Criminal Minds: How First-Year CS Students Plagiarize Code

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

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

Richard Glassey
glassey@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Dominik Fuchß
dominik.fuchss@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

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

Haoyu Liu
haoyu.liu@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Anne Koziolek
anne.koziolek@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Abstract

While academic integrity is essential in higher education, plagiarism by students in programming courses is a prevalent challenge. Plagiarism detectors are widely used, yet it is unclear if they match the ways students actually obfuscate copied code. Existing evaluation datasets are limited, often private, or rely on synthetically generated plagiarism instances. In a collaboration between two universities, we conduct an empirical study that challenges students at both institutions to plagiarize given solutions and document their changes. Using this dataset, we analyze their success in evading detection, present common plagiarism strategies, and provide a reliable resource for future research on academic integrity. While only few submissions plagiarize successfully, our participants incorporate a large variety of creative manual and LLM-based strategies.

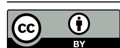
CCS Concepts

• **General and reference** → **Empirical studies**; • **Information systems** → **Near-duplicate and plagiarism detection**; • **Social and professional topics** → **Computing education**; • **Computing methodologies** → *Artificial intelligence*; • **Software and its engineering**;

Keywords

plagiarism detection, large language models, LLMs, software engineering education, SE education, computer science education

ACM Reference Format:
Robin Maisch, Larissa Schmid, Richard Glassey, Dominik Fuchß, Nils Niehues, Haoyu Liu, and Anne Koziolek. 2026. Criminal Minds: How First-Year CS Students Plagiarize Code. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 5–9, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3803437.3805789



This work is licensed under a Creative Commons Attribution 4.0 International License.
FSE Companion '26, Montreal, QC, Canada
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2636-1/26/07
https://doi.org/10.1145/3803437.3805789

1 Introduction

Academic integrity is a key principle of higher education. Students are expected to submit their own work that fairly represents their own skill and effort. In programming courses and other technical assignments, violations of this expectation, including deliberate collaboration or copying, present serious challenges for instructors and institutions [5, 18, 24]. To detect suspicious submission pairs, automatic plagiarism detection systems and manual inspection are widely used approaches [7].

Course instructors rely on plagiarism detection systems because they scale to large classes and provide a ranked shortlist of candidates for further human review, whereas manual inspection is costly and difficult to perform exhaustively in such dimensions [3, 17]. Manual candidate inspection remains essential to determine intent and context (e.g., allowed reuse, template code provided for the task, or common idioms). Most detection systems operate on structural representations of source code [25, 26], rendering them robust to superficial obfuscations such as identifier renaming or whitespace modifications. Based on the assumption that effective manual obfuscation requires similar effort and technical skill as completing the assignment themselves, academic integrity was guaranteed.

However, recent studies have broken this assumption and demonstrated that plagiarism detection can be avoided through automated *obfuscation attacks* [2, 7, 10, 28]. Coupled with the rapidly advancing obfuscation capabilities of large language models (LLMs) [6, 12, 37], this has prompted a growing body of research focused on developing defense mechanisms against such attacks [16, 22, 31–33].

On the other hand, empirical evidence on how students attempt to avoid detection when plagiarizing is limited. While there are some datasets commonly used for the evaluation of plagiarism detection systems, they suffer from different drawbacks: Institutional corpora are often private, as they are subject to data protection [16, 31], while public corpora may contain incomplete or incorrect programs and rely on self-labeling [27]. Consequently, it remains unclear which obfuscation strategies are used in practice and how robust commonly used detection systems and their defense mechanisms are against these strategies.

This paper presents an empirical study of student plagiarism in programming assignments at two European institutions. The students were instructed to plagiarize and provided loose guidelines about their methods, an initial solution to plagiarize from, and a form to document their steps to obfuscate their plagiarism. Building on this corpus of submissions, we analyze students' behavior to characterize common strategies they use when copying and adapting solutions from peers. Our goal is not only to quantify prevalence but also to (i) provide practical insights into the concrete transformations students apply, (ii) evaluate how well existing detection mechanisms surface submissions that use these transformations, and (iii) provide a reliable dataset with realistic solutions and clear ground truth. To remain useful in a rapidly changing landscape, including evolving LLM capabilities, we lay out a repeatable study design that can be replicated on future cohorts and assignments.

We make three contributions:

- **C1** *Study design.* We propose a repeatable study to collect student submissions, compute similarity scores, and establish ground truth through manual annotations to enable reproducible research on academic integrity.
- **C2** *Dataset.* We release the multi-institutional dataset of original and plagiarized programs with realistic solutions and similarity measurements for evaluating plagiarism detection [21].
- **C3** *Empirical analysis.* We analyze the strategies students apply when plagiarizing and assess whether similarity detection mechanisms remain effective.

Our dataset analysis aims to answer these four research questions:

- *RQ1:* How effectively did the participants plagiarize?
- *RQ2:* Which methods, tools, and sources do the participants use?
- *RQ3:* How do the resulting plagiarisms compare to the original submissions?
- *RQ4:* How does the success of the plagiarism approach relate to the participant's subjective perception of the result?

2 Limitations of Plagiarism Datasets

The effectiveness of plagiarism detection systems and their defense mechanisms against obfuscation attacks depends on their ability to determine high similarity between plagiarized programs and low similarity among independent originals [34]. This presumes that the underlying programming task admits a sufficiently large and diverse solution space, such that independent student submissions naturally exhibit low similarity to one another and plagiarized submissions emerge as statistical outliers. A defense mechanism that affects the similarity of obfuscated and unaltered submissions to the same extent offers little practical benefit, as it impedes educators' ability to identify plagiarized programs using similarity outliers. Consequently, potential defense mechanisms must be empirically evaluated using datasets that include both genuine student submissions and known instances of plagiarism.

While datasets that contain both student submissions and identified instances of obfuscated plagiarism are available, they are often limited in sample size and quality [30]; these limitations make it difficult to reliably evaluate the effects of defense mechanisms. Another typical restriction is that the task's solution space may be overly constrained. Thus, to enable meaningful evaluation, it is common practice to rely on authentic student-submission datasets of sufficient size and complexity and to synthetically generate obfuscated plagiarized versions for evaluation. There are several ways to achieve such semantics-preserving obfuscation. One approach involves manually modifying submissions using semantics-preserving transformations of varying scope and impact [13]. More sophisticated approaches employ automated obfuscation attacks, which can, for example, insert dead code [7], reorder independent statements [31], or apply a range of refactorings [22]. More recently, LLMs have been used for obfuscation, either by applying prompt-based refactorings or by generating complete submissions from scratch based on task descriptions [20, 22].

While these approaches provide useful evaluation data points, they are inherently biased, as the generated obfuscations are derived from the very attack models that the defense mechanisms are designed to counter. Consequently, it remains unclear whether these techniques accurately reflect how students actually plagiarize code in practice. In particular, there is limited empirical understanding of whether students primarily rely on superficial modifications, such as whitespace changes or identifier renaming, or engage in more substantial refactorings, either manually or with the aid of automated tools. Likewise, the extent to which students employ LLMs, as well as the particular ways in which these models are used for plagiarism obfuscation, remain largely unexplored.

3 Study Design

Our study aims to empirically characterize how novice programmers plagiarize source code when explicitly instructed to do so, focusing on their success in obfuscating their plagiarism, the obfuscation strategies they use, and their use of generative AI. We conduct the study at KIT (Germany) and KTH (Sweden) to increase external validity across institutional contexts.

Overview. The foundational idea of our study is to deliberately have students plagiarize code in a time-boxed session, simulating scenarios in which we observe plagiarism typically occurring: Students find themselves close to the deadline of an assignment, without success in their own attempts to solve it. In the face of severe time pressure and risk of failing the course, they obtain a working program from a fellow student. Attempting to bypass plagiarism detection—which they are fully aware is routinely used in their course—they spend the remaining time adapting the program to obfuscate the relation of their submission to the original.

We stage this scenario in an on-site group session with first-semester students, where we provide each participant with one working program solution to plagiarize, aiming to make it unrecognizable while still maintaining its function, yielding a valid *plagiarized* submission. The *original* program solutions are randomly selected from a small set of original submissions that all implement the same problem statement. As the instructions are unknown to participants before the session, they are challenged to come up with suitable obfuscation strategies on the spot.

3.1 Group Design

We divide the participants into groups according to two controlled variables: *instruction* (*A* and *B*) and *original submission* (*O1*, *O2*, ...). Additionally, the uncontrolled variable of the participants' *university* may influence the results.

Table 1: Assessment of KTH's participants' distribution over four experience groups, ranging from beginners (G0) to students with considerable skill prior to the course (G3).

Group	Σ	G0	G1	G2	G3
Enrolled course participants	198	40	100	48	10
Experiment registrants	27	2	12	8	5
Experiment attendants	16	2	5	7	2

We design two sets of instructions outlining the methods and tools participants should use. Group *A* is instructed to use any method that comes to mind, while group *B* is instructed to use LLM-based tools specifically, but not exclusively. As LLM usage increasingly threatens academic integrity, we aim to generate unbiased results (Group *A*) while also ensuring a sufficient number and variety of LLM-based plagiarized solutions (Group *B*). This allows us to assess how and to what extent participants use LLMs to cheat.

We provide a number of original submissions *O1–ON* to plagiarize, limiting the overall influence of each original's particular properties on the resulting plagiarized submissions. The number *N* of originals is determined based on the number of registrants to ensure meaningful group sizes of 5 to 10 participants.

3.2 Participant Recruitment

The study targeted first-term computer science students participating in introductory programming lectures, as their submissions are usually subject to plagiarism detection. The authors advertised the study to all such students of their respective universities during the lecture, by e-mail, and on online learning platforms. At both universities, free snacks and drinks were promised to the students as an incentive to participate. While we encouraged students to register in advance, they could also participate without prior registration. Throughout the recruitment, it was clearly communicated that participation was voluntary, and that neither participation nor non-participation, nor the study results would in any way influence grades in the course.

No screening was performed for prior programming experience beyond lecture enrollment. However, participants in voluntary studies are likely biased toward high motivation. Thus, we use the participants' data from the respective ongoing courses to assess their performance relative to the overall student body.

At KIT and KTH, 45 out of 732 and 27 out of 198 students registered for the study, respectively. At KTH, students are divided into four groups based on prior skills and assigned tasks that challenge them at their level, shown in Table 1. In our sample of attendants, the two more advanced groups **G2** and **G3** are clearly overrepresented, as their members make up 56 % of participants, but represent only 30 % of the student cohort; this effect applies also for the *registrants* of KTH, but less pronounced.

At KIT, 732 students enrolled in the course have a non-zero score for their exercise submissions at the time of the study session. Table 2 shows their current scores compared to the registered students and the students who attended. While the registrants' scores are only slightly above average, the attendants' mean score is 9.1 %p higher than the overall student body's, indicating a bias toward high-performing students. The records show that some lower-performing students have quit the course since registering, and other students did not submit the last exercise sheet because they had already passed the exercise; thus, the score is not a proportional indicator of performance.

Table 2: Assessment of KIT's participants' performance in the exercise after 4 out of 5 assignments à 20 points (max. 80 points). 50 points are sufficient to pass the exercise. For each group, the minimum (min), arithmetic mean (μ), median (med), and maximum score (max) are given.

Group	Σ	min	μ	med	max
Active course participants	732	2.0	48.1	51.5	79.0
Experiment registrants	47	12.0	50.9	52.7	74.0
Experiment attendants	19	42.5	57.2	57.0	74.0

3.3 Task Selection

We selected the task *Dots and Boxes* from the former year's programming exercise at KIT. The goal of the task is to implement a turn-based strategy game for two players, played on the command line using predefined text commands, CLI arguments, and a defined output format. *Dots and Boxes* is a non-trivial, moderately sized task that aligns with the students' assumed skill set at the time of the study session at both universities. At that level of complexity, we expect a sufficiently large solution space to conduct meaningful plagiarism checks—a key prerequisite for creating a useful dataset of programs authentically plagiarized by students, as discussed in Section 2. Also, as it is a text-based game, the visual component and the examples provided in the task description make the underlying concepts more intuitive.

3.4 Acquisition of Originals

We intended for the participants to base their plagiarized programs on existing real-world student programs to ensure validity. Many of the teaching assistants (TAs) in the current term participated in the Programming exercise in the past winter term and implemented *Dots and Boxes* for it; therefore, we asked them to provide their solutions. As some TAs were hesitant to share code from back when they were less experienced, we allowed them to revise their programs. In total, we gathered 14 original, correct program submissions from the TAs, written in Java.

Next, we manually checked the submissions for issues that would make them unsuitable as original submissions for the study. For example, advanced language features such as streams or lambda expressions were not introduced in the course yet, so most participants would not be familiar with them. Solutions with such elements were ruled out to ensure they matched the participants' current skill set, leaving 10 original submissions. Based on the number of registrants (KIT: 47, KTH: 25), we set the number of submissions for the study to five, allowing us to assess and limit their overall influence on the plagiarism quality while maintaining meaningful group sizes. We randomly selected five original submissions *O1–O5* for participants to plagiarize; Table 3 lists some of their characteristics.

Table 3: Number of files (#Files), lines of documentation (LOD), lines of code (LOC), and functional quality (FQ) for the five selected original submissions and all 10 original submissions from the preliminary selection.

Original	#Files	LOD	LOC	FQ
O1	8	182	382	100 %
O2	8	225	339	100 %
O3	6	184	341	100 %
O4	8	242	350	100 %
O5	7	173	362	100 %
Overall	2 to 8	105 to 242	277 to 479	100 %

3.5 On-Site Study Session

In the following section, we describe the plan of the on-site session procedure with participants, including the materials they are provided and the artifacts they submit.

We decided on an on-site study to ensure participants focus on the study task without distractions by other tasks or duties. Also, the on-site setting allows us to answer questions, solve problems quickly, and hand out the promised snacks and drinks. The study sessions at KIT and KTH took place in suitably sized lecture halls, allowing participants to work alone or in teams without interfering with each other in the process. Two study supervisors are present in each university's session.

Entry Stage. Upon arrival, participants complete and sign a consent form and receive a study task description, both in the respective language of the courses. After that, participants are left to work on their own without interaction with the instructors.

Working Phase. The study task description includes the group-specific methodological instructions for groups *A* and *B*, as well as a link to access group-specific *online course material* for groups *O1–O5* (cf. Section 3.1). There is no explicit mention of the differing instructions, and only the assigned course material is visible to each student. In the online course, participants may access a personal Git repository containing all the material required to complete the study. This material includes the original submission *O1–O5*, and a task description of *Dots and Boxes* in the respective language of the courses. Also, pushing to the Git repository triggers a set of 23 functional tests. Upon completion, the test pass rate and detailed reports are displayed to participants on the online course page, allowing them to revise their submission and improve their score. For KTH, one test for stylistic criteria was omitted, as their usual submission assessment does not include such requirements.

Participants are also instructed to record the steps they take to alter the code. To ensure a uniform log format, we provide a simple web-based form to enter code locations and descriptions for each change. Each log item is expected to reflect either an individual change or an interconnected series of changes, in principle allowing to reconstruct the final plagiarized program. The change logs will be used to classify and quantify participants' strategies.

Final Stage. Before the end of the time box, or as soon as participants are satisfied with the plagiarism quality and test pass rate of their final submission, they complete an online survey covering various aspects of their experience during the study. This is the structure of the survey:

- Please enter your student ID(s).
- How much time did you spend creating the altered solution? *(number input)*
- Please rate your submission with respect to the criteria below. *(Likert scale 1–Low to 5–High)*
  - Code Quality: Object-Oriented Modeling
  - Code Quality: Style
  - Functional Correctness
  - Proportion of your original code in the submission
- If you used AI-based tools for any specific tasks in this study:
  - how useful did you find their responses? *(Likert scale 1–Not useful to 5–Very useful + option "Did not use AI tools")*
  - did you have to iteratively rephrase/clarify your prompt or tweak the result to be able to use it? *(Likert scale 1–Never to 5–Very often + option "Did not use AI tools")*
- How confident are you in handing in this solution as your own? That is, would you assume that it passes plagiarism detection without raising suspicion? *(Likert scale 1–Low to 5–High)*

The responses to these questions let us draw connections between factors such as perceived plagiarism quality and the measured similarity of the submission, and provide insight into their opinions on AI-based tools and the study design.

When participants collect their belongings and prepare to leave, the study supervisors try to ensure they have submitted their final plagiarized program, the corresponding change log, and their survey responses. Should any material be missing, they are reminded to submit before leaving. Because the supervisors' capacity is limited, this is only possible if not too many participants leave at once.

3.6 Test Run

To validate our study setup, we conducted a test run with four teaching assistants, walking them through the entire study process. Such test runs, or *pilot studies*, are a common practice to evaluate educational tools or studies in computer science education [11]. Based on the TAs's feedback, we restructured the study's task description and adapted its wording to avoid misunderstandings that occurred for the teaching assistants:

- We added the instruction to read the whole study task description before beginning the first step.
- We added a list of content for the task repository to help participants become familiar with each item and find the right items for each step more easily.
- To avoid overwhelming the participants with a series of paragraphs describing the entire study, we separated the study task description into numbered items that the participants could complete before tending to the next one.
- We included a note to avoid using the log form in an IDE HTML preview window, but in a proper browser. This ensured that the form worked as intended.

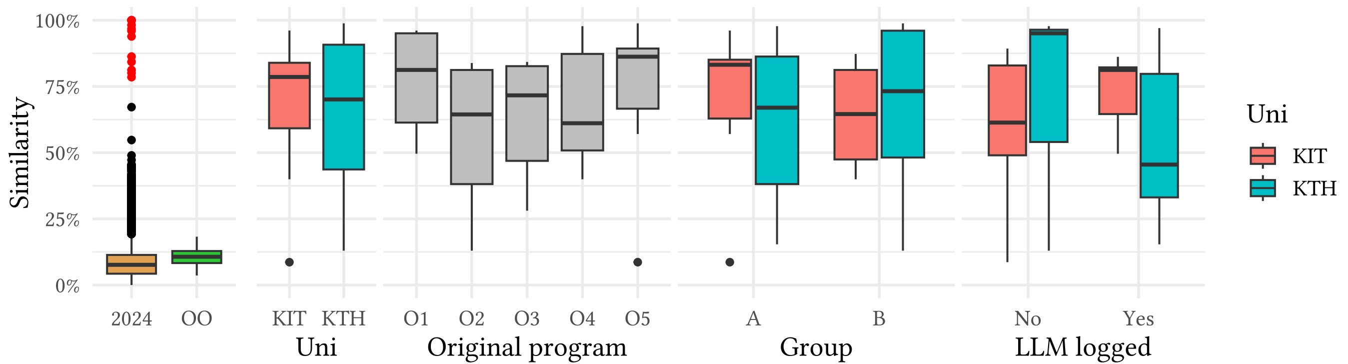


Figure 1: From left to right: The similarity among the submissions of 2024 (*2024*) and the TAs's original submissions (*OO*); the similarity of the plagiarized programs to their respective original grouped by their university, original submission, assigned study group, and whether LLM use was logged.

4 Results

In this section, we evaluate how successfully participants obfuscated plagiarism and which strategies they employed. We quantify plagiarism success using structural similarity scores from an automated plagiarism detection system and assess functional quality using pass rates in functional tests. To characterize the performed modifications, we analyze participants' change logs (types of changes and plagiarism levels). We additionally summarize participants' self-reported behavior and perceptions from the post-study survey. Finally, we compare the resulting submissions to their originals using lightweight software metrics (e.g., changes in LOC and LOD, and the number of files). We report these measures both in isolation and in relation to one another.

4.1 Methodology

With our evaluation, we aim to answer the research questions presented in Section 1. We now present how we analyze the result artifacts to this end.

From the experiment, we gathered 47 code repositories, 35 survey responses, and 26 log files. Of the code repositories, 12 had no submissions—these participants may have chosen to work with others instead. We remove these from the dataset, leaving only the 35 submissions that were actively changed.

*4.1.1 Plagiarism Success (**RQ1**).* For the first question, we use JPlag [29, 33], an open-source tool for automatic plagiarism detection, to identify structural similarities among the datasets of original and plagiarized programs. We analyze the similarity between original programs and their derived plagiarized programs (*OP*) and the similarity among the 10 original programs (*OO*), providing a baseline of independent, non-plagiarized programs. For comparison, we also include the similarity distribution for the class of 2024 at KIT (cf. Figure 1, *2024*), which clearly contained plagiarized programs, as determined by the instructors' inspection (shown in red).

Additionally, we assess the functional quality of the plagiarized programs by measuring their pass rates in functional tests. We use the same set of functional tests that were run after each push to the submission system for the participants to review their programs.

*4.1.2 Methods, Tools, and Sources (**RQ2**).* Next, we analyze the participants' approaches to altering the code. We use pairs of action words and object phrases to categorize change log items [41]. This allows us to classify and group them at a level that provides an accurate representation while avoiding clutter from overly nuanced distinctions. For this classification, we split multi-line log items into multiple single-line items and assign any number of action-object pairs to each one. Also, we tag mentions of LLM use.

We then use the classification system for software plagiarism by Faidhi and Robinson [9], extended by Karnalim [15], to classify the actions. This classification system defines 8 levels, arranged by scope and impact for plagiarism detection. Notably, each level contains all of its predecessors. **L0** describes verbatim copies. **L1** adds comment and whitespace modifications, such as formatting or Javadoc. **L2** includes changes to identifiers, e.g., of classes and variables. **L2.5** is an intermediate level added by Karnalim [15], addressing changes to the name or the structure of packages, as well as import statements. **L3** adds modifications to declarations, i.e., their location, order, visibility, and assigned value, as well as the usage of dummy variables. **L4** lifts changes to the module scope, including method extractions and declarations of dummy methods. **L5** addresses changes inside method bodies, e.g., altering of method calls, data types, operators, control structures, and the order of operations. Finally, **L6** adds changes in decision logic, such as introducing new control structures, shifting loop boundaries, or switching between loops and recursive code.

While levels **L0**–**L2** are limited to the textual code representation and thus inherently ineffective, levels **L2.5**–**L6** refer to increasingly finely-grained structural changes, from the package level down to statements and the decision logic. Structural changes affect the internal representation of code used by plagiarism detection systems such as Moss [1] and JPlag [29, 33], potentially impeding detection.

*4.1.3 Resulting Programs (**RQ3**).* Then, we apply software metrics to identify changes in lines of code (LOC), lines of documentation (LOD), including comments, and the number of files.

*4.1.4 Participants' Subjective Perception (**RQ4**).* Lastly, we analyze the survey results and relate participants' given data and personal impressions to their effectiveness at obfuscating plagiarism.

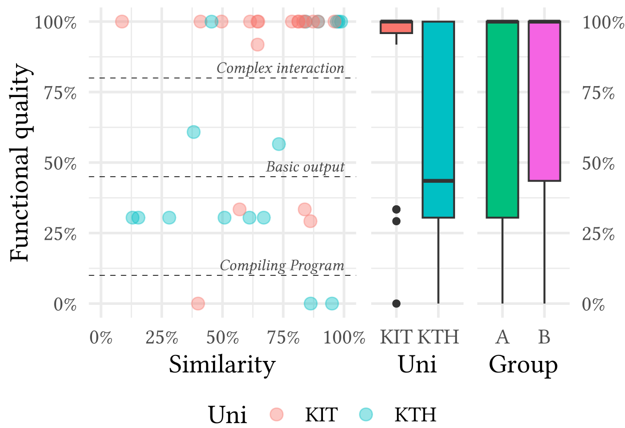


Figure 2: Correlation between similarity to the original submission and functional quality. Note: the top-right cluster at *similarity* > 75 % **and** *quality* = 100 % **contains 18 submissions.**

4.2 RQ1: Plagiarism Success

Similarity to the Original. Figure 1 shows the similarity distribution of pairs of submissions grouped by various criteria. The plot labeled *Uni* shows the similarity between the plagiarized submissions and their respective originals for universities KIT and KTH. With a median of 79 % and 70 % (mean: 69 % / 65 %), respectively, most participants from both universities were able to make changes that had a noticeable effect on similarity. However, the similarity distribution of pairs of originals (*OO*) is clearly separated from that of the *OP* groups: only three instances of plagiarism reduced the similarity below 26 %, the maximum similarity among originals. Between the interquartile ranges, there is a distance of 51.5 and 35.3 %p. Thus, in an unlabeled similarity analysis, nearly all *OP* pairs would come out on top, drawing immediate attention.

Functional Quality. Figure 2 shows the distribution of functional quality of the plagiarized programs, also related to their similarity to the original. Due to similar requirements, the functional pass rates of the plagiarized versions are divided into four levels, reflecting stages of program maturity. Most students of KIT succeeded in retaining functional quality but reduced similarity only slightly: 12 submissions exceeded a *test pass rate* of 80 % and a *similarity* of 59 %. However, the least similar submission also achieved full functional quality. KTH's participants, who were unfamiliar with the test framework and assessment mode, were generally more successful at reducing the program's similarity to the original, but functional quality was adversely affected; for example, the top 3 least similar programs could only pass basic functional tests. Overall, 21 participants retained close to full functional quality.

Answer to RQ1: *Our analysis shows that most participants failed to produce a plagiarized submission that satisfied both functional quality and low similarity to the original solution. Only a single participant achieved a solution that was both correct and exhibited low similarity to the original solution.*

Table 4: Number of occurrences (orange) and plagiarism level [9, 15] (green) of action words and object phrases manually assigned to the change log items. Counts of 1 are omitted.

Count	Action word	Object phrases
62	rename	variable(2) 18, constant(2) 14, class(2) 14, method(2) 9, field(2) 6, package(2.5)
53	change	documentation(1) 16, conditional(6) 4, expression(5) 4, logic(6) 4, API call(5) 3, call(5) 3, comment(1) 3, loop(6) 3, data type(5) 2, string(3) 2, structure(5) 2, switch case(5) 2, visibility(3) 2, everything(6), exception handling(5), statement(5)
19	add	documentation(1) 8, switch case(6) 4, API call(5) 2, method(4) 2, comment(1), conditional(6), statement(5)
16	reorder	if statement(5) 5, enum(3) 4, field(3) 2, method(3) 2, statement(3) 2, expression(5)
15	move	logic(4) 9, method(3) 4, class(2.5) 2
13	remove	comment(1) 3, method(4) 3, API call(5), call(5), class(2.5), code(6), constructor(4), documentation(1), emojis(3),
7	inline	class(2.5) 2, enum(5) 2, constant(5), method(4), variable(5)
7	reimplement	method(6) 5, class(6) 2
6	ask	LLM(0) 6
4	refactor	class(6) 2, code(6) 2
4	reformat	class(1) 2, code(1) 2
3	fix	error(6) 2, code(6)
3	rewrite	class(6) 3
1	repurpose	class(6)

4.3 RQ2: Methods

Types of Changes. In total, 213 action-object pairs were identified, listed in Table 4. Figure 3 shows the number of occurrences of changes accumulated for each level. The quantitative analysis suggests that textual changes (**L0**–**L2**), which are ineffective for obfuscating plagiarism, were most common. However, action-object pairs for higher plagiarism levels (e.g., *reimplement method*, *rewrite class*) often reflect changes of greater scope, effort, and impact on similarity; these factors are thus not fairly depicted relative to one another in this abstract overview. Taking the maximum level for each participant, 16 out of 26 participants report **L6** plagiarism.

Use of LLMs. 15 out of 26 logs mention the use of LLMs (A: 6/12; B: 9/14; KIT: 9/15, KTH: 8/11). Of these 15, 13 specify the model used: *ChatGPT* (11), *Gemini* (1), *Claude.ai* (1), *GitHub Copilot* (1), *Ecosia Chat* (1). Two logs mention multiple models: *ChatGPT* and *Claude.ai*, and *ChatGPT* and *GitHub Copilot*. For the remaining 11 participants, we cannot determine if they used LLM-based tools.

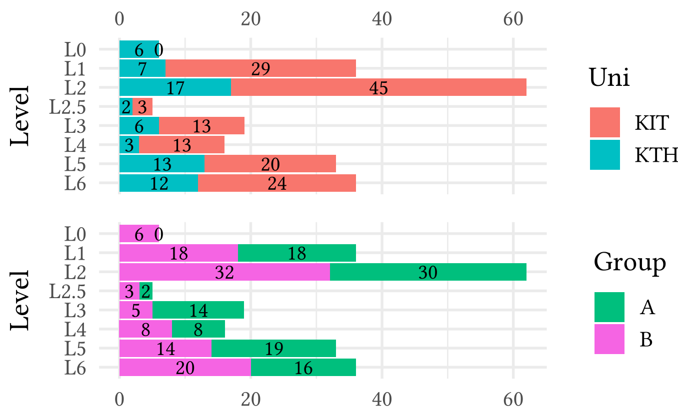


Figure 3: Distribution of plagiarism levels for action-object pairs, set by university (top) and study group (bottom).

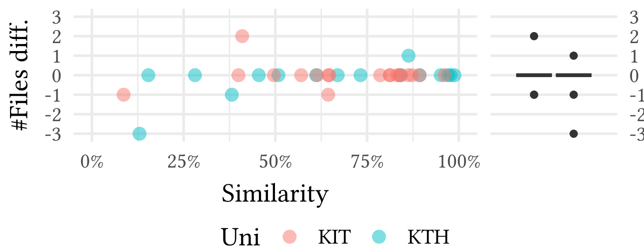


Figure 4: Difference in number of files of plagiarized submissions compared to the original submission.

The logs specify the prompts at different levels of detail. Many prompted for *refactoring*, *rewriting*, or *reimplementation*, sometimes specifying a concrete intention (e.g., "to make it look different but work the same"). Other logs mention prompting for specific changes such as changing or adding comments and documentation, replacing `if` and `switch` statements, inlining the functionality of a class, interchanging control flow and functional program elements (`Streams`, `Predicates`, etc.), fixing bugs and Checkstyle violations, changing visibility modifiers, or changing the order of methods, declarations, and `switch` cases. One participant prompted an LLM to "change everything", another to make a code snippet "a little ineffective". One participant specified in their prompts that they are participating in a study related to plagiarism. Another participant prompted for "fun ways to obscure copying someone else's code in Java". One student specified that they deliberately used the same LLM session for two consecutive prompts, while another stated that they used a new session to modify the output of the previous prompt. Several students realized that their changes violated mandatory assessment criteria, including limits on the number of lines per file and restrictions on the use of functional elements, such as `Streams`. They used the same model to resolve these violations.

Answer to RQ2: *Most participants used LLMs for some purpose, regardless of their study group, with intentions varying widely. While most individual obfuscation steps are text-based and thus ineffective, nearly all participants also employed high-level structural changes.*

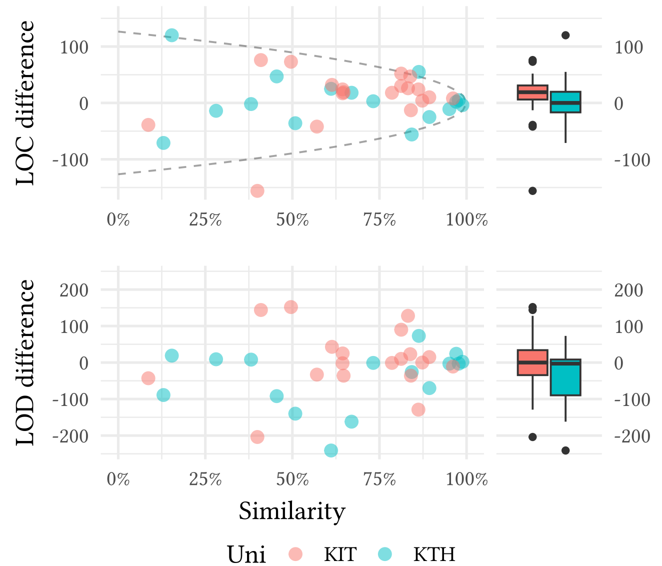


Figure 5: Difference in Lines of Code (LOC, top), and Lines of Documentation (LOD, bottom) of plagiarized submissions compared to the original submission.

4.4 RQ3: Changes

We measure changes in code metrics that do not directly relate to any plagiarism obfuscation strategies, specifically, but may still emerge as a result of them.

Number of Files. Figure 4 shows the distribution of the change in the number of files of the plagiarized programs compared to the original. Note that this does not measure the number of files changed, which may differ. Only six submissions altered the number of files across the whole range of similarity. Excessively partitioning functionality into new classes or inlining classes do not seem to be strategies that participants focused on.

Lines of Code. Figure 5 (top) shows the distribution of change in LOC of the plagiarized programs compared to the original. The range of differences expands as similarity decreases; however, not every plagiarized program with small similarity to its original version has a large difference in LOC. Two outliers have absolute differences exceeding 100.

Documentation. Figure 5 (bottom) shows the distribution of change in LOD of the plagiarized programs compared to the original. Note that this does not measure changed documentation, which may differ. Also, changes to documentation do not influence the structural similarity metric used by common plagiarism detection systems. Most submissions did not change the LOD significantly; for 22 submissions (62.8 %), the absolute change is less than 50. While the changes are evenly distributed around 0 for KIT (median: 0, mean: -7), participants from KTH tended to remove lines of documentation rather than adding them (median: -3, mean: -43).

Answer to RQ3: *Plagiarized submissions differ from the originals in lines of code and documentation without a consistent relationship to similarity scores, and rarely alter the number of files.*

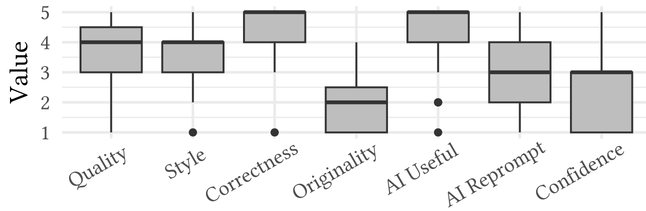


Figure 6: Results of the questions of the survey that were given on a 5-level Likert scale.

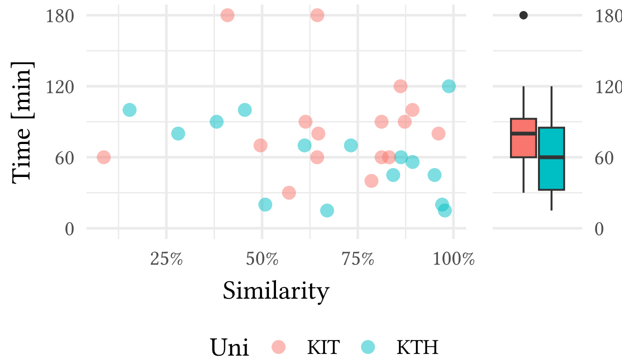


Figure 7: Participants' time spent working on the plagiarized program, related to its similarity to the original program.

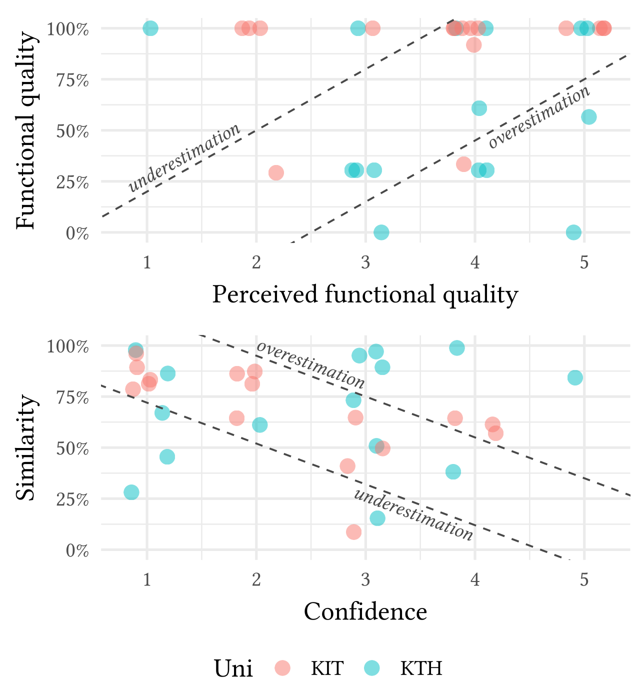


Figure 8: Accuracy of the participants' self-assessment, given on Likert scales (1–5): perceived functional quality vs. measured functional quality of their program (top); confidence in submitting their final program as their own, targeting their perceived risk of detection (bottom). Data points are jittered horizontally to make clusters of points more visible.

4.5 RQ4: Participants' subjective perception

In this section, we analyze the survey results. Figure 6 outlines the quantitative results of the questions which involved a Likert scale, while the time spent is detailed in Figure 7. We now discuss key results and connections we observed.

Time Investment. Figure 7 shows the correlation of time spent by the participants working on the plagiarized program to its similarity to the original program. Overall, participants spent between 15 and 180 minutes. For equal levels of similarity, the time invested varies greatly. Three participants effectively reduced similarity down to 67.0 to 50.8 % in 15 to 30 minutes. All submissions with a similarity below 50 % took at least 60 minutes.

Functional Quality. Figure 8 shows the distribution of participants' perception of their program's functional quality, related to its measured functional quality, given as the pass rate at functional tests. Assuming that accurate estimates are located on the diagonal (assessed quality of 5 corresponds to 100 % pass rate, 1 to 0 %), etc., the pass rates of six programs are more than 40 % lower than anticipated (below the line marked *overestimation*), but also, six programs were more than 40 % higher than anticipated (above the line marked *underestimation*). The remaining 19 assessments were fair in that regard (within a 22 % margin).

Confidence in Submitting Plagiarism. Figure 8 shows the distribution of participants' confidence in submitting their final program as their own, given the chance of detection, related to the program's similarity to its original. Assuming that accurate estimates are located on the negative diagonal (assessed confidence of 5 corresponds to 0 % similarity, 1 to 100 %), etc., the similarity of eight programs are more than 32 % higher than anticipated (above the line marked *overestimation*), but also, five programs' similarities to the original were more than 33 % below the anticipated value (below the line marked *underestimation*). The remaining 18 assessments were fair in that regard (within a 24 % margin).

Assumptions About Effective Strategies. In the change logs, several students included specific details about the intended effects of their changes, which provide insight into their understanding of plagiarism detection mechanisms. One participant changed string values to avoid detection by a system that would "check postcompile for similarities in available strings". Two participants pointed out that their changes would alter bytecode instructions of the resulting program; one intended to "change the control flow graph".

Usefulness of AI. According to the survey results displayed in Figure 6, participants largely agreed that the AI tools they selected were helpful for their purposes (median = 5), which included not only code transformation but also seeking advice on how to go about the task and how to fix certain bugs. Also, they usually did not have to iterate on their prompts very often (median = 3).

Answer to RQ4: *Most participants demonstrate a fair grasp of the similarity and functional quality of their program. Time investment and code metrics do not appear to be related to plagiarism success.*

5 Threats to Validity

As our study aims to capture authentic cheating and obfuscation behavior in a controlled setting, any operationalization necessarily involves trade-offs. We discuss the main threats to validity along common dimensions [35].

Internal Validity. We aimed to keep the setting comparable between KIT and KTH by using the same task, materials, and workflow; however, differences in infrastructure and unforeseen events may have affected participants' interaction with the submission process. In particular, technical difficulties at KTH required reconfiguration of the submission system during the session, delaying participants' ability to submit: While most students could still start working immediately, this disruption may have influenced how often they ran tests, how they used the change log, and how they allocated their time to obfuscation and debugging. Moreover, the task description initially included in the repositories at KTH was the German version meant for the run at KIT. Consequently, the English version was promptly shared with all participants at KTH via e-mail. Some participants at KIT did not bring a notebook and therefore worked in pairs or small groups. Collaboration may change obfuscation behavior (e.g., splitting work, discussing strategies) compared to individual work, thereby introducing variance not attributable to our controlled variables. In the survey, only two responses list multiple participants collaborating; neither team produced usable submissions. However, as collaboration can also occur in real-world plagiarism scenarios, we accepted this trade-off to avoid excluding participants without a device. Finally, one group of four collaborating students of KIT did not read the study task description sufficiently and consequently did not submit usable artifacts, either; excluding such cases may slightly bias the dataset toward participants who followed instructions more carefully.

External Validity. Our participants were first-term computer science students from two European universities, recruited voluntarily. While performing this experiment at two universities increases generalizability, the number of participants should still be increased in future replications to cover a broader spectrum of student populations. Also, as shown by the performance distributions in Table 1 and Table 2, attendants are biased toward higher-performing and potentially more motivated students; consequently, our results may overestimate both baseline programming ability and the sophistication of obfuscation strategies relative to the full cohorts. Moreover, the study simulates a realistic deadline-driven cheating scenario, but still differs from real-world plagiarism in some respects, such as being conducted on-site and with explicit instruction to plagiarize. Generalization beyond the studied task (*Dots and Boxes*), language (Java), and course context should therefore be done with care.

Construct Validity. Key study constructs such as *obfuscation success* and *strategy use* are approximated from the artifacts participants produced (code submissions, test outcomes, change logs, and survey responses). These proxies can be imperfect: participants may omit steps from the change log, interpret the instruction to make code "unrecognizable" differently, or provide socially desirable answers in the survey (e.g., regarding LLM use).

We mitigated misinterpretation through a pilot run and by supplying materials when issues occurred mid-session (e.g., promptly sharing the English task description at KTH). Still, residual measurement noise and site-specific effects are likely unavoidable.

Reliability. We provide extensive material at Zenodo [21], including study material, the resulting dataset, and evaluation data.

6 Discussion

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

Plagiarism Obfuscation Strategies. Among the identified individual steps to alter the code, the most prevalent strategies were changes to identifiers, documentation, and comments. These strategies change the code's outer appearance but are ineffective at obfuscating plagiarism. They also relied on structural changes to logic and statements, which are more effective but more challenging to implement on a large scale within a limited time while retaining functional quality. Across both study groups, *A* and *B*, participants frequently used LLMs with various intents and varying levels of specificity in their prompts. Often, LLM-generated code was faulty or violated mandatory checks; however, some submissions with LLM-based changes were among those with the lowest similarity.

Factors for Effective Plagiarism. Automatic plagiarism obfuscation, a threat commonly discussed in the literature, was not identified in the change logs; however, ChatGPT proposed such tools when one participant prompted for effective obfuscation strategies, listing several examples. While these tools were not used in our study, their availability suggests that automated obfuscation remains a realistic threat. At the same time, because such tools behave consistently across users, evaluation using synthetically generated plagiarism instances may be sufficient to assess defenses against this class of attacks. In our run of this study, the most effective form of plagiarism in our dataset was created using large-scale transformations that affected the code structure throughout. The authoring participant deliberately selected and documented these steps, indicating high knowledge and skill, which aligns with the long-standing assumption that successful plagiarism requires those.

LLM-based Obfuscation. We relied on participants' change logs to determine their use of LLM-based approaches; inspecting the changes resulting from specific prompts was out of scope. We suspect that general prompts for large-scale transformation, such as "change everything", currently still result in small changes with limited overall impact on the structure. We provide a dataset [21] reflecting students' use patterns of LLMs from two universities, whose effectiveness may improve as LLMs become increasingly capable and integrated into technical workflows.

Methodology. We present a foundational study design that yielded a dataset of 35 medium-sized, well-documented plagiarized programs. In future instances of this setup, tasks with different underlying conceptualizations may yield additional insights into how plagiarism strategies vary across assignments. Conducting the study at additional institutions, including universities outside Europe, may further reveal how educational context and prior training influence student behavior. Finally, since plagiarizing students are usually familiar with the task before adapting an obtained solution, participants may look into the task description at their own pace before the session, allowing them to start working more quickly.

7 Related Work

Feasibility of LLM-based cheating. Shallari and Hussain [36] conduct a case study on the feasibility of using LLMs to cheat in a hardware design course. They evaluate submissions for course assignments generated using ChatGPT and relate the results to the assignments' cognitive demand [40]. Their results suggest that ChatGPT is generally capable of procedural coding tasks, while rather conceptual, applied, and synthesis-oriented assignments that require deeper domain understanding often present a greater challenge. This suggests that instructors may control the feasibility of cheating using LLMs by adapting the assignment design to engineering tasks other than coding.

Biderman and Raff [2] show that GPT-J can generate solutions to introductory Java assignments that pass functional tests while avoiding detection by MOSS [1]. However, it remains unclear to what extent these results generalize to more advanced assignments.

Taylor et al. [38] expand on this question by conducting an empirical comparison of 209 implementations of the same task. The dataset includes 59 student programs and 150 programs generated using GPT-3: 75 based on a "regular" prompt, the other 75 including a request to design the program to evade plagiarism detection specifically. They conclude that deliberate prompt evasion was not effective in their setup; however, it adversely affected the functional quality of their programs. As LLM capabilities advance rapidly, future models may improve both the correctness and the ease of evasion, necessitating continual re-evaluation.

Impact of Prevalence of LLMs. Chen et al. [4] investigate whether the high popularity of ChatGPT with students after its public release influenced the plagiarism behavior in an introductory programming course. Comparing two consecutive runs of a large introductory Python course, they report increasing indicators of plagiarism and degrading performance in the final exam linked to overreliance on plagiarism during the exercise. To identify plagiarism, they rely on an empirically selected set of code features that they consider *markers* of plagiarism, such as usage of advanced language constructs or line comments. While these markers provide a practical proxy for plagiarism, their reliability was not validated; thus, some occurrences might be attributable to individual students' coding styles rather than plagiarism.

Kann [14] studies the potential effects of the public release of ChatGPT on students' attitude toward cheating. He reports no severe changes in attitude, but observed frequent confusion about rules governing LLM use.

Applications and Obfuscation of LLM Use. Lee et al. [19] present a survey-based study of how high school students use AI tools when completing programming assignments. They measure cheating tendencies using an adapted 12-item cheating scale [23]. They find that common uses include generating ideas and explaining new concepts [19]. The study's interpretability is limited by typical survey restrictions such as potential nonresponse bias and social desirability effects in self-reports. Also, student use patterns may have evolved since the publication, as have the capabilities of LLMs and their integration into devices and applications.

Dickey [8] studies plagiarism in a competitive programming course over seven semesters (2022–2025), with around 100 students participating each semester. They report that students used ChatGPT to obtain an approach and then manually reimplemented the solution, which proved an effective way to bypass automated plagiarism detection tools. To counteract students relying on LLMs for problem solving, the author proposes manual code reviews and student interviews in addition to automated code assessments.

Tran et al. [39] conduct an empirical study on students' attitude towards and purpose of LLM-based plagiarism. From 269 participating students across various study programs, 648 reports of prior LLM use were collected and classified. About 40 % of these cases were deemed improper behavior, ranging from failing to acknowledge using an LLM to "creating entire essays, reports, or assignments without mentioning AI use" [39, p. 13]. In individual interviews, some students outline how their transparency regarding LLM usage relates to lecturers' guidelines for specific assignments: For some homework about LLMs, it was required to list ChatGPT as a source, while other times, students' submissions might be rejected or have points deducted if they acknowledge using ChatGPT in their sources, thereby also damaging their reputation with the lecturer. Facing this risk, students opted to have LLMs generate reports answering the assignment questions based on course material, and adapting the report to their own writing style.

Summary. While the purpose and effectiveness of their use varies, LLM-based tools shape how students deal with assignments today [4, 19, 38]. Rephrasing LLM output in one's own words is a prevalent strategy to obfuscate LLM use and also often effective [8, 39], potentially requiring refocusing on new task formats [36].

This motivates us to continue our efforts, investigating open questions on students' approaches and attitude regarding LLM use.

8 Conclusion

The evaluation of plagiarism detectors is limited by the lack of datasets that reflect how students actually plagiarize programming assignments. We present an empirical study of how students plagiarize programming assignments under controlled conditions. Our results show that most participants did not produce plagiarized submissions that were both functionally correct and sufficiently dissimilar from the original, even when using LLMs. The change logs reveal that students primarily rely on local code transformations and LLM-assisted rewriting, while rarely modifying higher-level structures such as file organization. By publishing a dataset [21] with realistic solutions and clear ground truth, we provide a resource for evaluating plagiarism detectors against obfuscation strategies used by students. As future work, we plan to repeat the study at other universities and over time to expand the dataset and to study how student strategies and the use of LLMs evolve.

Acknowledgments

This work was supported by the Swedish Foundation for Strategic Research (SSF) and funded by the Deutsche Forschungsgemeinschaft (DFG) under the National Research Data Infrastructure – NFDI 52/1 – project number 501930651, NFDIxCS, and under SFB 1608 – 501798263. It was also supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF).

References

[1] Alex Aiken. 2022. *MOSS Software Plagiarism Detector Website*. Stanford University. http://theory.stanford.edu/~aiken/moss/ Accessed on Jan. 07, 2026.

[2] 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). *International Conference on Information and Knowledge Management*, 2933–2943. doi:10.1145/3511808.3557079

[3] 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

[4] Binglin Chen, Colleen M. Lewis, Matthew West, and Craig Zilles. 2024. 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)*. ACM, New York, NY, USA, 75–85. doi:10.1145/3657604.3662046

[5] 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

[6] 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 (ITiCSE 2023)*. ACM, 110–116. doi:10.1145/3587102.3588815

[7] 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

[8] Ethan Dickey. 2025. The Failure of Plagiarism Detection in Competitive Programming. doi:10.48550/arXiv.2505.08244 arXiv:2505.08244 [cs].

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

[10] 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

[11] Andreas Giannakoulas and Stelios Xinogalos. 2018. A pilot study on the effectiveness and acceptance of an educational game for teaching programming concepts to primary school students. *Education and Information Technologies* 23, 5 (Sept. 2018), 2029–2052. doi:10.1007/s10639-018-9702-x

[12] 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/

[13] Jurriaan Hage, Peter Rademaker, and Nikè Van Vugt. 2010. A comparison of plagiarism detection tools. *Utrecht University. Utrecht, The Netherlands* 28, 1 (2010).

[14] Viggo Kann. 2025. Students' Attitudes Towards Cheating Before and After ChatGPT. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Nijmegen, Netherlands) *(ITiCSE 2025)*. ACM, New York, NY, USA, 291–297. doi:10.1145/3724363.3729108

[15] 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

[16] Oscar Karnalim, Hapnes Toba, and Meliana Christianti Johan. 2024. Detecting AI assisted submissions in introductory programming via code anomaly. *Education and Information Technologies* 29, 13 (01 9 2024), 16841–16866. doi:10.1007/s10639-024-12520-6

[17] Cynthia Kustanto and Inggriani Liem. 2009. Automatic Source Code Plagiarism Detection. *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, 481–486. doi:10.1109/SNPD.2009.62

[18] 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*. IEEE, 98–105. doi:10.1109/LaTiCE.2013.37

[19] Victor R. Lee, Denise Pope, Sarah Miles, and Rosalía C. Zárate. 2024. Cheating in the age of generative AI: A high school survey study of cheating behaviors before and after the release of ChatGPT. *Computers and Education: Artificial Intelligence* 7 (Dec. 2024), 100253. doi:10.1016/j.caeai.2024.100253

[20] Robin Maisch, Timur Sağlam, Larissa Schmid, Sebastian Hahner, Nils Niehues, and Tobias Hey. 2026. Too Hot To Handle: The Effects of Temperature on AI-Generated Code Used to Cheat in Programming Assignments. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26)* (Rio de Janeiro, Brazil). ACM, New York, NY, USA. doi:10.1145/3774748.3795678

[21] Robin Maisch, Larissa Schmid, Richard Glassey, Dominik Fuchß, Nils Niehues, Haoyu Liu, and Anne Koziolek. 2026. *Replication Package for "Criminal Minds: How First-Year CS Students Plagiarize Code"*. doi:10.5281/ZENODO.19115558

[22] 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)* (Rio de Janeiro, Brazil). ACM, New York, NY, USA. doi:10.1145/3744916.3773225

[23] Donald McCabe. 2005. Cheating: Why students do it and how we can help them stop. In *Guiding students from cheating and plagiarism to honesty and integrity: Strategies for change*. Libraries Unlimited Westport, Connecticut, 237–246.

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

[25] 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) *(ITiCSE '19)*. ACM, New York, NY, USA, 555–561. doi:10.1145/3304221.3319789

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

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

[34] Timur Sağlam, Sebastian Hahner, Jan Willem Wittler, and Thomas Kühn. 2022. Token-based plagiarism detection for metamodels. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS '22)*. ACM, 138–141. doi:10.1145/3550356.3556508

[35] William R. Shadish, Thomas D. Cook, and Donald T. Campbell. 2002. *Experimental and quasi-experimental designs for generalized causal inference*. Houghton Mifflin, Boston, Massachusetts, USA.

[36] Irida Shallari and Mazhar Hussain. 2024. Assignments in the ChatGPT-era: Case Study on Plagiarism in Digital Systems Design Courses. *Hawaii International Conference on System Sciences 2024 (HICSS-57)* (Jan. 2024). https://aisel.aisnet.org/hicss-57/st/research_and_education/4

[37] Archer Simmons, Maristela Holanda, Christiana Chamon, and Dilma Da Silva. 2024. AI Generated Code Plagiarism Detection in Computer Science Courses: A Literature Mapping. In *2024 IEEE Frontiers in Education Conference (FIE)*. 1–7. doi:10.1109/FIE61694.2024.10893117

[38] Zachary Taylor, Cy Blair, Ethan Glenn, and Thomas Ryan Devine. 2023. Plagiarism in Entry-Level Computer Science Courses Using ChatGPT. In *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*. 1135–1139. doi:10.1109/CSCE60160.2023.00189

[39] Khue Van Tran, Thi Truc Mai Le, and Anh Tuan Pham. 2025. Exploring Vietnamese students' plagiarism awareness and actual practices through using of ChatGPT as a learning tool. *International Journal for Educational Integrity* 21, 1 (Nov. 2025), 32. doi:10.1007/s40979-025-00207-5

[40] Norman L Webb. 2002. Depth-of-knowledge levels for four content areas. *Language Arts* 28, March (2002), 1–9.

[41] Mimi Zhang. 2009. Using action-object pairs as a conceptual framework for transaction log analysis. In *Handbook of research on Web log analysis*. IGI Global, Chapter 21, 416–435.