation attacks while also offering broad protection against unknown or emerging techniques. Layering multiple defenses is a well-established strategy in information security and risk assessment, often referred to as the *Swiss cheese model* [55] or *defense in depth* [66, 36, 4]. When using this layered approach, it is critical to ensure compatibility between defense mechanisms to avoid unintended side effects that could reduce overall obfuscation resilience or detection quality. In the context of software plagiarism detection, it is beneficial to allow users to enable or disable different defense mechanisms depending on their needs or to mitigate potential side effects. The defense mechanisms are designed to be minimally intrusive, enabling them to be layered with other approaches. In our evaluation, we examine the combination of defense mechanisms to check for adverse side effects. While in some cases, individual mechanisms outperformed the combination, the difference from the second-best mechanism is always negligible. Therefore, our mechanisms can be safely used in a layered defense strategy.

8 Related Work

This section discusses research from areas intersecting with this paper's contributions.

8.1 Software Plagiarism Detection Systems

Despite its early roots [47], research in software plagiarism detection has seen a resurgence in recent years Novak et al. [45]. Most software plagiarism detection approaches compare the structure of the code [43, 45]; among them, token-based approaches are the most popular tools employed in practice. JPlag [53] and MOSS [2] are the most widely used tools [5, 45]. Furthermore, JPlag is most frequently referenced and compared to [45]. Other tools mentioned frequently are Sherlock [25] and SIM [22]. However, they are partially outdated or no longer maintained. Dolos [39] is a more recent tool inspired by MOSS and JPlag but currently supports only single-file programs, which limits its applicability. All mentioned approaches are token-based and find matching fragments via hashing and tiling [54, 2]. Some recent approaches also employ machine learning for plagiarism detection [19]. We use JPlag as a baseline in our research; however, the approaches extend to any token-based detector and can be generalized to structure-based methods. *The mentioned works evaluate their tools with manually-obfuscated plagiarism. We specifically focus on evaluating automated obfuscation.*

8.2 Obfuscation Attacks and Their Mitigation

Obfuscation attacks present a significant challenge for software plagiarism detection. While obfuscation has long been a concern [75, 27, 45], research on defending existing state-of-the-art plagiarism detection tools from automated obfuscation is limited. Most recent studies focus on developing entirely new detection systems that often remain inaccessible to the public, as noted by Novak et al. [45]. Research on mitigating obfuscation usually focuses on manual obfuscation. In the following, we discuss notable exceptions. Devore-McDonald and Berger [18] introduce MOSSad, a tool that uses genetic programming techniques to automatically generate semantically equivalent but undetectable plagiarized code variants, defeating detectors like Moss and JPlag. Its non-deterministic transformations mimic authentic student submissions. Biderman and Raff [6] show that language models like *GPT-J* can produce correct, syntactically diverse solutions that evade Moss detection with minimal human input, raising concerns for academic integrity as AI tools become more accessible. Similarly, Karnalim et al. [29] evaluate 16 preprocessing techniques for source code similarity detection, finding that methods like identifier removal and syntax tree linearization improve detection effectiveness. However, such techniques offer limited resilience against broader obfuscation strategies. *These* *works typically focus on individual obfuscation attacks. In contrast, our work addresses a broader spectrum of automated obfuscation strategies and shifts the focus from attack feasibility to evaluating concrete defense mechanisms.*

8.3 Generative AI in Programming Education

Chen et al. [10] investigate the impact of generative AI on academic integrity in an introductory programming course. They show that suspected plagiarism increased and shifted from traditional sources to AI tools. The results of their regression suggests that increased plagiarism may lead to decreased learning outcomes. In contrast to these results, other studies observe no difference. Xue et al. [73] conduct a controlled study on ChatGPT's impact in CS1 programming education with 56 participants. Their results showed no significant difference in learning outcomes between groups. Most students held neutral views but expressed concerns about ethical issues and ChatGPT's inconsistent results. Choudhuri et al. [11] explore the impact of conversational generative AI on supporting students in software engineering tasks. Their study with 22 participants found no significant difference in productivity or self-efficacy compared to traditional resources, but noted significantly higher frustration levels. Karnalim [28] investigate student perceptions of AI-assisted plagiarism in programming education by comparing it to traditional plagiarism scenarios. Based on survey responses from 66 introductory and intermediate programming students, the study finds that students view AI assistance as morally comparable to help from peers. The study suggests that student awareness and interpretation of AI-assisted plagiarism vary by experience level. Cipriano and Alves [12] examine the performance of large language models in object-oriented programming (OOP) exercises. Using real-world educational tasks and automatic assessment tools, they found that while LLMs often produce working solutions, they frequently neglect OOP best practices. The study highlights the need to emphasize code quality in programming education. *Generative AI is becoming an integral part of programming education, and as educators, we have to deal with its impact. For this reason, this paper specifically investigates AI-based obfuscation.*

8.4 Detecting AI-Generated code

Karnalim et al. [31] propose a lightweight AI-assisted code detector based on code anomaly features, which uses 34 features spanning various program elements to identify unusual patterns that may indicate AI assistance. Evaluated across three datasets, their approach shows promising results. However, the detection effectiveness drops significantly when students collaborate or use AI only partially. Orenstrakh et al. [46] evaluate the effectiveness of eight publicly available detectors for identifying LLM-generated content. They collected 124 human-written student submissions from before ChatGPT and compared them with 40 ChatGPT-generated samples. They find that detection accuracy significantly drops for programming code, non-English text, and content modified with paraphrasing tools, highlighting current limitations of such detectors. Similarly, Suh et al. [67] investigate the challenge of detecting AI-generated code, noting that current detection tools perform poorly and lack generalizability. To address this, they propose enhanced approaches such as fine-tuning LLMs and using machine learning classifiers with static code metrics or AST-based embeddings. Their best model outperforms GPTSniffer, achieving an F1 score of 82.55. Moreover, Pan et al. [49] examine the ability of ChatGPT to evade detectors for AI-generated content in programming education. Using a dataset of 5,069 human-written Python solutions, they prompted ChatGPT with 13 code problem variants and evaluated five detectors. Results show that current detectors struggle to distinguish AI-generated code from human-written code reliably. *Given that AI code detectors are unreliable, it is especially relevant that software plagiarism detectors can produce suspiciously high similarity values for AI-generated programs produced by the same model. We analyze this effect in our evaluation.*

8.5 Clone Detection

Reusing source code via copying commonly leads to code clones [57], which impedes modern software development [26]. Code clones are created accidentally [26], while plagiarism is a deliberate act. While both clone detection and plagiarism detection are software similarity problems [45], they ultimately differ in many aspects [41]. In contrast, code clone detection does not consider scenarios where an adversary attempts to affect the process, as code clones typically arise inadvertently [26]. As a consequence, clone detectors are vulnerable to obfuscation attacks. Plagiarism detection approaches must deal with an additional layer of complexity introduced by the adversary-defender-scenario [60]. Still, many clone detection approaches share similarities in their employed techniques [69]. *In summary, while clone detection is a related field, these works are not applicable for automated obfuscation.*

9 Conclusion

In this paper, we analyze the state of plagiarism detection by evaluating defense mechanisms for plagiarism detection systems that provide resilience against automated obfuscation attacks. The defense mechanisms include both a tailored approach that targets specific obfuscation types and a broad, attack-independent mechanism that ensures resilience against a wide range of obfuscation strategies, including emerging and unknown threats. Our evaluation shows that the defense mechanisms demonstrate significant improvements in detection rates across a wide variety of obfuscation methods, achieving significant gains in similarity scores even under AI-based obfuscation. In conclusion, this paper offers key insights into emerging challenges in academic integrity. By contributing empirical insights and discussion about the current state of plagiarism detection, we support educators in maintaining integrity across programming assignments in a landscape increasingly shaped by automated obfuscation [21, 6].

Despite these advancements, it would be inadvisable to consider the problem of automated obfuscation in software plagiarism as solved. The rapid development of generative AI continues to reshape education. Educators and researchers must learn to adapt to these developments to anticipate possible implications on academic integrity. Addressing AI-based obfuscation attacks will require even greater focus in future investigations. Nevertheless, artificial intelligence should not be viewed as an all-encompassing threat; instead, it creates new challenges *and* opportunities. Critically, we should focus on understanding its limitations and leveraging its strengths [64]. At the same time, it is essential to complement technical efforts with an open educational and ethical discourse on what constitutes plagiarism in the age of AI. Detection tools can identify suspicious patterns, but they cannot define the boundaries of misconduct – this remains a fundamentally human and pedagogical responsibility.

Plagiarism detection systems excel at identifying structural similarities, while humans are uniquely capable of quickly identifying semantic nuances between programs given these similarities. This combination of large-scale analysis and human inspection has thus been very effective and will remain so for the foreseeable future. Fundamentally, however, tool-based plagiarism detection must always involve a human decision as a final step. In addition to detection, prevention is just as important in combating plagiarism [65]. Students often resort to plagiarism when overwhelmed and believe they have no other options [3]. Proactive strategies such as student training, thoughtful assessment design, clear institutional policies, and counseling should thus be combined with detection approaches [34].

Acknowledgements This work is supported by the CHAINS project funded by the Swedish Foundation for Strategic Research (SSF).

Declarations

The authors declare that they have no conflict of interest. All data and code related to this paper is available at [59].

References

1. Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, et al. Gpt-4 technical report. Technical report, OpenAI, 2024.
2. Alex Aiken. Moss software plagiarism detector website, 7 2022. URL `http://theory.stanford.edu/~aiken/moss/`.

3. Alexander Amigud and Thomas Lancaster. 246 reasons to cheat: An analysis of students' reasons for seeking to outsource academic work. *Computers & Education*, 134:98–107, 6 2019. ISSN 0360-1315. doi:https://doi.org/10.1016/j.compedu.2019.01.017.
4. Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, Third Edition*. John Wiley & Sons Inc., United States, 3 edition, 12 2020. ISBN 9781119642787. doi:10.1002/9781119644682.
5. Rodrigo C Aniceto, Maristela Holanda, Carla Castanho, and Dilma Da Silva. Source code plagiarism detection in an educational context: A literature mapping. In *2021 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 10 2021. doi:10.1109/FIE49875.2021.9637155.
6. Stella Biderman and Edward Raff. Fooling moss detection with pretrained language models. In Mohammad Al Hasan and Li Xiong 0001, editors, *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, CIKM '22, page 2933–2943, NY, USA, 10 2022. ACM. ISBN 9781450392365. doi:10.1145/3511808.3557079.
7. Bear F. Braumoeller and Brian J. Gaines. Actions do speak louder than words: Deterring plagiarism with the use of plagiarism-detection software. *PS: Political Science and Politics*, 34(4):835–839, 10 2001. ISSN 1049-0965. doi:10.1017/s1049096501000786.
8. Moritz Brödel. *Preventing Automatic Code Plagiarism Generation Through Token String Normalization*. bachelor's thesis, Karlsruhe Institute of Technology (KIT), 2023.
9. Tracy Camp, W. Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. Generation cs: The growth of computer science. *ACM Inroads*, 8(2):44–50, 5 2017. ISSN 2153-2184. doi:10.1145/3084362.
10. Binglin Chen, Colleen M. Lewis, Matthew West, and Craig Zilles. Plagiarism in the age of generative ai: Cheating method change and learning loss in an intro to cs course. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale*, L@S '24, page 75–85, NY, USA, 2024. ACM. ISBN 9798400706332. doi:10.1145/3657604.3662046.
11. Rudrajit Choudhuri, Dylan Liu, Igor Steinmacher, Marco Gerosa, and Anita Sarma. How far are we? the triumphs and trials of generative ai in learning software engineering. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, page 1–13. ACM, 4 2024. doi:10.1145/3597503.3639201.
12. Bruno Pereira Cipriano and Pedro Alves. Llms still can't avoid instanceof: An investigation into gpt-3.5, gpt-4 and bard's capacity to handle object-oriented programming assignments. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '24, page 162–169, NY, USA, 2024. ACM. ISBN 9798400704987. doi:10.1145/3639474.3640052.
13. Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509, 11 1993. ISSN 0033-2909. doi:10.1037/0033-2909.114.3.494.
14. Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Routledge, 5 1988. ISBN 9781134742707. doi:10.4324/9780203771587. First published: 1988.
15. Georgina Cosma and Mike Joy. Towards a definition of source-code plagiarism. *IEEE Transactions on Education*, 51(2):195–200, 5 2008. ISSN 0018-9359. doi:10.1109/te.2007.906776.
16. Fintan Culwin and Thomas Lancaster. Plagiarism issues for higher education. *VINE*, 31(2):36–41, 1 2001. ISSN 0305-5728. doi:10.1108/03055720010804005.
17. Marian Daun and Jennifer Brings. How chatgpt will change software engineering education. In Mikko-Jussi Laakso, Mattia Monga, Simon, and Judithe Sheard, editors, *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, page 110–116, NY, USA, 6 2023. ACM. ISBN 979-8-4007-0138-2. doi:10.1145/3587102.3588815.
18. Breanna Devore-McDonald and Emery D. Berger. Mossad: Defeating software plagiarism detection. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 1–28, 11 2020. ISSN 2475-1421. doi:10.1145/3428206.
19. Fahad Ebrahim and Mike Joy. Semantic similarity search for source code plagiarism detection: An exploratory study. In Mattia Monga, Violetta Lonati, Erik Barendsen, Judithe Sheard, and James Paterson, editors, *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2024, page 360–366, NY,

USA, 7 2024. ACM. ISBN 9798400706004. doi:10.1145/3649217.3653622.

20. J.A.W. Faidhi and S.K. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, 11(1):11–19, 1 1987. ISSN 0360-1315. doi:10.1016/0360-1315(87)90042-x.
21. Tomáš Foltýnek, Terry Ruas, Philipp Scharpf, Norman Meuschke, Moritz Schubotz, William Grosky, and Bela Gipp. Detecting machine-obfuscated plagiarism. In Anneli Sundqvist, Gerd Berget, Jan Nolin, and Kjell Ivar Skjerdingstad, editors, *Sustainable Digital Communities*, volume 12051, pages 816–827, Cham, 3 2020. Springer. ISBN 978-3-030-43687-2. doi:10.1007/978-3-030-43687-2_68.
22. David Gitchell and Nicholas Tran. Sim: a utility for detecting similarity in computer programs. In Jane Prey and Robert E. Noonan, editors, *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, volume 6 of *SIGCSE '99*, page 266–270, NY, USA, 3 1999. ACM. ISBN 1581130856. doi:10.1145/299649.299783.
23. Robert J. Grissom and John J. Kim. *Effect Sizes for Research.* Routledge, 4 2012. ISBN 9781136632358. doi:10.4324/9780203803233.
24. Zhengyuan Jiang, Jinghuai Zhang, and Neil Zhenqiang Gong. Evading watermark based detection of ai-generated content. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1168–1181, NY, USA, 11 2023. ACM. ISBN 9798400700507. doi:10.1145/3576915.3623189.
25. Mike Joy and Micheal Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, 5 1999. ISSN 0018-9359. doi:10.1109/13.762946.
26. Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, USA, 1 2009. IEEE Computer Society. ISBN 9781424434534. doi:10.1109/ICSE.2009.5070547.
27. Oscar 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)*, pages 63–68,, 10 2016. IEEE. doi:10.1109/icts.2016.7910274.
28. Oscar Karnalim. Similarities of human and ai assistance in programming plagiarism: Student perspective. In Michael E. Auer and Tiia Rüütmann, editors, *Futureproofing Engineering Education for Global Responsibility*, pages 149–156, Cham, 2025. Springer Nature. ISBN 978-3-031-83520-9. doi:10.1007/978-3-031-83520-9_14.
29. Oscar Karnalim, Simon, and William Chivers. Preprocessing for source code similarity detection in introductory programming. In Nick Falkner and Otto Seppälä, editors, *Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, volume 36 of *Koli Calling '20*, pages 1–10, NY, USA, 11 2020. ACM. ISBN 9781450389211. doi:10.1145/3428029.3428065.
30. Oscar Karnalim, Simon, William Chivers, and Billy Susanto Panca. Educating students about programming plagiarism and collusion via formative feedback. *ACM Trans. Comput. Educ.*, 22(3):1–31, 6 2022. ISSN 1946-6226. doi:10.1145/3506717.
31. Oscar Karnalim, Hapnes Toba, and Meliana Christianti Johan. Detecting ai assisted submissions in introductory programming via code anomaly. *Education and Information Technologies*, 29(13):16841–16866, 9 2024. ISSN 1573-7608. doi:10.1007/s10639-024-12520-6.
32. Mohammad Khalil and Erkan Er. Will chatgpt get you caught? rethinking of plagiarism detection. *Interacción*, 14040:475–487, 2 2023. ISSN 0302-9743. doi:10.48550/arXiv.2302.04335. 10.48550/arXiv.2302.04335.
33. Cynthia Kustanto and Inggriani Liem. Automatic source code plagiarism detection. In Haeng-Kon Kim and Roger Y. Lee, editors, *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pages 481–486,, 5 2009. IEEE. doi:10.1109/SNPD.2009.62.
34. Thomas Lancaster. Artificial intelligence, text generation tools and chatgpt – does digital watermarking offer a solution? *International Journal for Educational Integrity*, 19(1), 7 2023. ISSN 1833-2595. doi:10.1007/s40979-023-00131-6.

35. Tri Le, Angela Carbone, Judy Sheard, Margot Schuhmacher, Michael de Raath, and Chris Johnson. Educating computer programming students about plagiarism through use of a code similarity detection tool. In *2013 Learning and Teaching in Computing and Engineering*, pages 98–105. IEEE, 3 2013. doi:10.1109/LaTiCE.2013.37.
36. Richard Lippmann, Kyle Ingols, Chris Scott, Keith Piwowarski, Kendra Kratkiewicz, Mike Artz, and Robert Cunningham. Validating and restoring defense in depth using attack graphs. In *MILCOM 2006 - 2006 IEEE Military Communications conference*, pages 1–10. IEEE, 10 2006. doi:10.1109/MILCOM.2006.302434.
37. Vedran Ljubovic and Enil Pajic. Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories. *IEEE Access*, 8:96505–96514, 2020. ISSN 2169-3536. doi:10.1109/ACCESS.2020.2996146.
38. Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering*, 43 (12):1157–1177, 12 2017. ISSN 0098-5589. doi:10.1109/TSE.2017.2655046.
39. Rien Maertens, Charlotte Van Petegem, Niko Strijbol, Toon Baeyens, Arne Carla Jacobs, Peter Dawyndt, and Bart Mesuere. Dolos: Language-agnostic plagiarism detection in source code. *Journal of Computer Assisted Learning*, 38(4):1046–1061, 8 2022. ISSN 0266-4909. doi:https://doi.org/10.1111/jcal.12662.
40. Robin Manuel Maisch. *Preventing Refactoring Attacks on Software Plagiarism Detection through Graph-Based Structural Normalization.* master's thesis, Karlsruhe Institute of Technology (KIT), 2024.
41. Leonardo Mariani and Daniela Micucci. Audentes: Automatic detection of tentative plagiarism according to a reference solution. *ACM Trans. Comput. Educ.*, 12(1):1–26, 3 2012. ISSN 1946-6226. doi:10.1145/2133797.2133799.
42. William Murray. Cheating in computer science. *Ubiquity*, 2010(October):2, 06 2010. ISSN 1530-2180. doi:10.1145/1865907.1865908.
43. Lawton Nichols, Kyle Dewey, Mehmet Emre, Sitao Chen, and Ben Hardekopf. Syntax-based improvements to plagiarism detectors and their evaluations. In Bruce Scharlau, Roger McDermott, Arnold Pears, and Mihaela Sabin, editors, *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 555–561, NY, USA, 7 2019. ACM. ISBN 9781450368957. doi:10.1145/3304221.3319789.
44. Matija Novak. *Effect of source-code preprocessing techniques on plagiarism detection accuracy in student programming assignments.* PhD thesis, University of Zagreb. Faculty of Organization and Informatics, 2 2020.
45. Matija Novak, Mike Joy, and Dragutin Kermek. Source-code similarity detection and detection tools used in academia: A systematic review. *ACM Transactions on Computing Education*, 19(3):1–37, 9 2019. ISSN 1946-6226. doi:10.1145/3313290.
46. Michael Sheinman Orenstrakh, Oscar Karnalim, Carlos Aníbal Suárez, and Michael Liut. Detecting llm-generated text in computing education: Comparative study for chatgpt cases. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 121–126. IEEE, 2024. doi:10.1109/COMPSAC61105.2024.00027.
47. K. J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bulletin*, 8(4):30–41, 12 1976. ISSN 0097-8418. doi:10.1145/382222.382462.
48. José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. Progpedia: Collection of source-code submitted to introductory programming assignments. *Data in Brief*, 46: 108887, 2 2023. ISSN 2352-3409. doi:https://doi.org/10.1016/j.dib.2023.108887.
49. Wei Hung Pan, Ming Jie Chok, Jonathan Leong Shan Wong, Yung Xin Shin, Yeong Shian Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Kuan Lim. Assessing ai detectors in identifying ai-generated code: Implications for education. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, volume 18 of *ICSE-SEET '24*, page 1–11, NY, USA, 4 2024. ACM. ISBN 979-8-4007-0498-7. doi:10.1145/3639474.3640068.

50. Chris Park. In other (people's) words: Plagiarism by university students–literature and lessons. *Assessment & Evaluation in Higher Education*, 28(5):471–488, 10 2003. ISSN 0260-2938. doi:10.1080/02602930301677.
51. Dieter Pawelczak. Benefits and drawbacks of source code plagiarism detection in engineering education. In *2018 IEEE Global Engineering Education Conference (EDUCON)*, pages 1048–1056. IEEE, 4 2018. doi:10.1109/EDUCON.2018.8363346.
52. Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon: Program Analysis and Transformation in Java. Research Report RR-5901, Inria, 2006.
53. Lutz Prechelt, Guido Malpohl, and Michael Philippsen. *JPlag: Finding plagiarisms among a set of programs.* Karlsruhe Institute of Technology, 2000. doi:10.5445/ir/542000. Technical Report.
54. Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8(11):1016, 11 2002. doi:10.3217/jucs-008-11-1016.
55. 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*, 327(1241):475–484, 7 1990. ISSN 00804622. doi:10.4324/9781315092898-2.
56. J. Romano, J.D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys? In *annual meeting of the Florida Association of Institutional Research*, pages 1–3, 2006.
57. Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 5 2009. ISSN 0167-6423. doi:10.1016/j.scico.2009.02.007.
58. Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 12 2008. ISSN 1382-3256. doi:10.1007/s10664-008-9102-8.
59. Timur Sağlam. Replication Package for "Evaluating Software Plagiarism Detection in the Age of AI", 5 2025. URL `https://github.com/tsaglam/EMSE25-SM`. Final replication package will be published on Zenodo.
60. Timur Sağlam, Moritz Brödel, Larissa Schmid, and Sebastian Hahner. Detecting automatic software plagiarism via token sequence normalization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 113:1–113:13, NY, USA, 4 2024. ACM. ISBN 9798400702174. doi:10.1145/3597503.3639192.
61. Timur Sağlam, Sebastian Hahner, Larissa Schmid, and Erik 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*, ICSE-SEET '24, pages 297–308, NY, USA, 4 2024. ACM. ISBN 9798400704987. doi:10.1145/3639474.3640084.
62. Timur Sağlam, Sebastian Hahner, Larissa Schmid, and Erik Burger. Obfuscation-resilient software plagiarism detection with jplag. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, volume 8 of *ICSE-Companion*, page 264–265. IEEE, 4 2024. ISBN 9798400705021. doi:10.1145/3639478.3643074.
63. Timur Sağlam, Nils Niehues, Sebastian Hahner, and Larissa Schmid. Mitigating obfuscation attacks on software plagiarism detectors via subsequence merging. In *46th IEEE/ACM International Conference on Software Engineering: Companion Proceedings*, CSEE&T 2025, 2025. doi:10.5445/IR/1000179016.
64. Timur Sağlam. Detecting modelling plagiarism: Navigating automated obfuscation and generative ai. In *Educators Symposium, MODELS 2024*, MODELS, 9 2024. Invited Keynote.
65. Simon, Judy Sheard, Michael Morgan, Andrew Petersen, Amber Settle, Jane Sinclair, Gerry Cross, and Charles Riedesel. Negotiating the maze of academic integrity in computing education. In *Proceedings of the 2016 ITiCSE Working Group Reports*, ITiCSE '16, page 57–80, NY, USA, 7 2016. ACM. ISBN 9781450348829. doi:10.1145/3024906.3024910.

66. M.R. Stytz. Considering defense in depth for software applications. *IEEE Security & Privacy*, 2(1):72–75, 1 2004. ISSN 1540-7993. doi:10.1109/MSECP.2004.1264860.
67. Hyunjae Suh, Mahan Tafreshipour, Jiawei Li, Adithya Bhattiprolu, and Iftekhar Ahmed. An empirical study on automatically detecting ai-generated source code: How far are we? In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, NY, USA, 4 2025. ACM. doi:10.48550/arXiv.2411.04299. To appear.
68. Anna Sutton, David Taylor, and Carol Johnston. A model for exploring student understandings of plagiarism. *Journal of Further and Higher Education*, 38(1):129–146, 1 2014. ISSN 0309-877X. doi:10.1080/0309877X.2012.706807.
69. Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. Ccaligner: A token based large-gap clone detector. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1066–1077. ACM, 5 2018. doi:10.1145/3180155.3180179.
70. Debora Weber-Wulff. Plagiarism detectors are a crutch, and a problem. *Nature*, 567 (7749):435–435, 3 2019. ISSN 1476-4687. doi:10.1038/d41586-019-00893-5.
71. Debora Weber-Wulff, Alla Anohina-Naumeca, Sonja Bjelobaba, Tomáš Foltýnek, Jean Guerrero-Dib, Olumide Popoola, Petr Šigut, and Lorna Waddington. Testing of detection tools for ai-generated text. *International Journal for Educational Integrity*, 19(1): 1–39, 12 2023. ISSN 1833-2595. doi:10.1007/s40979-023-00146-z.
72. Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer, Berlin, Heidelberg, 6 2012. ISBN 978-3-642-29043-5. doi:10.1007/978-3-642-29044-2.
73. Yuankai Xue, Hanlin Chen, Gina R. Bai, Robert Tairas, and Yu Huang. Does chatgpt help with introductory programming? an experiment of students using chatgpt in cs1. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '24, page 331–341, NY, USA, 2024. ACM. ISBN 9798400704987. doi:10.1145/3639474.3640076.
74. Lisa Yan, Nick McKeown, Mehran Sahami, and Chris 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*, SIGCSE '18, page 110–115, NY, USA, 2 2018. ACM. ISBN 9781450351034. doi:10.1145/3159450.3159490.
75. Fangfang Zhang. *Toward Obfuscation-resilient Plagiarism Detection*. PhD dissertation, The Pennsylvania State University, University Park, PA, 03 2014. Graduate Program: Computer Science and Engineering.
76. Xuandong Zhao, Prabhanjan Vijendra Ananth, Lei Li, and Yu-Xiang Wang. Provable robust watermarking for AI-generated text. In *The Twelfth International Conference on Learning Representations*. OpenReview.net, 2024. doi:10.48550/arXiv.2306.17439.