# Same Same But Different: Preventing Refactoring Attacks on Software Plagiarism Detection

Robin Maisch
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
robin.maisch@kit.edu

Larissa Schmid*
KTH Royal Institute of Technology
Stockholm, Sweden
lgschmid@kth.se

Timur Sağlam
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
timur.saglam@kit.edu

Nils Niehues
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
nils.niehues@kit.edu

## Abstract

Plagiarism detection in programming education faces growing challenges due to increasingly sophisticated obfuscation techniques, particularly automated refactoring-based attacks. While code plagiarism detection systems used in education practice are resilient against basic obfuscation, they struggle against structural modifications that preserve program behavior, especially caused by refactoring-based obfuscation.

This paper presents a novel and extensible framework that enhances state-of-the-art detectors by leveraging code property graphs and graph transformations to counteract refactoring-based obfuscation. Our comprehensive evaluation of real-world student submissions, obfuscated using both algorithmic and AI-based obfuscation attacks, demonstrates a significant improvement in detecting plagiarized code.

## CCS Concepts

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

## Keywords

Software Plagiarism Detection, Plagiarism Obfuscation, Obfuscation Attacks, Code Property Graph, Refactoring, Tokenization

**ACM Reference Format:**
Robin Maisch, Larissa Schmid, Timur Sağlam, and Nils Niehues. 2026. Same Same But Different: Preventing Refactoring Attacks on Software Plagiarism Detection. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773225

*Work was conducted while affiliated with KASTEL – Institute of Information Security and Dependability, Karlsruhe Institute of Technology (KIT).


*ICSE '26, Rio de Janeiro, Brazil*

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773225

## 1 Introduction

Plagiarism is a prevalent challenge in computer science education, facilitated by the ease of duplicating and modifying digital assignments [14, 32, 45]. Addressing this challenge is crucial to maintain academic integrity. Due to the high number of students in computer science courses, manual inspection is impractical [10, 30]. Moreover, students are creative in *obfuscating* their plagiarism to hide the relation to the original source [52]. In the case of programming assignments, students commonly utilize techniques such as renaming, reordering, or restructuring [27, 48, 49].

In light of these issues, educators commonly use software plagiarism detection systems [18], which automate parts of the detection process and enable plagiarism detection at scale. These detectors analyze sets of programs to identify pairs with a suspiciously high degree of similarity [54]. Most approaches compare the code structure [46, 49], with token-based approaches like MOSS [1] and JPlag [54] most widely used in practice [4, 31, 49].

Token-based approaches parse and linearize programs, capturing their structure in an internal representation. On pairs of these linearized representations, matching code fragments of significant length are identified. The match coverage is then used to compute a similarity score and derive suspicious candidates [60]. Nevertheless, assessing which candidates qualify as plagiarism is ultimately a human decision, given the underlying ethical considerations [15, 69]. By only including essential program elements into the internal representation for comparison, these approaches intentionally abstract from details such as names, types, and literals [54]. This abstraction makes these detectors inherently resilient against simple obfuscation attempts based on renaming, retyping, and other lexical changes [23]. However, for more advanced obfuscation attacks, their resilience remains insufficient [18, 38, 60].

Despite these shortcomings, for novice programmers, evading detection is not feasible, as manually obfuscating a program is tedious and requires a profound understanding of programming languages [23]. The essential role of software plagiarism detectors in guiding educators' inspections of suspicious candidates [9] relied on the premise that defeating them takes more effort than completing the actual assignment [18]. However, this premise has been broken with the recent rise of automated *obfuscation attacks* [8, 18, 20, 52]. These *obfuscation attacks* aim to avoid detection by automatically altering the structural properties of a program without changing its underlying behavior [61].

Several recent works in the field of software plagiarism detection present countermeasures against specific types of automated obfuscation attacks, such as independent statement reordering or dead statement insertion [61–63]. However, existing approaches are not resilient against obfuscation attacks based on refactoring operations [42]. *Refactoring-based obfuscation attacks* modify programs via refactoring operations to change their structure while preserving their behavior to ensure plagiarism instances qualify as valid solutions. These attacks modify the program structure by applying refactoring operations at suitable locations. These attacks include refactoring operations such as function extraction or inlining, introducing unnecessary control flow constructs, and restructuring loops and conditionals. These attacks significantly alter the syntactic representation of programs, making it challenging to identify plagiarized code. While early automated attacks relied solely on algorithmic approaches [18], generative artificial intelligence can also be used to automate refactoring attacks [16, 21], making the obfuscation more accessible than ever before [28, 62]. Thus, both algorithmic and AI-based obfuscation attacks pose a viable threat to the detectors used in educational practice. Even attack-independent approaches [64] do not provide sufficient resilience as they cannot systematically address complex program structure changes.

*Approach.* This paper presents Nocte, a framework to provide token-based plagiarism detection systems with resilience against refactoring-based obfuscation attacks. Nocte transforms programs into a normalized structure concerning semantically equivalent alternatives. While token-based code plagiarism detectors usually tokenize the parsed program code directly, Nocte transforms all input programs into Code Property Graphs (CPGs) [72]. CPGs, typically used for software vulnerability analysis, combines a program's abstract syntax tree, evaluation order graph, and program dependence graph. On these graphs, we apply a set of graph transformations specifically designed to counter obfuscation attacks. Next, we generate linear program representations from the normalized CPGs and pass them to the plagiarism detector, allowing it to proceed as usual. To ensure a deterministic order, we employ topological sorting [26] during tokenization. Nocte is modular and extensible, as we separate the CPG-based framework from the set of deobfuscation transformations. Thus, additional graph transformations can be integrated seamlessly to address emerging threats.

*Evaluation.* We evaluate Nocte based on the code plagiarism detector JPlag [24, 54]. To that end, we employ four real-world datasets of student submissions and both algorithmic and AI-based obfuscation attacks. We evaluate 1037 original and 540 obfuscated programs, encompassing more than 3.5 million pairwise comparisons. The results show that Nocte significantly outperforms the state of the art for insertion-based and refactoring-based obfuscation attacks. While AI-based obfuscation is less reliable than algorithmic attacks, we show that it remains an insufficiently addressed challenge for *all* current approaches. Moreover, the results show that Nocte enhances the detection of AI-generated programs despite not being designed for this task. We provide our code and the evaluation data via the supplementary material [40].

**Table 1: Original code and modified variant after inserting two statements (+), removing one (−), and altering two (~).**

| # | Original | → | Variant |
|---|---|---|---|
| 1 | `printRoots(int n) {` | | `printRoots(int n) {` |
| 2 | | (+) | `int i = 0;` |
| 3 | `for (int i=0; i<n; i++) {` | (~) | `while (i < n) {` |
| 4 | `double d = sqrt(i);` | | `double d = sqrt(i);` |
| 5 | `d++;` | (−) | |
| 6 | `println(d);` | (~) | `println(++d);` |
| 7 | | (+) | `i++;` |
| 8 | `}` | | `}` |
| 9 | `}` | | `}` |

**Table 2: The two token sequences corresponding to the programs in Table 1 with matching subsequences highlighted.**

| id | Original Tokens | → | id | Variant Tokens |
|---|---|---|---|---|
| 1 | method start | | 1 | method start |
| 2 | variable | | 2 | variable |
| | | (+) | 2 | variable |
| 3 | loop start | | 3 | loop start |
| 2 | variable | | 2 | variable |
| 4 | assign | (~) | 5 | apply |
| 2 | variable | (−) | | |
| 5 | apply | | 5 | apply |
| 4 | assign | | 4 | assign |
| 5 | apply | (~) | 4 | assign |
| 6 | loop end | | 6 | loop end |
| 7 | method end | | 7 | method end |

*Contributions.* In this paper, we present three contributions:

- **C1** Nocte, a *CPG transformation framework*, including a graph linearization and tokenization component, which allows to counter refactoring-based obfuscation attacks.
- **C2** A *set of fourteen refactoring transformations* covering a wide variety of refactoring-based obfuscation attacks, thus providing resilience against such attacks.
- **C3** A comprehensive evaluation using real-world data sets with automatically obfuscated plagiarism instances based on both algorithmic and AI-based obfuscation.

## 2 Refactoring-based Obfuscation Attacks

State-of-the-art detection approaches compare program structures by identifying similarities between code fragments [46]. Thus, obfuscation attacks aim to prevent the detector from matching related fragments by changing the program structure beyond trivial and lexical changes, such as renaming program elements.

Table 1 shows the effects of two specific refactoring operations on a program: First, the index-based `for` loop on line 3 is replaced with a `while` loop, requiring an increment statement on line 7. Second, the increment operation in line 5 is inlined into the print statement, changing it from a post-increment to a pre-increment operator. Both resulting programs–the original and the variant–still print the square roots of a continuous sequence of integers. Yet, the two refactoring steps alter the program structure enough to evade detection through common software plagiarism detection systems.

Before we illustrate the effect of structural changes on the detection process, we look more closely at the internal program representation used by such detection systems. Plagiarism detectors used in education practice linearize programs by parsing them, traversing the parse tree, and capturing the type of relevant program elements as tokens [61]. The resulting token sequence is an abstract representation of the program structure, discarding details such as names, data types, and values. For the comparison step, the token sequence is encoded using distinct natural numbers for each structural element type, yielding a finite sequence of natural numbers [60]. Table 2 depicts the token sequences of the two programs in Table 1, with *id* indicating the numerical representation. Plagiarism detectors compute the similarity between program pairs by identifying matching subsequences within their corresponding token sequences. To ensure that matches are meaningful, these detectors discard matches below a set *minimum match length*. While this threshold is crucial to avoid false positives, it also introduces a vulnerability: Obfuscation attacks may systematically alter the original program code to fragment long, continuous token matches into smaller parts, eventually pushing them below the set matching threshold and thus causing them to be discarded [18]. This illustrates that effective obfuscation attacks operate on code to affect the token sequence.

Returning to the example, the modifications that derived the variant from the original program also altered its token representation. Four matching subsequences are apparent between the token sequences, highlighted in gray, each confined by unmatched tokens. If we assume a minimum match length of 3, all matches fall below it. Thus, the plagiarism detector would discard these matches, resulting in a similarity of 0 % for these two programs. In real scenarios, far higher minimum match lengths are common.

These concepts, illustrated in Table 1 at a rather small scale, also apply to larger programs. Moreover, the refactoring operations used here are two typical examples among many possible refactoring operations that can be effectively exploited as obfuscation attacks. Other possible operations include extracting and inlining methods, replacing `if` cascades with `switch` statements, and creating dummy variables, methods, or classes [27].

## 3 Approach

As established in the previous section (cf. Section 2), refactorings are an effective method to obfuscate plagiarism: They alter the code structure, thereby breaking up matches and reducing the similarity of the resulting token sequence when compared to the original. To mitigate the effects of refactoring-based obfuscation attacks on token-based plagiarism detection, we introduce the NOrmalization-driven Code Transformation Engine (Nocte), a code transformation framework that provides structural normalization for code submissions as a preprocessing step for token-based plagiarism detection (**C1**). The transformations that Nocte employs for the normalization are modular and easily adaptable. We provide an initial set of transformations (**C2**) which address refactorings frequently referenced in the literature [27, 47].

The core idea of our approach is as follows: Instead of directly linearizing the program into a token sequence, as is common for token-based plagiarism detectors, we first construct a graph-based representation for each program. On these graphs, we may apply graph transformations to normalize their structures and thus

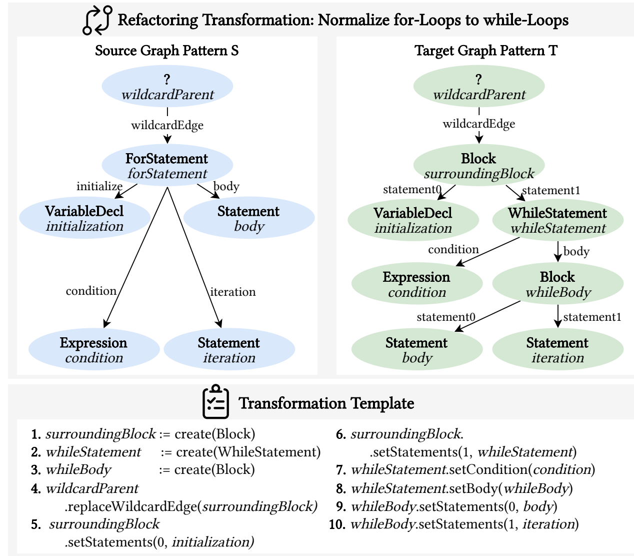


**Figure 1: Example of a refactoring transformation with a corresponding transformation template containing operations on how to refactor the source graph pattern $S$ into the target graph pattern $T$. Each node pattern shows its type (bold) and its role (italicized).**

**Table 3: Normalized version of both the original and variant code of Table 1, illustrated here as code instead of a CPG.**

| # | Normalized Version |
|---|---|
| 1 | `printRoots(int n) {` |
| 2 | `  int i = 0;` |
| 3 | `  while (i < n) {` |
| 4 | `    println(sqrt(i++) + 1);` |
| 5 | `  }` |
| 6 | `}` |

reverse the effects of obfuscation attacks. For this purpose, we employ Code Property Graphs (CPGs) due to their expressiveness and detailed program representation. After executing the graph transformations, we linearize the graph representations and pass them to the plagiarism detector, allowing the detection process to proceed as usual.

Returning to the example submissions illustrated in Table 1, for example, we could realign both versions by replacing *all* `for` loops with `while` loops and inlining single-use assignments. As the resulting *normalized* code structures of both versions (Table 3) are identical, their corresponding token sequences are identical as well. Thus, the plagiarism detectors will easily identify this pair as highly similar, raising the attention of instructors, and successfully overcoming the initial obfuscation attempt.

In detail, Nocte incorporates four steps, as illustrated in Figure 2: As a one-time initialization, each transformation is analyzed to derive its individual transformation operations from its textual representation (Section 3.1). For each submission, a CPG is constructed first, capturing detailed syntactic and semantic information about the code structure (Section 3.2). Then, each transformation is applied to the CPG until it reaches a fixed, normalized state (Section 3.3). Finally, the normalized CPG is traversed to extract a token

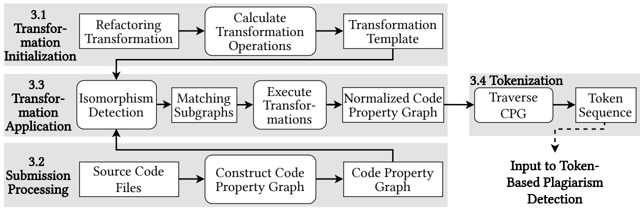


**Figure 2: Pipeline of our Normalization Procedure.**

sequence for the submission (Section 3.4). This token sequence is the output of the transformation engine and the basis for pairwise comparison of submissions by the plagiarism detection tool.

Once integrated into the user's token-based plagiarism detection system of choice, the entire normalization process is a fully automated step in the existing pipeline, requiring no further user interaction.

The following sections describe each step of the pipeline in detail.

## 3.1 Transformation Initialization

We represent graph transformations as pairs $(S, T)$ of a source and target graph pattern, adapted from [44], shown in Figure 1 (top). First, we determine the individual *transformation operations* needed to perform each transformation, i.e., to adapt the structure of $S$ to $T$. To this end, we compare $S$ and $T$ using in-parallel depth-first search, selecting the appropriate operation to address each difference between the two patterns. The resulting sequence of operations *ops*, shown in Figure 1 (bottom), and the source graph $S$ constitute the *transformation template* for a transformation.

## 3.2 Submission Processing

For each code submission in the corpus, we construct a CPG, the intermediate representation on which we will perform the normalization step. This CPG generation entails parsing the submission into an Abstract Syntax Tree (AST) and then iteratively extending the tree with additional edges and node attributes, capturing in-depth structural and semantic connections across the graph. The resulting CPG combines the AST with an Evaluation Order Graph (EOG), a Data Dependency Graph (DDG), and a Control Dependency Graph (CDG). We base our CPG construction pipeline on the CPG framework presented by Fraunhofer AISEC [70].

## 3.3 Transformation Application

The CPG $\mathcal{G}$ is normalized using the given set of transformations. To this end, each transformation $\mathcal{T}$ is iteratively applied at every substructure of $\mathcal{G}$ possible. After that, the process continues with the subsequent transformation. The individual steps of the transformation application are described in more detail in the following paragraphs.

*3.3.1* ***Isomorphism detection.*** For each transformation template $(S, ops)$, a graph isomorphism detection algorithm iteratively identifies subgraphs of the submission CPG $\mathcal{G} = (N, V)$ on which the transformation can be applied, i.e., subgraphs which match the structure of the source graph pattern $S$. Many transformations specify additional conditions that a matching subgraph must satisfy to ensure the transformation will not change the program behavior. Each *match* of $S$ in $\mathcal{G}$ stores the mapping of concrete CPG nodes of the matching subgraph to the *role* that they assume in the course of the transformation.

*3.3.2* ***Transformation instantiation and execution.*** For each match of $S$, the transformation operations *ops* are applied to the concrete CPG nodes corresponding to the different roles of the transformation. After all matches are transformed, Nocte searches for matches with the same transformation template again, as new matching subgraphs may have emerged as a result of the previous transformations, where the transformation can be applied.

*3.3.3* ***Normalization of statement order.*** State-of-the-art token-based plagiarism detection is already resilient against reordering of large code elements such as methods; however, at the statement level, obfuscations via permutation and insertion remain effective. To defend against these obfuscation attacks, we use the control flow information of the EOG subgraph to determine data dependencies between method statements. Then, we deterministically sort the statements, taking into account their mutual dependencies. Also, statements with no transitive data dependencies to the program state (*dead code*) are removed from the CPG, including not only the declaration of unused variables, but also unused assignments.

This normalization step cannot be expressed as a set of CPG transformations, as it does not target a specific structure in the code. Rather, it is applied as an intermediate step between two phases of transformation.

## 3.4 Tokenization

*3.4.1 Linearization.* To derive a linear token sequence out of an inherently non-linear code graph, we have to specify the order in which we want to traverse various elements on various levels, e.g., the members of a class, the subexpressions of a statement. While the state-of-the-art approach uses preorder depth-first search to replicate the AST structure, we opt for postorder depth-first search, which resembles the EOG structure—the order in which subexpressions may be evaluated at runtime. In particular cases, this eliminates the opportunity to use simple syntactical variants for plagiarism obfuscation, e.g., defining a `for` loop variable inside the `for` statement vs. before the `for` statement.

*3.4.2 Token Selection.* To allow seamless integration of Nocte into an existing token-based plagiarism detection system, we may directly reuse the existing tokenization rules, thereby replicating the abstraction level and the specific language feature selection that will be represented in the token sequence.

## 3.5 Limitations

This section discusses the limitations and potential areas for extension of the presented normalization.

**Language dependence** While our CPG approach supports multiple programming languages, the refactoring transformations are designed for Java code. Some of them are applicable in related languages like C#, Python, or C++; however, testing for wide language support was outside of the scope of our evaluation. We expect that language-specific refactoring transformations must be tailored to each programming language to ensure effectiveness.

**Selection of transformations** NOCTE is designed to counter precisely the set of refactoring-based obfuscation attacks covered by the supplied refactoring transformations, providing tailored, modular defense against specific attacks. This means that attacks not part of the refactoring transformations may still be effective. We propose refactoring transformations that cover common refactoring attacks (cf. Section 4) and provide easy ways to extend the list of refactoring transformations.

**Conservative semantic-preserving approach** To avoid an increased false-positive rate, we ensure that each step of the normalization pipeline does not alter the semantics of the code. Adversaries, on the other hand, may employ semantic-changing obfuscation attacks, which our approach does not address. However, these types of attacks are uncommon [60], as semantically deviant programs typically do not pass as correct solutions. Even if semantic-changing obfuscation is employed, we expect that NOCTE will still help distinguish plagiarism from unrelated programs.

# 4 Definition of Refactoring Transformations

In this section, we present our second main contribution (**C2**), the selection of transformations: We selected 13 refactoring strategies from the literature [27, 48] and designed 14 corresponding transformations to counter them. Our goal was a diverse set of attacks with a wide range of complexity and scope, broadly applicable to programs and thus easy to automate. Large-scale refactoring operations, such as the introduction of design patterns, are less broadly applicable as they must be tailored to the program at hand and require expertise to be applied correctly; thus, we consider large-scale refactoring out of scope. Instead, we focus on broadly applicable operations, ranging in granularity from statement-level to class-level transformations. We group the attack schemes in four categories: (i) *Inserting Elements*, (ii) *Moving Elements*, (iii) *Extracting Elements*, and (iv) *Semantically Equivalent Replacement*.

In the next sections, we describe each category of refactoring attack and the corresponding normalizing transformations in detail.

## 4.1 Inserting Elements

The insertion of elements may split matching token subsequences, or dilute the overall similarity by adding large amounts of code that do not contribute to the program behavior. According to Novak [48], many of these elements are considered *common code* and may be removed in the context of plagiarism detection. Also, empty classes may result from other transformations. We support the removal of these inserted elements with the following transformations:

*(1) Remove Empty Methods:* void-type methods with an empty block are removed.
*(2) Remove Empty Constructors:* constructors with an empty block are removed.
*(3) Remove Empty Classes:* classes with no methods, fields, or inner classes are removed.
*(4) Remove Getter Methods:* non-void methods that directly return either a field of the surrounding class or a constant are removed.
*(5) Remove Unsupported Methods:* methods that immediately throw an unconditional exception are removed.
*(6) Remove Unsupported Constructors:* constructors that immediately throw an unconditional exception are removed.

Since unsupported methods and constructors are not part of the intended program behavior, we treat them as inserted elements and remove them.

## 4.2 Moving Elements

Another attack scheme is to move code fragments away from their immediate context, according to a usage analysis. A strategy we often observe in real-life submissions is the creation of dedicated utility classes, where class constants and auxiliary methods are moved to obfuscate plagiarism. We support the analysis and context restoration for the following elements:

*(7) Move Constants To Only Using Class:* Constants which are used only in one particular class, different from the current declaring class of the constant, are moved into that class.

## 4.3 Extracting elements

Refactoring-based obfuscation attacks in this category may create additional declarations and/or function calls by extracting or wrapping code elements. We support inlining the following elements:

*(8) Inline Single-use Variables:* Local variables are inlined if (i) they are referenced exactly once after their declaration, (ii) that reference is a read access, and (iii) the expression assigned to it in its declaration does not change value until that reference.
*(9) Inline Single-use Constants:* Class constants are inlined if they are referenced exactly once. In contrast to single-use variables, we expect constants to be initialized with a constant value.
*(10/11) Inline Optional Values:* Values wrapped in an Optional object, which handles the case that the value is `null`. This comprises a transformation that replaces the creation of the wrapped value with the value itself, and another that replaces the unwrapping method call with the proper value.

To ensure the preservation of the semantics through the transformation *Inline Single-use Variable*, it is only applied if the value of the assigned expression remains unchanged up to the potential inlining position. We use the CPG's data flow information to verify this precondition.

## 4.4 Semantically Equivalent Replacement

By replacing specific structures with semantically equivalent alternatives in the source code, the token sequence of a plagiarism instance can be altered to obfuscate its relation to the original. We consider the following replacements:

*(12) Revert Negated If-Else:* If the condition expression of an `if-else` statement is a negation expression, the `then` and `else` blocks are swapped, and the negated inner expression replaces the condition.
*(13) Revert If-Unequal-Else:* If the condition expression of an `if-else` statement is an inequality expression, the `then` and `else` blocks are swapped, and an equality expression replaces the condition.
*(14) For Loop To While Loop:* All `for` loops are replaced by the equivalent `while` statement, moving the declaration of the loop variable before the `while` statement, and the iteration statement to the end of the `while` block.

To ensure the preservation of the semantics through the transformation *For Loop To While Loop*, the scope of the newly created loop variable must end after the `while` statement. To that end, the `while` statement and the definition of its loop variable must be moved into an inner block. There are countless variants of possible refactorings involving if-else blocks, e.g., the inclusion of the optional `else` block if the `if` block always ends in a `return statement`, and also notably, the usage of a condition which is constant at runtime, but complex enough that conservative static analysis is unable to determine that it is constant.

## 5 Evaluation

In this section, we present the evaluation of Nocte (**C3**).

We employ four real-world datasets and generate plagiarism instances with four types of real-world automated obfuscation attacks. In total, we used 1037 original and 540 obfuscated programs, resulting in over 3.5 million pairwise comparisons. The results demonstrate that Nocte not only matches the resilience of other approaches but also significantly outperforms them for refactoring-based and insertion-based obfuscation attacks. We provide our code and the evaluation data via the supplementary material [40].

### 5.1 Methodology

With our evaluation, we set out to answer the following questions:

**Q1** Does Nocte provide resilience to insertion-based attacks?
**Q2** Does Nocte provide resilience to refactoring-based attacks?
**Q3** Does Nocte provide resilience to AI-based attacks?
**Q4** Does Nocte improve the detection of AI-generated programs?

We choose the plagiarism detector JPlag [24, 54] as a baseline as it is widely used, frequently referenced in literature, and open-source [4, 49]. We also compare Nocte with two related approaches from our earlier work:

- Token Sequence Normalization (TSN) [61], a pre-comparison optimization approach on the token sequences; and
- Subsequence Match Merging (SMM) [64], a post-comparison optimization approach on the matches.

To our knowledge, these are the only state-of-the-art, tool-independent approaches that target *automated* obfuscation in the context of plagiarism detection. Note that while clone detection approaches are related, they do not consider attacker-defender scenarios and are thus tampering-prone (see section 6). TSN enables normalization based on program dependence graphs, thus providing resilience against insertion-based and reordering-based obfuscation. SMM is an algorithm that heuristically merges interrupted code fragments for plagiarism detectors as post-processing. Thus, it provides resilience against a variety of obfuscation attacks. Nocte, TSN, and SMM are all implemented based on JPlag and can thus be directly compared to the baseline.

For all evaluation questions, we employ the following metrics:

**M1** Separation between values plagiarism instances and unrelated programs (median similarity difference).
**M2** Statistical significance of the results compared to the baseline (based on one-sided Wilcoxon signed-rank tests).
**M3** Practical significance of the results compared to the baseline (based on Cliff's delta [13] and its interpretation [57]).

As educators use plagiarism detectors to identify suspicious outliers, **M1** measures how well the plagiarism instances are separated from unrelated programs. To test if the approaches improve the baseline JPlag, **M2** and **M3** evaluate for statistical and practical significance via statistical tests.

We now discuss the obfuscation attacks and datasets we used.

#### 5.1.1 *Obfuscation Attacks.*

We generate plagiarism instances from existing programs by applying three different automated obfuscation attacks: dead code insertion, refactoring, and AI-based code rewriting. We omit obfuscation by reordering independent statements, as it is not an effective obfuscation attack [61]. Moreover, we evaluate the similarity of programs entirely generated by an AI based on the assignment description, which does not constitute obfuscation in a classical sense. In the following, we will only briefly discuss these obfuscation attacks, as we do not want to encourage employing these attacks.

For insertion-based obfuscation, we utilize the attack outlined by Devore-McDonald and Berger [18] and implemented for Java code by Sağlam et al. [61]. This attack inserts dead statements from the original program and a pool of pre-defined statements. After each insertion, the resulting program is compiled to ensure syntactical correctness and preservation of program behavior.

For refactoring-based obfuscation, we leverage *Spoon* [53] and automatically apply multiple behavior-preserving refactoring operations at random positions at the AST level to obfuscate a given program, simulating an automated combined obfuscation attack pattern. While this only includes refactoring operations that Nocte can address, we do not evaluate the effect of individual refactoring types in isolation; instead, we assess the impact of repeatedly applying all refactoring types in combination, ensuring that our evaluation focuses on the overall obfuscation effect. In detail, the refactoring operations include optional wrapping, extracting expressions as new variables, introducing constant container classes and extracting constants, swapping if-else statements and inverting the corresponding conditions, inserting methods and constructors, and introducing access methods for existing fields.

To complement the two algorithmic obfuscation attacks, we employ an AI-based attack using OpenAI's GPT-4 as a third strategy. We use 16 different zero-shot prompts to obfuscate pre-existing programs, mimicking how students might use generative AI to obfuscate their plagiarism. This selection of prompts results from a preliminary study in which we evaluated nearly 50 prompts with varying degrees of specificity. Some prompts specified obfuscation techniques (e.g., *insert dead code*), while others were formulated in a deliberately general style (*modify the code to look different but behave as before*). The study revealed that more specific prompts are less effective in modifying structural properties of the programs, thus resulting in high similarities between the original and obfuscated program; consequently, we restrict our evaluation to rather unrestrictive, more effective prompts—listed in [40]—which range from requesting minor structural changes to requesting a refactored version of the original program. Note that there is no guarantee that the program behavior is preserved by GPT-4.

For the final evaluation stage, we generate complete programs from the textual assignment description.

**Table 4: Evaluation datasets, their number of programs, mean lines of code (LOC), programming language, and source.**

| Dataset Name | Programs | Mean LOC | Language | Source |
|---|---|---|---|---|
| PROGpedia19 | 27 | 131 | Java | [50] |
| PROGpedia56 | 28 | 85 | Java | [50] |
| TicTacToe | 626 | 236 | Java | [61] |
| BoardGame | 434 | 1529 | Java | [61] |

**Table 5: Overview of the number of plagiarized programs per dataset and obfuscation attack type (540 programs in total).**

| Attack Type | Pp19 | Pp56 | TTT | BG |
|---|---|---|---|---|
| Insertion-based Obf. | 27 | 28 | 50 | 20 |
| Refactoring-based Obf. | 27 | 28 | 50 | 20 |
| AI-based Obf. | 80 | 80 | 80 | - |
| AI-based Generation | - | - | 50 | - |

*5.1.2 Datasets.* For our evaluation, we use two tasks from the publicly available dataset PROGpedia [50] and the datasets *TicTacToe* and *BoardGame* from our prior work [61]. In total, we use four real-world datasets of programs created by university students, originating from different courses and assignment types (see Table 4). We filter out manual, human-made plagiarism across all datasets to evaluate automated obfuscation. Table 5 shows the number of plagiarism instances generated from these datasets.

We use two tasks from the *PROGpedia* dataset [50]. Task 19 involves the design of a graph data structure and a depth-first search to analyze a social network, while Task 56 concerns minimum spanning trees using Prim's algorithm. Both datasets contain small to medium-sized Java programs, and incorrect solutions were omitted. Additionally, we use two datasets from mandatory assignments of an introductory programming course, *TicTacToe* and *BoardGame*. These datasets contain command-line-based Java implementations of the paper-and-pencil game Tic-Tac-Toe and a comprehensive board game, respectively. *TicTacToe* includes 626 medium-sized programs, while *BoardGame* contains 434 very large programs.

## 5.2 Results

In this section, we present the evaluation results. We discuss each obfuscation attack separately. For each approach, we compare the pairwise similarity values of plagiarism pairs (ideally high) with those of unrelated program pairs (ideally low). The greater their separation, the easier it is for educators to spot plagiarism cases (cf. **M1**). To show statistical and practical significance (cf. **M2** and **M3**), we conduct statistical tests, as illustrated in Table 6 and Table 7.

*5.2.1 Insertion-based Obfuscation.* First, we examine the resilience against obfuscation via dead statement insertion (**Q1**). Figure 3 shows the results for each dataset and each approach, as well as the unmodified version of JPlag as the baseline. The plots show all median similarity values and the difference between them. For insertion-based obfuscation, Nocte and TSN provide the strongest separation between plagiarism pairs and unrelated programs, with median differences of 80–99 percentage points (pp). Both approaches achieve near-immunity against insertion-based obfuscation. While SMM also increases separation compared to the baseline, the effect is less pronounced, with median differences of 7.8–58.1 pp. Thus, some overlap between plagiarism pairs and unrelated programs remains for SMM. The statistical tests (Table 6) show that all three approaches are statistically and practically significant improvements over the baseline. Notably, both Nocte and TSN exhibit a *very large* effect size for all datasets.

**Answer to Q1:** *Nocte provides strong resilience against insertion-based obfuscation, outperforming SMM, and matching TSN.*

*5.2.2 Refactoring-based Obfuscation.* Then, we examine resilience against refactoring-based obfuscation (**Q2**). Figure 4 shows the corresponding results. For refactoring-based obfuscation, Nocte outperforms the other approaches and is the only one to achieve strong separation between plagiarism pairs and unrelated programs with median differences of 76–92 percentage points (pp), thus achieving near-immunity. TSN offers no benefit over the baseline, as it is designed only to provide resilience against insertion- and reordering-based obfuscation. While SMM provides improved separation over the baseline, the effect is less pronounced, with median differences of 23–39 pp. Again, some overlap remains for SMM. The statistical tests (Table 6) show that the improvements of Nocte and SMM are both statistically and practically significant. For Nocte, the effect size is *very large*, while for SMM, it is only *medium* to *large*.

**Answer to Q2:** *Nocte provides strong resilience against refactoring-based obfuscation, significantly outperforming both SMM and TSN.*

*5.2.3 AI-based Obfuscation.* Subsequently, we examine resilience against AI-based obfuscation (**Q3**). Here, we omit the *BoardGame* dataset because the programs exceed GPT-4's token limit. Note that GPT-4 offers no guarantee of preserving the original program behavior, even when instructed to do so in the prompt. This is unlike algorithmic obfuscation, which ensures semantic preservation by design. Figure 5 shows the corresponding results. Compared to algorithmic obfuscation, the variance of the similarity values is high. Furthermore, AI-based obfuscation is not particularly strong, as the overlap between plagiarism and original programs is limited. However, all approaches show only limited improvements. Again, TSN shows no improvement over the baseline. SMM and Nocte show limited improvement, with SMM showing the best results. However, no approach achieves clear separation. The statistical tests (Table 6) show statistical significance for Nocte and SMM (except for PROGpedia56 for Nocte). However, practical significance is *negligible* to *small* for both.

**Answer to Q3:** *Although AI-based obfuscation is less reliable than other attacks, Nocte currently has no significant impact on resilience against it.*

*5.2.4 AI-generated Programs.* Finally, we examine the similarity of AI-generated programs (**Q4**). We only use one dataset here, as the full assignment description is required. Note that AI-based code generation does not constitute plagiarism in a classical sense [49, 62]. Thus, source-code plagiarism detection approaches are not generally designed for AI-generated code detection. Figure 6 shows the results comparing the similarity of unrelated human programs with the similarity among AI-generated programs. Note that even for the baseline, AI-generated programs exhibit a higher similarity than programs by humans. Among the approaches, Nocte and SMM

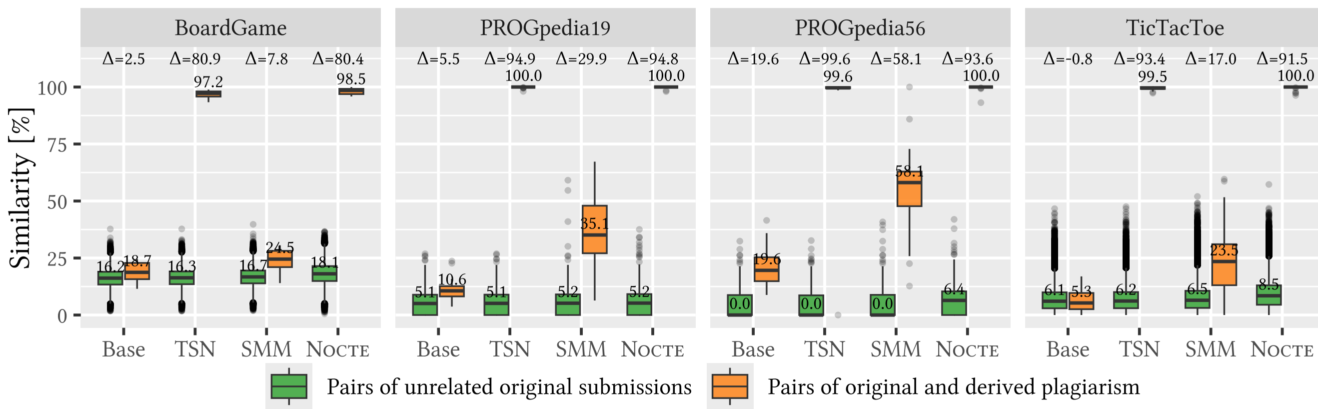


**Figure 3: Similarities for unrelated human programs and plagiarism instances based on insertion-based obfuscation.**

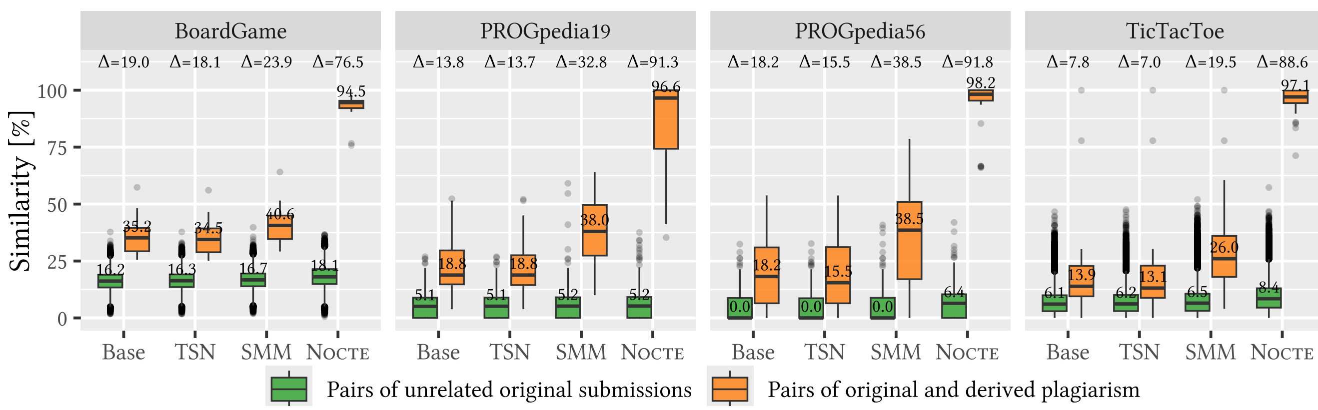


**Figure 4: Similarities for unrelated human programs and plagiarism instances based on refactoring-based obfuscation.**

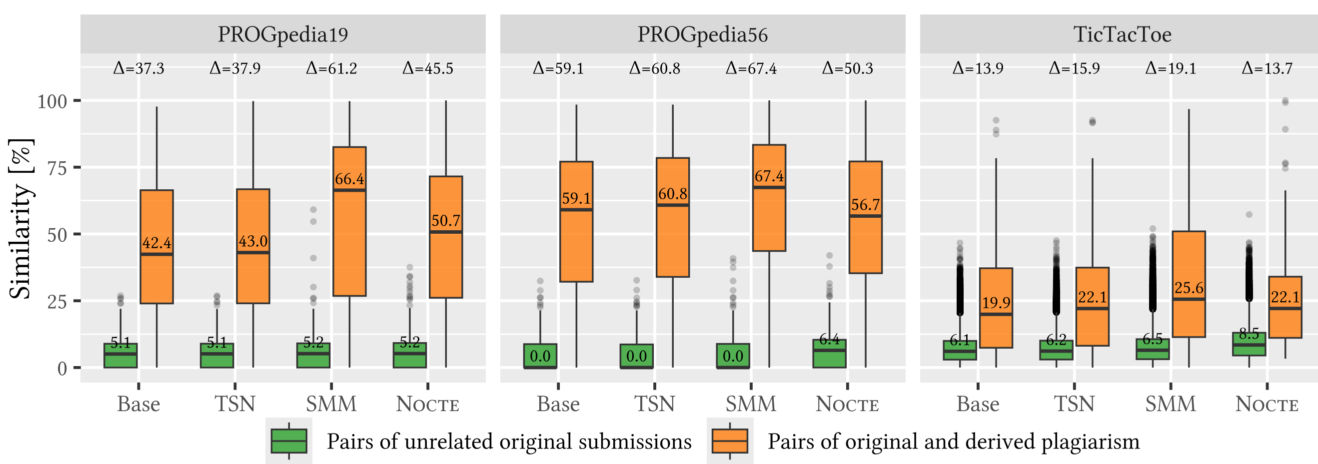


**Figure 5: Similarities for unrelated human programs and plagiarism instances based on AI-based obfuscation.**

provide the largest improvement with a separation of around 17 pp. NOCTE shows the highest median similarity for a single approach of around 25 percent. Notably, combining NOCTE with SMM achieves even better results. Here, the separation reaches around 20 pp, and the generated programs exhibit a median similarity of 29 percent. The statistical tests (Table 6) show that the improvement over the baseline is statistically significant for NOCTE, SMM, as well as the combination of both. The practical significance is limited for either approach in isolation, but their combination reaches a measurable, albeit *low*, effect size.

**Answer to Q4:** *NOCTE improves the detection of AI-generated programs, especially when combined with SMM.*

**Table 6: One-sided Wilcoxon signed-rank test comparing improvements over the baseline ($\alpha = 0.01$, $H1 = greater$), with effect size (Cliff's $\delta$), interpretation ($\delta Int.$), and 95 % confidence interval ($CI$). Low $p$ and high $\delta$ are ideal.**

| Obf. | ds | Approach | $p$ | $\delta$ | $\delta$ Int. | $\delta$ 95 % CI |
|---|---|---|---|---|---|---|
| Insertion-based | BG | TSN | 4.8e-05 | 1 | Very Large | [ 1.00, 1.00] |
| | | SMM | 4.8e-05 | 0.56 | Large | [ 0.22, 0.78] |
| | | Nocte | 1.1e-04 | 1 | Very Large | [ 0.99, 1.00] |
| | Pp19 | TSN | 3e-06 | 1 | Very Large | [ 1.00, 1.00] |
| | | SMM | 2.2e-05 | 0.82 | Very Large | [ 0.55, 0.93] |
| | | Nocte | 3e-06 | 1 | Very Large | [ 1.00, 1.00] |
| | Pp56 | TSN | 2.2e-06 | 0.93 | Very Large | [ 0.63, 0.99] |
| | | SMM | 4.4e-06 | 0.88 | Very Large | [ 0.66, 0.96] |
| | | Nocte | 2e-06 | 1 | Very Large | [ 1.00, 1.00] |
| | TTT | TSN | 3.9e-10 | 1 | Very Large | [ 1.00, 1.00] |
| | | SMM | 2.7e-09 | 0.79 | Very Large | [ 0.63, 0.89] |
| | | Nocte | 3.8e-10 | 1 | Very Large | [ 1.00, 1.00] |
| Refactoring-based | BG | TSN | 1 | -0.06 | Negligible | [-0.39, 0.29] |
| | | SMM | 4.8e-05 | 0.42 | Medium | [ 0.05, 0.69] |
| | | Nocte | 1.6e-04 | 1 | Very Large | [ 0.99, 1.00] |
| | Pp19 | TSN | 1 | -0.04 | Negligible | [-0.33, 0.26] |
| | | SMM | 1.4e-05 | 0.53 | Large | [ 0.23, 0.74] |
| | | Nocte | 3e-06 | 0.98 | Very Large | [ 0.91, 0.99] |
| | Pp56 | TSN | 0.31 | -0.01 | Negligible | [-0.31, 0.28] |
| | | SMM | 1.6e-04 | 0.35 | Medium | [ 0.04, 0.59] |
| | | Nocte | 2e-06 | 1 | Very Large | [ 1.00, 1.00] |
| | TTT | TSN | 0.98 | -0.02 | Negligible | [-0.24, 0.21] |
| | | SMM | 2.7e-09 | 0.51 | Large | [ 0.29, 0.68] |
| | | Nocte | 5.7e-10 | 0.96 | Very Large | [ 0.81, 0.99] |
| AI-based | Pp19 | TSN | 0.26 | 0.01 | Negligible | [-0.14, 0.16] |
| | | SMM | < 1e-10 | 0.19 | Small | [ 0.03, 0.34] |
| | | Nocte | 1.8e-10 | 0.09 | Negligible | [-0.06, 0.24] |
| | Pp56 | TSN | 0.016 | 0.05 | Negligible | [-0.10, 0.20] |
| | | SMM | < 1e-10 | 0.2 | Small | [ 0.05, 0.35] |
| | | Nocte | 0.067 | 0.04 | Negligible | [-0.12, 0.19] |
| | TTT | TSN | 0.012 | 0.04 | Negligible | [-0.12, 0.19] |
| | | SMM | < 1e-10 | 0.11 | Negligible | [-0.04, 0.26] |
| | | Nocte | 3.1e-05 | 0.06 | Negligible | [-0.09, 0.21] |
| AI Gen. | TTT | TSN | 1 | -0.07 | Negligible | [-0.12, -0.03] |
| | | SMM | < 1e-10 | 0.1 | Negligible | [ 0.05, 0.15] |
| | | Nocte | < 1e-10 | 0.13 | Negligible | [ 0.09, 0.18] |
| | | Nocte+SMM | < 1e-10 | 0.29 | Small | [ 0.25, 0.33] |

**Table 7: One-sided Wilcoxon signed-rank test comparing potential adverse effects on false-positives over the baseline ($\alpha = 0.01$, $H1 = greater$), with effect size (Cliff's $\delta$), interpretation ($\delta$ Int.), and 95 % confidence interval ($CI$). High $p$ and low $\delta$ are ideal.**

| Dataset | Approach | $p$ | $\delta$ | $\delta$ Int. | $\delta$ 95 % CI |
|---|---|---|---|---|---|
| BoardGame | TSN | < 1e-10 | 0.02 | Negligible | [ 0.01, 0.03] |
| | SMM | < 1e-10 | 0.07 | Negligible | [ 0.07, 0.08] |
| | Nocte | < 1e-10 | 0.24 | Small | [ 0.24, 0.25] |
| PROGpedia19 | TSN | 8.3e-05 | 0 | Negligible | [-0.08, 0.09] |
| | SMM | 0.011 | 0.02 | Negligible | [-0.07, 0.10] |
| | Nocte | 0.0098 | 0.03 | Negligible | [-0.05, 0.12] |
| PROGpedia56 | TSN | 0.23 | -0.03 | Negligible | [-0.11, 0.04] |
| | SMM | 0.018 | 0.02 | Negligible | [-0.06, 0.09] |
| | Nocte | 3.4e-08 | 0.14 | Negligible | [ 0.06, 0.21] |
| TicTacToe | TSN | < 1e-10 | 0.01 | Negligible | [ 0.01, 0.01] |
| | SMM | < 1e-10 | 0.05 | Negligible | [ 0.04, 0.05] |
| | Nocte | < 1e-10 | 0.23 | Small | [ 0.22, 0.23] |
| | Nocte+SMM | < 1e-10 | 0.28 | Small | [ 0.28, 0.29] |

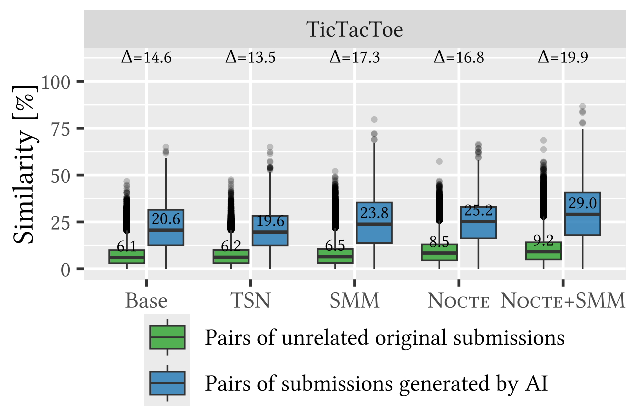


**Figure 6: Similarities for unrelated human programs (green) and AI-generated programs (orange).**

## 5.3 Discussion

In this section, we discuss our findings and their implications.

*Algorithmic Obfuscation.* The evaluation results for the algorithmic obfuscation attacks—insertion-based and refactoring-based—show that Nocte outperforms the state of the art. The normalization provided by Nocte achieves near-optimal separation between plagiarism instances and unrelated original programs, reverting virtually all effects of the obfuscation. The results show that existing approaches lack resilience to refactoring-based obfuscation; Nocte overcomes this limitation. Since the defense mechanisms for insertion-based and refactoring-based obfuscation are separate steps in the pipeline of Nocte, a combined obfuscation attack using both strategies is no more effective than either attack alone.

The mechanisms used to automatically obfuscate the submissions bear no direct relation to the defense mechanisms of Nocte: Insertion-based obfuscation is a real-world attack [18] which is straightforward to implement but requires data flow analysis to reverse. For refactoring-based obfuscation, we employed refactorings that conceptually map to the counter-transformations, but implemented them independently: The obfuscation is based on the code-level *Spoon* meta-programming library for code transformation, while Nocte operates at the graph level via the CPG.

*AI-based Obfuscation.* We observe that AI-based obfuscation is less effective than algorithmic obfuscation. Even for the baseline, there is a limited overlap between plagiarism and originals, especially for the PROGpedia datasets (median similarity for PROGpedia56: Insertion: 19.6 %, Refactoring: 18.2 %, AI-based plagiarism: 60.8 %, see Figs. 3, 4, 5). This high baseline similarity of AI-obfuscated programs limits improvement across all approaches.

The high variance in similarity suggests that AI-based obfuscation is inconsistently effective and thus less reliable. Students do not know the overall similarity distribution, so they do not know what obfuscation level is sufficient to evade detection. Moreover, AI-based obfuscation does not guarantee the preservation of the original program behavior.

Despite these limitations of AI-based obfuscation, our results indicate that AI-based obfuscation attacks remain an insufficiently addressed challenge in plagiarism detection. No approach we evaluate provides a notable improvement over the baseline. Evidently, further research is required to adapt these approaches for AI-based obfuscation. TSN only addresses insertion-based and reordering-based obfuscation and is thus conceptually limited. While SMM operates heuristically, providing limited resilience for this obfuscation attack, it offers no inherent room to adapt its design to specific attack scenarios.

For Nocte, the current resilience against AI-based obfuscation is also limited. The results indicate that some of the modifications by GPT-4 are currently not covered by our set of transformations. In an analysis of the generated programs, we identified some of these additional obfuscation strategies, such as (a) loop changes, (b) algorithm changes, and (c) usage of API function calls. We can easily extend Nocte for additional refactorings, e.g., (a), whereas reimplementation techniques and semantic clones, such as (b) and (c), are out of scope for Nocte. Nonetheless, in light of the variety of modifications GPT-4 used to obfuscate programs, further analysis of the resulting plagiarism instances is required to extend the transformation set and thereby strengthen the resilience of Nocte against AI-based obfuscation.

*AI-generated Programs.* Although Nocte was not designed to detect AI-generated programs, it notably improves the similarity of generated programs and thus increases the chance of their detection. For better results, we recommend using it in combination with SMM. Interestingly, programs generated with GPT-4 exhibit higher similarity among each other than human programs; this is the case across all analyzed approaches, and may be largely attributed to the inherent determinism of large language models [60].

However, it is crucial to note that software plagiarism detectors are not the ideal solution for detecting AI-generated code. Current research points towards signature- or watermark-based methods, for example, based on recognizing specific patterns or characteristics inherent to AI-generated content [22, 73]. It should also be noted that cheating via AI-generated programs is only viable if the generated programs are valid. Currently, we observe that AI-generated programs work reliably only for small-scale tasks; however, they exhibit a higher structural similarity than programs created by humans (see Section 5.2.4). To exploit this, instructors can add AI-generated programs to the submission set for the plagiarism check [51]. To address the rapid developments of generative AI, future research is necessary.

*Emerging Threats.* To address emerging threats, additional defense mechanisms are needed for both AI-based obfuscation and AI-generated code. As discussed, signature- or watermark-based methods constitute promising options. Due to recent advances, another direction is to incorporate AI itself into detection techniques. However, several challenges remain: 1) limited explainability, which is crucial for ethical transparency; 2) the need for traceability beyond similarity scores, such as fragment matching and visualization, which vector embeddings, for example, lack; 3) scarce real-world educational data, increasing the risk of overfitting; and 4) ethical concerns, as ML-based methods may produce false positives; LLMs, for example, are known to be prone to hallucination. Despite these challenges, exploring the incorporation of AI remains promising.

*Impact on False Positives.* Plagiarism detectors must achieve a low false-positive rate to be ethical and viable solutions for educators. Thus, we also analyzed the impact of all approaches on the unrelated, original programs. Table 7 shows the result of the statistical tests regarding a potential adverse impact on these programs compared to the baseline. Note that we are testing for an increase in similarity relative to the original value; Thus, a high $p$-value and low effect size are desirable. While all approaches show statistically significant effects on unrelated programs in some datasets, the effect sizes show that the effects are practically insignificant: SMM and Nocte have *small* to *negligible* effect sizes; TSN has *negligible* effect sizes. Any normalization technique is inherently bound to have *some* effect on all programs; nevertheless, the median similarity differences show that normalization generally provides more substantial separation between plagiarism pairs and unrelated pairs.

**Table 8: Runtime of 10 runs of Nocte for each dataset.**

| Task | Mean time | Std.dev. | $n$ | Total LOC |
|---|---|---|---|---|
| BoardGame | 09:20 min | 00:06 min | 407 | 660 KLOC |
| PROGpedia19 | 00:33 min | 00:02 min | 161 | 25 KLOC |
| PROGpedia56 | 00:24 min | 00:01 min | 164 | 17 KLOC |
| TicTacToe | 03:44 min | 00:08 min | 845 | 200 KLOC |

*Impact on Performance.* On a consumer notebook (Ryzen 7 Pro, 64GB RAM), we measured the runtimes for all datasets (10 runs each). Table 8 shows the results. As plagiarism detection occurs only a few times per semester, these low runtimes are negligible.

## 5.4 Threats to Validity

In the following section, we discuss how we address threats to validity as outlined by Wohlin et al. [71], Runeson and Höst [59].

*Internal Validity.* To ensure internal validity, we use JPlag as a foundation for all approaches and keep all other factors unchanged. The original programs on which the automatic obfuscation attacks are applied to generate plagiarized instances are randomly selected from each dataset to avoid cherry-picking. For AI-based obfuscation, we use 16 prompts derived through systematic prompt engineering to achieve consistently reproducible results. While the refactoring-based obfuscation mechanism and the normalization engine cover conceptually equivalent refactorings, both were designed independently and rely on different technologies to ensure meaningful results.

*External Validity.* To ensure external validity, we base our evaluation datasets on real-world student submissions from multiple sources, each with varying corpus sizes, submission lengths, code complexity, and assignment characteristics. While Nocte was integrated into JPlag for our evaluation, its conceptual design is independent from JPlag; thus, it can be incorporated into any other token-based plagiarism detection system such as Moss and Dolos. Furthermore, it is possible to combine Nocte with other optimization approaches, as demonstrated with SMM.

While the selected refactoring transformations (**C2**) are listed as common in reference literature, they do not cover all strategies students may use to obfuscate plagiarized programs. A thorough study of real student plagiarism cases may address this gap.

*Construct Validity.* To ensure construct validity, we employ the methodological standards of prior software plagiarism detection research. This is reflected in the fact that we carefully prepared the labeled datasets, used a GQM plan [6, 7], selected the evaluation metrics, and established an approach-independent ground truth.

*Reliability.* For reliability, we provide the code for our approach and the evaluation data as supplementary material [40].

# 6 Related Work

In this section, we discuss various related works.

*Plagiarism Detection and Obfuscation Resilience.* Sağlam et al. [61] use PDGs as an intermediate representation to detect dead code and counter the reordering of independent statements. While Nocte also addresses these obfuscation attacks, it also counters refactoring-based attacks. Karnalim [27] uses tokens to represent Java Bytecode instructions instead of structural code elements. This approach is immune to textual changes like renaming, but also against semantic-preserving instruction replacement. However, it is conceptually limited to JVM-based languages. Sağlam et al. [64] counter various obfuscation attacks by heuristically combining adjacent token matches separated by only a few tokens. While this provides broad resilience, it is less suited against complex, refactoring-based obfuscation that Nocte targets (Section 5.2.2). Maisch et al. [39] introduce tolerant token matching, an alternative approach to dealing with syntactic variety by accepting different, but conceptually similar token types as matches. While this approach is effective, it faces the problem of interdependent syntactic structures; e.g., `for` loops usually contain a loop variable declaration, whereas `while` loop variables must be declared before the loop. Nocte's structural normalization solves this problem. Novak [47] proposes preprocessing code by, e.g., removing comments and common code to emphasize more original fragments of code. We build on these ideas by extracting 14 concrete refactorings (Section 4) and integrating them into a flexible framework. Moreover, our approach does not modify the code itself, as it operates on CPGs and normalizes the internal code representation of plagiarism detectors.

In addition to token-based approaches, there are some graph-based approaches to plagiarism detection [12, 19, 34, 49]. While graph-based approaches may be resilient against some obfuscation attacks like dead code insertion, they are not feasible in practice [34] due to the $\mathcal{NP}$ nature of subgraph isomorphism [37, 43, 66]. Our approach leverages the potential of graph-based techniques while preserving the scalability of token-based techniques.

*Automated Refactorings.* There is extensive research on automated refactoring of source code [5]. Liu et al. [35] perform refactorings on code to achieve architectural consistency. Lin et al. [33] apply design patterns to code via automated refactorings. Alizadeh et al. [3] propose an interactive tool to take developers' feedback on proposed refactorings into account. These approaches refactor source code to make it more understandable and support the development process, and thus must respect developers' needs [68], such as providing understandable transformations. In contrast, Nocte applies refactorings to normalize code towards a common representation, enabling assessment of code pairs.

*Code Property Graphs.* We also relate to research on CPGs beyond plagiarism detection. Yamaguchi et al. [72] introduce CPGs for software vulnerability analysis, modeling common vulnerabilities as templates that can be detected by traversing a program's CPG. Based on this, neural networks have been used to learn these vulnerable patterns in CPGs [11, 74]. Other approaches aim to detect vulnerabilities using large language models [36]. These approaches use CPGs for security analysis, whereas our use case is plagiarism detection.

*Software Clone Detection.* Software systems often contain multiple similar code fragments called *code clones* [58], which impede modern software development [25]. There is extensive research about the detection of these clones [2, 56, 67]. While the detection mechanisms for clones and plagiarism are similar [49], they ultimately differ [41]: Clone detection seeks similarities in a single program, while plagiarism detection compares sets of programs. Moreover, clones are created accidentally [25], while plagiarism is a deliberate act. Most importantly, clone detectors are insufficient for plagiarism detection because they do not account for adversarial scenarios [60]. Thus, they are vulnerable to obfuscation attacks [62].

However, some works in code clone detection specifically create a normalized representation of code before performing similarity detection, thus establishing a relation to our work. Davis and Godfrey [17] propose normalizing code through intermediate representations to abstract away non-significant syntactic differences. To that end, they analyze compiler-generated assembly code. However, this approach relies on binaries and faces challenges due to platform-specific compilation artifacts. Kononenko et al. [29] show in a study that clone detection results can differ substantially between source code and Java bytecode, especially in large systems. Selim et al. [65] improve the detection of Type 3 clones (*near-miss clones*) [56] with a hybrid approach that analyzes both Java source code and compiler-generated intermediate representations. The approach benefits from the compiler normalization but is inherently specific to Java. Ragkhitwetsagul and Krinke [55] propose decompiling Java bytecode back into source code to allow existing Java-based clone detectors to identify clones that may be obscured in the original source. While this pre-processing step could be applied for plagiarism detection, it would obscure idiosyncrasies used as evidence during misconduct investigations. In contrast, our approach can be implemented for other languages and is platform-independent. Moreover, it only normalizes the program's internal representation, thus preserving traceability and explainability.

# 7 Conclusion

This paper introduces Nocte, a CPG-based framework designed to enhance token-based plagiarism detectors against refactoring-based obfuscation attacks (**C1**). Our approach significantly improves obfuscation resilience by applying targeted graph transformations (**C2**) before tokenization while maintaining a minimal impact on unrelated programs. Our evaluation (**C3**) is based on real-world datasets. The results show that Nocte significantly outperforms the state of the art for insertion-based and refactoring-based obfuscation attacks. As code obfuscation techniques continue to evolve, our framework provides an extensible solution to strengthen academic integrity in programming education. As future work, a controlled study where students use AI tools to plagiarize given programs could provide deeper insight into obfuscation strategies beyond our current selection.

# Acknowledgments

This work was supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF) and the BMBF (German Federal Ministry of Education and Research) grant number 16KISA086 (ANYMOS).

# References


[1] Alex Aiken. 2022. *MOSS Software Plagiarism Detector Website*. Stanford University. http://theory.stanford.edu/~aiken/moss/ Accessed on Oct. 22, 2025.

[2] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A Systematic Review on Code Clone Detection. *IEEE Access* 7 (5 2019), 86121–86144. doi:10.1109/ACCESS.2019.2918202

[3] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ó Cinnéide, Ali Ouni, and Yuanfang Cai. 2020. An Interactive and Dynamic Search-Based Approach to Software Refactoring Recommendations. *IEEE Transactions on Software Engineering* 46, 9 (2020), 932–961. doi:10.1109/TSE.2018.2872711

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

[5] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. 2020. Automatic software refactoring: a systematic literature review. *Software Quality Journal* 28, 2 (2020), 459–502. doi:10.1007/s11219-019-09477-y

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

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

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

[9] Miguel A. Botto Tobar, Mark G.J. van den Brand, and Alexander Serebrenik. 2022. Cross-Language Plagiarism Detection: Methods, Tools, and Challenges: A Systematic Review. *International Journal on Advanced Science, Engineering and Information Technology* 12, 2 (20 5 2022), 589–599. doi:10.18517/ijaseit.12.2.14711

[10] Tracy Camp, W. Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. 2017. Generation CS: The Growth of Computer Science. *ACM Inroads* 8, 2 (5 2017), 44–50. doi:10.1145/3084362

[11] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48, 9 (2022), 3280–3296. doi:10.1109/TSE.2021.3087402

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

[13] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114, 3 (11 1993), 494–509. doi:10.1037/0033-2909.114.3.494

[14] Georgina Cosma and Mike Joy. 2008. Towards a Definition of Source-Code Plagiarism. *IEEE Transactions on Education* 51, 2 (5 2008), 195–200. doi:10.1109/te.2007.906776

[15] Fintan Culwin and Thomas Lancaster. 2001. Plagiarism issues for higher education. *VINE* 31, 2 (01 1 2001), 36–41. doi:10.1108/03055720010804005

[16] Marian Daun and Jennifer Brings. 2023. How ChatGPT Will Change Software Engineering Education, In Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (Turku, Finland), Mikko-Jussi Laakso, Mattia Monga, Simon, and Judithe Sheard (Eds.). *Annual Conference on Innovation and Technology in Computer Science Education*, 110–116. doi:10.1145/3587102.3588815

[17] Ian J. Davis and Michael W. Godfrey. 2010. From Whence It Came: Detecting Source Code Clones by Analyzing Assembler. In *2010 17th Working Conference on Reverse Engineering*. 242–246. doi:10.1109/WCRE.2010.35

[18] Breanna Devore-McDonald and Emery D. Berger. 2020. Mossad: Defeating Software Plagiarism Detection. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 138 (11 2020), 28 pages. doi:10.1145/3428206

[19] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (7 1987), 319–349. doi:10.1145/24039.24041

[20] Tomáš Foltýnek, Terry Ruas, Philipp Scharpf, Norman Meuschke, Moritz Schubotz, William Grosky, and Bela Gipp. 2020. Detecting Machine-Obfuscated Plagiarism, In Sustainable Digital Communities, Anneli Sundqvist, Gerd Berget, Jan Nolin, and Kjell Ivar Skjerdingstad (Eds.). *iConference* 12051, 816–827. doi:10.1007/978-3-030-43687-2_68

[21] Henner Gimpel, Kristina Hall, Stefan Decker, Torsten Eymann, Luis Lämmermann, Alexander Mädche, Maximilian Röglinger, Caroline Ruiner, Manfred Schoch, Mareike Schoop, Nils Urbach, and Steffen Vandrik. 2023. Unlocking the power of generative AI models and systems such as GPT-4 and ChatGPT for higher education. (3 2023). http://opus.uni-hohenheim.de/volltexte/2023/2146/

[22] Zhengyuan Jiang, Jinghuai Zhang, and Neil Zhenqiang Gong. 2023. Evading Watermark based Detection of AI-Generated Content, In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (Copenhagen, Denmark), Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). *Conference on Computer and Communications Security*, 1168–1181. doi:10.1145/3576915.3623189

[23] Mike Joy and Micheal Luck. 1999. Plagiarism in programming assignments. *IEEE Transactions on Education* 42, 2 (5 1999), 129–133. doi:10.1109/13.762946

[24] JPlag. 2025. *JPlag GitHub Repository*. GitHub. https://github.com/jplag/JPlag Accessed on Oct. 22, 2025.

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

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

[27] Oscar Karnalim. 2016. Detecting source code plagiarism on introductory programming course assignments using a bytecode approach. In *2016 International Conference on Information & Communication Technology and Systems (ICTS)*. IEEE, 63–68. doi:10.1109/icts.2016.7910274

[28] Mohammad Khalil and Erkan Er. 2023. Will ChatGPT get you caught? Rethinking of Plagiarism Detection. *Interacción* 14040 (2 2023), 475–487. doi:10.48550/arXiv.2302.04335 10.48550/arXiv.2302.04335.

[29] Oleksii Kononenko, Cheng Zhang, and Michael W. Godfrey. 2014. Compiling Clones: What Happens?. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 481–485. doi:10.1109/ICSME.2014.78

[30] Cynthia Kustanto and Inggriani Liem. 2009. Automatic Source Code Plagiarism Detection, In 2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, Haeng-Kon Kim and Roger Y. Lee (Eds.). *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, 481–486. doi:10.1109/SNPD.2009.62

[31] Thomas Lancaster and Fintan Culwin. 2004. A Comparison of Source Code Plagiarism Detection Engines. *Computer Science Education* 14, 2 (6 2004), 101–112. doi:10.1080/08993400412331363843 arXiv:https://doi.org/10.1080/08993400412331363843

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

[33] Yun Lin, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun Zhao. 2016. Interactive and guided architectural refactoring with search-based recommendation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 535–546. doi:10.1145/2950290.2950317

[34] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis, In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (Philadelphia, PA, USA), Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos (Eds.). *Knowledge Discovery and Data Mining*, 872–881. doi:10.1145/1150402.1150522

[35] Jingwen Liu, Wuxia Jin, Junhui Zhou, Qiong Feng, Ming Fan, Haijun Wang, and Ting Liu. 2024. 3Erefactor: Effective, Efficient and Executable Refactoring Recommendation for Software Architectural Consistency. *IEEE Transactions on Software Engineering* 50, 10 (2024), 2633–2655. doi:10.1109/TSE.2024.3449564

[36] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software* 212 (2024), 112031. doi:10.1016/j.jss.2024.112031

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

[38] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Transactions on Software Engineering* 43, 12 (12 2017), 1157–1177. doi:10.1109/TSE.2017.2655046

[39] Robin Maisch, Nathan Hagel, and Alexander Bartel. 2025. Towards Robust Plagiarism Detection in Programming Education: Introducing Tolerant Token Matching Techniques to Counter Novel Obfuscation Methods. In *Proceedings of the 6th European Conference on Software Engineering Education (ECSEE '25)*. Association for Computing Machinery, New York, NY, USA, 11–19. doi:10.1145/3723010.3723019

[40] Robin Maisch, Larissa Schmid, Timur Sağlam, and Nils Niehues. 2025. *Supplementary Material for "Same Same But Different: Preventing Refactoring Attacks on Software Plagiarism Detection"*. doi:10.5281/zenodo.17424417

[41] Leonardo Mariani and Daniela Micucci. 2012. AuDeNTES: Automatic Detection of TeNtative Plagiarism According to a REference Solution. *ACM Trans. Comput. Educ.* 12, 1, Article 2 (3 2012), 26 pages. doi:10.1145/2133797.2133799

[42] Eduard-Constantin Marin, Andrei Oțetea, and Alexandru-Corneliu Olteanu. 2025. A Comparative Study of Source Code Plagiarism Detection Tools for Programming Education. In *2025 25th International Conference on Control Systems and Computer Science (CSCS)*. 328–334. doi:10.1109/CSCS66924.2025.00055

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

[44] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. 2005. Formalizing Refactorings with Graph Transformations. *Journal of Software Maintenance and Evolution: Research and Practice* 17, 4 (2005), 247–276. doi:10.1002/smr.316

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

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

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

[48] Matija Novak. 2020. *Effect of source-code preprocessing techniques on plagiarism detection accuracy in student programming assignments*. Ph. D. Dissertation. University of Zagreb. Faculty of Organization and Informatics.

[49] Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review. *ACM Transactions on Computing Education* 19, 3, Article 27 (9 2019), 37 pages. doi:10.1145/3313290

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

[51] Ashley Pang and Frank Vahid. 2024. ChatGPT and Cheat Detection in CS1 Using a Program Autograding System. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 367–373. doi:10.1145/3649217.3653558

[52] Dieter Pawelczak. 2018. Benefits and drawbacks of source code plagiarism detection in engineering education, In 2018 IEEE Global Engineering Education Conference (EDUCON). *IEEE Global Engineering Education Conference*, 1048–1056. doi:10.1109/EDUCON.2018.8363346

[53] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. 2006. *Spoon: Program Analysis and Transformation in Java*. Research Report RR-5901. Inria. https://inria.hal.science/inria-00071366

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

[55] Chaiyong Ragkhitwetsagul and Jens Krinke. 2017. Using compilation/decompilation to enhance clone detection. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*. 1–7. doi:10.1109/IWSC.2017.7880502

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

[57] J. Romano, J.D. Kromrey, J. Coraggio, and J. Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?. In *annual meeting of the Florida Association of Institutional Research*. 1–3.

[58] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (5 2009), 470–495. doi:10.1016/j.scico.2009.02.007

[59] Per Runeson and Martin Höst. 2008. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (12 2008), 131–164. doi:10.1007/s10664-008-9102-8

[60] Timur Sağlam. 2025. *Mitigating Automated Obfuscation Attacks on Software Plagiarism Detection Systems*. Ph. D. Dissertation. Karlsruhe Institute of Technology (KIT). doi:10.5445/IR/1000179018/v2

[61] Timur Sağlam, Moritz Brödel, Larissa Schmid, and Sebastian Hahner. 2024. Detecting Automatic Software Plagiarism via Token Sequence Normalization, In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal). *International Conference on Software Engineering*, Article 113, 13 pages. doi:10.1145/3597503.3639192

[62] Timur Sağlam, Sebastian Hahner, Larissa Schmid, and Erik Burger. 2024. Automated Detection of AI-Obfuscated Plagiarism in Modeling Assignments, In Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training (Lisbon, Portugal). *SEET@ICSE*, 297–308. doi:10.1145/3639474.3640084

[63] Timur Sağlam, Sebastian Hahner, Larissa Schmid, and Erik Burger. 2024. Obfuscation-Resilient Software Plagiarism Detection with JPlag, In Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (Lisbon, Portugal). *ICSE Companion* 8, 264–265. doi:10.1145/3639478.3643074

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

[65] Gehan M.K. Selim, King Chun Foo, and Ying Zou. 2010. Enhancing Source-Based Clone Detection Using Intermediate Representation. In *2010 17th Working Conference on Reverse Engineering*. 227–236. doi:10.1109/WCRE.2010.33

[66] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.* 1, 1 (8 2008), 364–375. doi:10.14778/1453856.1453899

[67] G. Shobha, Ajay Rana, Vineet Kansal, and Sarvesh Tanwar. 2021. Code Clone Detection—A Systematic Review, In Emerging Technologies in Data Mining and Information Security, Aboul Ella Hassanien, Siddhartha Bhattacharyya, Satyajit Chakrabati, Abhishek Bhattacharya, and Soumi Dutta (Eds.). *Advances in Intelligent Systems and Computing*, 645–655. doi:10.1007/978-981-33-4367-2_61

[68] Gábor Szóke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2016. Designing and Developing Automated Refactoring Transformations: An Experience Report. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 693–697. doi:10.1109/SANER.2016.17

[69] Debora Weber-Wulff. 2019. Plagiarism detectors are a crutch, and a problem. *Nature* 567, 7749 (3 2019), 435–435. doi:10.1038/d41586-019-00893-5

[70] Konrad Weiss and Christian Banse. 2022. A Language-Independent Analysis Platform for Source Code. arXiv:2203.08424 [cs.CR] https://arxiv.org/abs/2203.08424

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

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

[73] Xuandong Zhao, Prabhanjan Vijendra Ananth, Lei Li, and Yu-Xiang Wang. 2024. Provable Robust Watermarking for AI-Generated Text. In *The Twelfth International Conference on Learning Representations*. OpenReview.net. doi:10.48550/arXiv.2306.17439

[74] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f-Paper.pdf