Preventing Refactoring Attacks on Software Plagiarism Detection through Graph-Based Structural Normalization

Master's Thesis of

Robin Manuel Maisch

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr. Ralf H. Reussner
Second examiner:	Prof. Dr.-Ing. Anne Koziolek
First advisor:	Timur Sağlam M.Sc.
Second advisor:	Nils Niehues M.Sc.

20. November 2023 – 21. May 2024

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 21. May 2024

. . . .

(Robin Manuel Maisch)

Abstract

Detecting software plagiarisms among code by students remains a challenge. Plagiarists often obfuscate their work by modifying it just enough to avoid detection while preserving the code's runtime behavior in order to create an equally valid solution. This type of modification is commonly known as *refactoring*. State-of-the-art plagiarism detection tools use token-based comparison of submissions, which renders them immune against several types of refactoring obfuscation by their very design. Other types of refactorings, however, still create very effective plagiarisms. This thesis presents a novel approach that uses graph transformations as a means to normalize the structure of code submissions. This normalized structure is not affected by refactoring attacks. The normalization engine, implemented as a transformation system for code graphs, was integrated into a token-based plagiarism detection tool. We evaluate our approach on four relevant types of obfuscation attack schemes. From the results, we conclude that the approach is not only on-par with the state of the art in their efficacy against all attack schemes, but it even outperforms it by a large margin on combined refactoring attacks.

Zusammenfassung

Die Erkennung von Software-Plagiaten aus einer Menge von studentischen Code-Abgaben stellt nach wie vor eine Herausforderung dar. Solche Code-Plagiate werden häufig äußerlich verändert, um unentdeckt zu bleiben; jedoch soll sich ihr Laufzeitverhalten nicht verändern, damit das Plagiat eine ebenso gültige Lösung darstellt wie das Original. Genau solche Veränderungen werden als *Refaktorisierungen* bezeichnet. Aktuelle Werkzeuge zur Plagiatserkennung verwenden in der Regel tokenbasierten Vergleich zwischen Abgaben, und sind dadurch inhärent gegen einige Arten von Refaktorisierungsangriffen immun. Komplexere Refaktorisierungen können Plagiate hingegen erfolgreich vor Plagiatserkennern verbergen. Diese Arbeit stellt einen Ansatz vor, der Code-Abgaben in eine normalisierte Struktur überführt, auf der Refaktorisierungsangriffe unwirksam sind. Diese Normalisierung wird durch ein Transformationssystem für Code-Graphen umgesetzt, das wir in einen tokenbasierten Plagiatserkenner integriert haben. Wir werten den Ansatz auf vier relevanten Kategorien von Verschleierungsangriffen aus. Die Ergebnisse zeigen, dass dieser Ansatz nicht nur in allen betrachteten Kategorien mindestens auf dem Niveau des aktuellen Stands der Technik ist, sondern diesem in seiner Resilienz gegen kombinierte Refaktorisierungsangriffen sogar weit überlegen ist.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

Lecturers commonly use assignments as a means to enforce engagement with the subject matter of the course throughout the semester. While this is intended as an opportunity for the students to gain valuable insight and skills that will help them in their future careers [43], students often find assignments to be a considerable burden, given that they must develop and submit a solution prior to a deadline, which is then graded and may affect their final grade or their admission for the exam.

A number of factors may cause some students to be tempted to resort towards cheating [11] as a way to escape the ongoing pressure, such as the inability to keep up with the subject, failure to meet the time constraints, or fear that an inadequate submission will adversely impact their academic performance [6]. Depending on the type and context of the assignment, universities consider plagiarism to be a serious academic offense which can result in severe penalties. Still, plagiarism in student submissions remains a prevalent issue.

In computer science education, assignments often take the form of programming assignments where students submit a set of code files. As these code files are fully digital, they are particularly easy to share, modify, and resubmit by other students. For this reason, code plagiarism in computer science education has been subject to research for decades [43], which has produced a wide range of strategies and automatic tools to compare submissions and identify suspiciously similar pairs among them [28]. Currently, *token-based* plagiarism detection is the state of the art [21, 10, 19], which does not compute similarity scores from the source code of submissions, but from a list of tokens intended to “characterize the essence of a program’s structure” [33]. The abstraction introduced by the tokens makes plagiarism detection immune against a range of basic attacks, such as identifier renaming and the modification of comments.

Several recent contributions to the field of plagiarism detection aim to design a countermeasure against other specific types of obfuscation attacks, such as reordering of independent statements [38], or the insertion of dead statements [20]. This thesis presents an approach to make plagiarism detection resilient against *refactoring attacks*, where code is modified so as to change its structure—thus effectively obfuscating the plagiarism against token-based plagiarism detection—but its visible behavior is preserved, so that the refactored code should be an equally adequate solution as the original. As this definition is rather general, many attack schemes in the literature can be regarded as a refactoring attack, including all attack schemes mentioned so far.

The present approach to mitigating refactoring attacks revolves around transforming code into a normalized structure with regard to semantically equivalent alternatives. The concept of code normalization was used in other works to address the fact that with the certain degree of freedom which frequently arises when writing code comes opportunities to obfuscate plagiarisms. While the approach can cover a wide variety of attacks, each attack has to be addressed separately by a specialized transformation, or possibly a set of transformations, and due to the sheer number of possible attacks, we can never cover all of them. The evaluation was conducted in four stages, using automatically generated plagiarisms of one specific attack type each. These plagiarized programs were generated based off of real-world solutions to programming assignments, submitted by university students.

1.1. Contribution

This thesis provides three main contributions to the state of the art.

(C1) A *CPG transformation system* was designed and implemented as a framework to counter refactoring obfuscation attacks. This transformation system was subsequently extended by a graph linearization and tokenization component, so that it could be integrated into a token-based plagiarism detection tool.

(C2) An initial *selection of thirteen refactoring transformations* was collected and integrated in the CPG transformation system. These transformations, covering a wide variety of refactoring attack schemes, lay the foundation for the resiliency of token-based plagiarism detection against refactoring attacks.

(C3) The effectiveness of the CPG transformation system and its integrated selection of transformations as a means to normalize code and counter refactoring attacks was *systematically evaluated* in four stages, covering various attack schemes inside and outside the area of refactoring attacks.

1.2. Structure of the Thesis

The rest of this thesis is laid out as follows: In Chapter 2, key concepts related to token-based plagiarism detection, refactorings, and code graphs are introduced. Chapter 3 shows how the present approach is distinct from related work. Chapter 4 demonstrates how refactorings constitute effective attacks on token-based plagiarism detection, develops the basic idea of how such attacks can be mitigated, and defines what we can and cannot expect from this mechanism. Chapter 5 is a detailed description of the components of the CPG transformation system, which is the core contribution of this thesis. The selected transformations, the attacks that they target, and how they affect the submissions in terms of their code structure and the resulting token list, are illustrated in Chapter 6. Chapter 7 presents the design and the results of the evaluation, which assesses the resilience of the approach against four distinct obfuscation attack schemes. In Chapter 8, we discuss limitations of the present approach, pointing towards potential further research on the topic. Finally, Chapter 9 summarizes the content of this work, concluding the thesis.

2. Foundations

In this chapter, we introduce the key concepts for the thesis. We discuss the problem of software plagiarism in programming assignments, the state-of-the-art approach to detect such plagiarism as well as other interesting proposals, the use of graphs in code analysis, and how refactorings can be conceptualized as transformations on a code graph.

2.1. Code Plagiarism

Definition. In her PhD thesis, Cosma [6] observes that there is no universally accepted definition of what constitutes *software plagiarism*, which contributes to the fact that among educators, there is no consent as to whether a given instance of plagiarism is punishable, and which sanctions are adequate. She then suggests a first definition:

“Source-code plagiarism in programming assignments can occur when a student reuses source-code authored by someone else and, intentionally or unintentionally, fails to acknowledge it adequately, thus submitting it as his/her own work. [...]” [6, p. 66f.]

Cosma also names multiple aspects about the act of plagiarism that may be considered when determining the severity of an instance of plagiarism, e.g., the source, whether the source deliberately shared their solution with the plagiarist, and the software artifact subject to plagiarism (e.g., design, architecture, code, comments). Simon et al. [43] argue that a key factor to reaching academic integrity is that clear guidelines are communicated to the students indicating which sources of assistance and code are acceptable and which are not, and that the students must be encouraged to see programming assignments as an opportunity to reaching learning goals that will prove important for their future.

Much like code plagiarism, *code cloning* involves reusing code without acknowledging it. However, the context is different: while code plagiarism occurs in academia where students are required to submit code according to a specification or task description, code cloning occurs when existing code is copied and pasted as a basis for further development in the same project, generally resulting in redundant code which is considered bad practice [14, p. 72]. Still, code clones are prevalent in productive code. The study of Baker [1] on two pieces of software concluded that 19 and 20 percent of the code were redundant, respectively.

Code cloning is also subject to extensive amounts of research, yielding a multitude of tools for clone detection and elimination. Similar to software plagiarism detection, code clone detection revolves around finding large matches between parts of code, so the mechanisms of both are similar. Rattan, Bhatia, and Singh [34] review 72 tools for clone detection, classifying them into 24 comparison algorithms based on 13 intermediate representations.

Classification. Fiaidhi and Robinson [13] introduce a classification scheme of software plagiarisms which consists of seven levels, increasing in their potential to alter the underlying program structure. Karnalim [18] lists 50 types of plagiarism attacks of all seven levels (plus a newly introduced intermediate level), listed in Table 2.1.

Level	Changes	Examples
L0	Verbatim copy	
L1	Changes in comments and whitespace	
L2	L1 + Changes in identifiers	Rename variable, rename class
L2.5	L2 + Changes in packages and imports	Fully qualified class name ↔ import
L3	L2.5 + Changes in declarations	Declaration location, order, visibility, and assigned value; dummy variables
L4	L3 + Changes in program modules	Extracted methods; dummy methods
L5	L4 + Changes in program statements	Method calls; data types; usage of operators and control structures; order of operands
L6	L5 + Changes in decision logic	introduction of control structure; loop ↔ recursive method; loop boundary

Table 2.1.: Levels of plagiarism and characteristic examples [13], [18].

In the domain of code cloning, Davey et al. [9, p. 4] introduced a different classification system that originally featured four types, but has since been expanded to include other types based on artifact and granularity [34, p. 1167], all listed in Table 2.2.

Type	Type name	Typical changes/ relation between clones	Plagiarism Level
T1	Exact clones	White space, comments	L0-1
T2	Renamed/ parameterized clones	T1 + Identifiers, literals, types	L2-5
T3	Near miss clones	T2 + Statement insertion/deletion	L5
T4	Semantic clones	Functionally similar, but no textual similarity	>L6
	Structural clones	Common design basis	–
	Function clones	Clone on function-level granularity	–
	Model-based clones	Clone in (e.g., graphical) model artifacts	–

Table 2.2.: Types of code clones [34, p. 1167]. Types 1-3 are also referred to as the group of textual clones [36, p. 14]. The rightmost column gives an approximate mapping of the code clone types to the plagiarism levels.

2.2. Software Plagiarism Detection

In the history of research about plagiarism detection, *metric-based* approaches were prevalent for some time, e.g., [13], where as many as 24 metrics are used. Since the early 2000s, token-based plagiarism detection is the state of the art [21, 10, 19], although other approaches have been suggested, such as *graph-based plagiarism detection* [21]. This section describes the structure and principles of token-based plagiarism detection tools, like JPlag [33], Moss [41], and Dolos [22].

2.2.1. Token-based Plagiarism Detection Pipeline

Figure 2.1 gives an overview over the typical architecture of a token-based plagiarism detection tool. Each submission in the set of submissions (the *corpus*) may consist of one file or a directory containing any number of files of the same language, e.g., a programming language like Java, C/C++, or Python, a model description language like EMF, or natural language.

Parsing and Tokenization. Each file is first transformed to a list of *tokens* (see below). While there are no specified requirements concerning the inner workings of a language module, they typically lex and parse each file to an *abstract syntax tree* (*AST*), then traverse the AST and generate a specific token when entering and/or exiting a visited node. Each submission is mapped to a single token list, concatenating the tokens from the individual files.

Tokens. A token is a language-agnostic abstract representation of a language-specific syntactic element of the code, like a variable reference, a record declaration, or the beginning of an *else* block. Tokenization is intended to reduce the input down to its *essence*, i.e., its structure is preserved, whereas other aspects are discarded, e.g., formatting. Therefore, abstraction by tokenization renders similarity detection immune to *textual plagiarism obfuscation*.

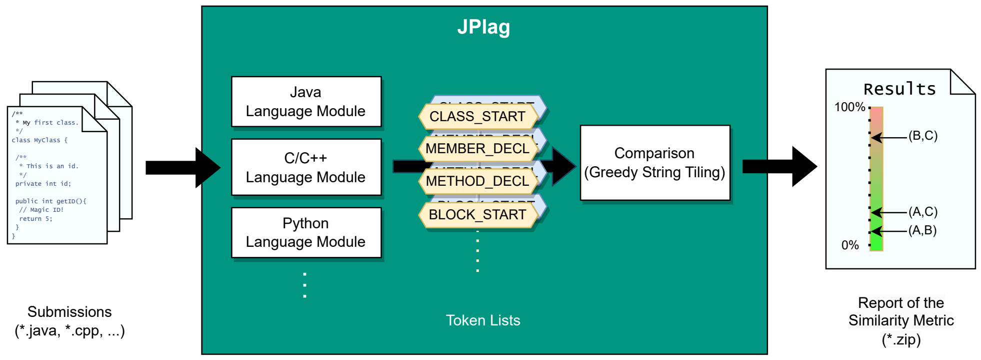


Figure 2.1.: The pipeline of a token-based plagiarism detection tool, using JPlag [33] as an example. Adapted from [20].

The comparison algorithm. The token lists generated from the corpus are fed to a comparison algorithm in pairs. The comparison algorithm used may differ between tools, but often a variation of Greedy String Tiling with Running Karp-Rabin Matching [47] is used, which is a runtime-optimized heuristic approach to determining a *maximum matching* of token subsequences between the pair. This means that, in total, the portion of both submissions covered by the matches is maximal. Wise suggests that matches of a length of 1 or 2 are insignificant for the purpose of matching programming languages [47, p. 3]. Thus, the minimal length of a significant match is a customizable parameter and serves as the central means to balance the precision/recall trade-off appropriate to the nature of the used language.

Output. For every pair of submissions, a range of similarity metrics are calculated from the found matches, usually given in percent. Finally, the results are made available to the user, e.g., in a text file, or via a UI in a web browser. Some plagiarism detection tools allow to examine pairs of submissions with special highlighting indicating matching sections, so that the user can easily assess whether highly similar submissions are suspicious.

2.3. Current Plagiarism Detection Tools

The next section presents software plagiarism tools to highlight the diversity of the various approaches.

2.3.1. Moss

Moss [41], short for *Measure Of Software Similarity*, is a closed-source "automatic system for determining the similarity of programs"[1] provided as an online service for free, where users can upload submissions via a script. Moss uses *winnowing* and *fingerprinting* on short subsequences of tokens which enables it to find matching subsequences in other submissions very efficiently. Users can access the results online for 14 days after submission.

Even though few changes seem to have been made to its core functionality since its original release in 1994, Moss is very popular to this day, and treated as the de-facto standard in the literature.

2.3.2. JPlag

JPlag [33] is an open-source[2] token-based software plagiarism detector that is commonly cited in the literature as "well known" and "widely used" [7, p. 2]. From a corpus of submissions, JPlag generates a set of files (the *report*) that, for each pair of submissions,

[1]Source: `https://theory.stanford.edu/~aiken/moss/`, visited last on 09.05.2024

[2]`https://jplag.com`, visited last on 07.05.2024.

contain information about matching parts between them, and a resulting similarity metric. The JPlag package comes with the report viewer, a browser-based UI which lists the submission pairs ordered by the similarity metric. For each pair of submissions, the UI can display code of both submissions alongside each other, highlighting matching parts in a manner similar to common diff visualizers, which aids the user during manual inspection of suspicious cases.

Since its release, JPlag has been described as a “commonly used” [21, p. 875], “promising” [29, p. 25] tool for software plagiarism detection in the literature. In their survey of source code similarity detection tools, Novak et al. [29, p. 14, p. 25] found that, among a set of five tools, JPlag was the most frequently referenced tool in the articles reviewed, and Moss and JPlag were most frequently compared to each other, indicating high relevance. At present, more innovation is put into JPlag than on any other similar tool.

2.3.3. Dolos

Maertens et al. [22] present Dolos, a token-based software plagiarism detection tool that offers support for a broad range of languages, and a modern, user-friendly graphical user interface. The ample language support is made possible through its use of the *tree-sitter parsing library*; parsers based on that library are available for more than twenty languages. Based on its use of winnowing and fingerprinting, the comparison algorithm is apparently closely related to Moss.

2.3.4. GPlag

Unlike token-based plagiarism detection tools, *graph-based* approaches to software plagiarism detection use a graph representation of the input code as the basis for comparison. As isomorphism detection on graphs and subgraphs is $\mathcal{NP}$-complete, measures must be taken for these approaches to scale with the size and length of the corpus.

Liu et al. [21] present GPlag, a tool that identifies similar submissions by detecting subgraph isomorphism between PDGs. To decrease the computation time, they propose two methods to identify pairs of PDGs that are unlikely to be isomorphic, based on metric properties of the PDGs. Park et al. [31] propose to eliminate dead code through backwards slicing, which reduces the size of the PDGs and thus optimizes the computation time of the subgraph isomorphism detector.

2.3.5. A-CFG

Chae et al. [4] introduce the *API-labeled control flow graph (A-CFG)* which indicates the frequency and order of API calls in a submission. From the A-CFG, a score vector is computed, which in turn is used to determine the relative similarity of pairs of submissions.

2.4. Plagiarism Generation Tools

Devore-McDonald and Berger [10] note that software plagiarism detectors which use string matching can be broken by insertion attacks. They present MOSSAD, an automatic plagiarism generation tool that uses random insertion of source code lines—from the source itself and an optional *entropy file*—to break up any matching token sequence of relevant length. After each iteration of insertion, the tool checks whether the result still compiles and whether the similarity score of the original and the result falls below a set target threshold, using the software plagiarism detector Moss as an oracle. The authors conclude that MOSSAD can successfully break current versions of Moss and JPLAG, which has inspired further research and development in this area, see Section 3.1.

Ko et al. [19] present COAT, a "Code ObfuscAtion Tool to evaluate the performance of code plagiarism detection tools". COAT applies a set of eight types of obfuscation attacks on code, roughly covering the range of Level 1 to Level 5 plagiarisms (see Table 2.1 in Section 2.1). Comparing Moss, JPLAG, and the token-based tools SIM and SHERLOCK which both use the Running Karp-Rabin Matching algorithm, they conclude that "overall, [...] JPLAG produced the best results." [19, p. 36]

As the ability of large language models (LLMs) to generate vast amounts of high-quality code in any programming language continues to improve, they provide a new opportunity to obtain a solution to a programming assignment from an external source. Niehues [25] created a sample of LLM-generated solutions by submitting the assignment instructions to ChatGPT, and another sample where ChatGPT was tasked to obfuscate a given solution. In the evaluation of his approach to improve the effectiveness of token-based plagiarism detection, he concluded that in most cases, LLMs failed to effectively obfuscate the input submissions, and that state-of-the-art token-based plagiarism detection finds high similarities in submissions fully generated by LLMs. His approach raises the minimum similarity between ten fully generated solutions to over forty percent, which is considerably higher than the average pair of original submissions of his data set. These results suggest that, if more than one fully LLM-generated solution is submitted, token-based plagiarism detectors might be able to detect them as if they were plagiarisms of each other.

2.5. Refactoring

Refactoring is the act of applying a series of incremental modifications to the internal structure of software, each preserving the observable behavior of the program [11, p. 45], but improving certain qualities about the code. This is an effective way to un-clutter code that has been extended many times over the course of the project's lifetime and may facilitate further extensions to the code [11, p. 50 ff.].

In the domain of refactoring, Fowler's book [14] is commonly named the standard reference. It encompasses a catalogue of more than seventy refactorings, with a case study detailing the effect of each refactoring and the steps required to apply it by hand.

```
logger.warn("Similarity of {} and {}: {}%", id1, id2, similarity*100);
```

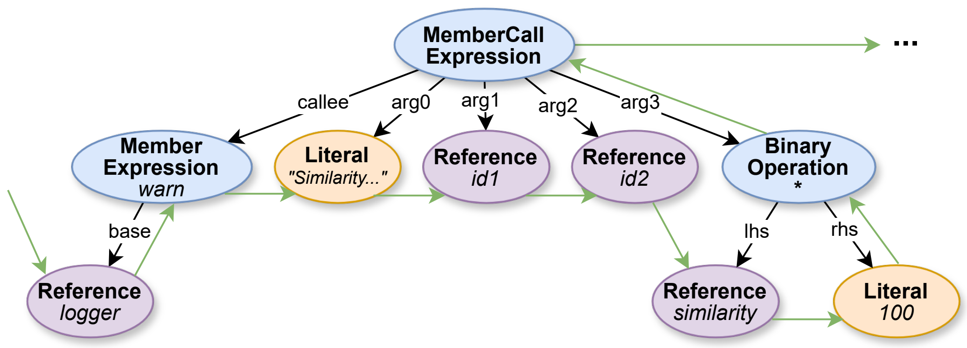


Figure 2.2.: A code snippet and the corresponding AST (black) and EOG (green).

2.6. Graph-Based Code Analysis

Due to the intricate interactions between elements of code, graphs are a commonly used representation as a basis for many types of static code analysis. In this section, a selection of types of tree and graph structures are briefly described.

Abstract Syntax Trees (ASTs) represent the structure of code as specified by the grammar of the used programming language. As the structure of the code is implicit to the AST, structural elements—such as parentheses around expressions determining the precedence of evaluation—are not represented explicitly. The AST is the result of lexing and parsing source code.

(Intraprocedural) Control Flow Graphs (CFGs) operate on individual method bodies. Their nodes represent blocks of code, and an edge between block A and block B indicates that there is an execution path where the code of B is executed immediately after the code of A.

Evaluation Order Graphs (EOGs) are an extension to CFGs from blocks to nodes: the nodes of methods are connected via EOG edges in the order of a hypothetical evaluation. EOGs are well suited as a basis for data flow analyses like variable liveliness analysis, variable definition and usage analysis, name analysis, type analysis etc. Given an AST node n, a EOG typically represents the traversion of n by post-fix depth-first order: It connects the predecessor of n to the child nodes of n in order, and finally reaches n itself.

Example 1: Standard statement. The EOG of the statement depicted in Figure 2.2 shows the typical postfix depth-first order of traversion.

Example 2: Control statement. In the case of control statements, the EOG order is adapted: the node of the control statement (`for`, `while`, `if`, `switch`, ...), is evaluated after the condition expression and serves as the branching node. The iteration statement of a `for` statement is evaluated after its body, and before its condition. This is illustrated in Figure 2.3. For branching nodes, the EOG edges are attributed with their respective Boolean value.

```
for (int index = 0; index <= 10; index++) sum += index;
```

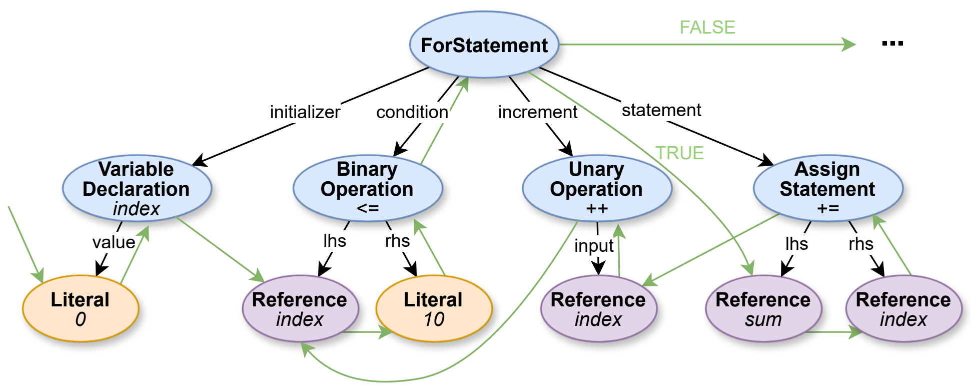


Figure 2.3.: A control statement and the corresponding AST (black) and EOG (green).

Another interesting type of syntactical element with an unusual EOG is shorthand logical operators, where the first operand also serves as a branching node. One EOG edge connects the left operand to the operation node (in case the left operand already determines the result of the operation), the other to the right operand (otherwise).

Dependency Graphs.

- *Control Dependency Graphs (CDGs)* connect control statements to their control-dependent statements, i.e., the control statement has multiple direct successors, and the control statement determines which successor is executed.
- *Data Dependency Graphs (DDGs)* connect variable definitions def_x and variable references ref_x where the value of the variable x at the evaluation time of ref_x may be the one set by def_x. The transitive DDG relation $n \xrightarrow[DDG]{}^* ref_x$ thus determines all nodes n that could impact the value of x at the time the ref_x is evaluated.
- *Program Dependency Graphs (PDGs)* [12] combine CDGs and DDGs. A typical use case of PDGs is *program slicing*. The *backwards slice* of a node `n` is the set of nodes that is reachable in a backwards search starting at `n`, and this set contains all nodes that may affect the computation of `n` [46]. Interprocedural PDGs, connected via *call edges*, are also referenced by their alternative name *System Dependency Graphs (SDGs)* [17].

Call Graphs connect method calls to the declaration of the called method. In object-oriented languages featuring dynamic binding, any number of methods may be the target in response to the method call at runtime. Compilers use call graphs for interprocedural analyses and optimizations.

Code Property Graphs (CPGs) combine properties of ASTs, CFGs and PDGs. Yamaguchi et al. [48] introduced CPGs as a tool for software vulnerability analysis, which remains one of its main applications to this day. Current open-source libraries offer automatic generation of CPGs for many programming languages, and even additional edges that cover data types, for example.

2.7. Refactorings as Code Graph Transformations

A formalization of refactorizations as graph transformations has been suggested by Mens et al. [24], which is outlined below. They used their formalism to define two refactoring transformations, *Encapsulate Variable* and *Pull Up Method*, and were able to prove preservation of behavior on both of these transformations. Note that the following summary is only for the sake of a better understanding of the theoretical concepts behind the approach described in this thesis; many of the notions and notations will not be used in the rest of this work.

A CPG is a *typed, labeled, directed graph* $G = (V_G, E_G, \ell_G)$ *over* Σ, Δ where

- V_G is a set of nodes,
- Σ is a set of node labels (indicating their *node type*),
- $\ell_G : V_G \longrightarrow \Sigma$ is a mapping between nodes and their label,
- Δ is a set of edge labels (indicating their *edge type*), and
- $E_G \subseteq (V_G \times \Delta \times V_G)$ is a set of labeled directed edges.

An *occurrence* of a graph S in a graph G (both over Σ, Δ) is an injective mapping

$$oc = (oc_V, oc_E), oc_V : V_S \longrightarrow V_G, oc_E : E_S \longrightarrow E_G$$

so that

- $\forall v_S \in V_S : \ell_G(oc_V(v_S)) = \ell_S(v_S)$
  (the node types of corresponding nodes are equal).
- $\forall e_S = (v, \delta, w) \in E_S : oc_E(e_S) = (oc_V(v), \delta, oc_V(w))$
  (the source node, type, and target node of corresponding edges are consistent).

A refactoring can be described as a *parameterized graph production* $\pi = (L, R, emb_{in}, emb_{out})$ where

- $L = (V_L, E_L)$ and $R = (V_R, E_R)$, the left-hand side and right-hand side of the production, are graphs over (Σ, Δ) and
- $emb_{in}, emb_{out} \subseteq (\Delta \times V_L) \times (\Delta \times V_R)$ are called *embeddings* (to be explained shortly).

The production π can then be applied to a graph $G = (V_G, E_G)$ iff there is an occurrence oc_L of L in G. Let $S = (V_S, E_S) := oc_L(L) \subseteq G$ be the image of L in G. The application of π to G results in a graph G' s.t. there is an occurrence oc_R of R in G', while $G \setminus S = G' \setminus S'$ remains unchanged. Let $S' = (V_{S'}, E_{S'}) := oc_R(R) \subseteq G'$ be the image of R in G'.

The embedding sets emb_{in} and emb_{out} specify the mapping of the *outer edges*, i.e., edges that connect the parts of the graph involved in the refactoring, S and S', to the unchanging parts of the graph, $G \setminus S = G' \setminus S'$. emb_{in} contains pairs $((\delta, v), (\delta', w))$ s.t. each *incoming* edge $(u, \delta, oc_L(v)) \in E_G$ must correspond to an *incoming* edge $(u, \delta', oc_R(w)) \in E_{G'}$. emb_{out} is constructed analogously for *outgoing* edges. See Figure 2.4 for an illustration. Notice that the type of the edge might change, e.g., when encapsulating a field, a variable reference is replaced by a method call to a getter method.

Remark. In model transformations, a generalization of graph transformations on a CPG, an *interface* or *glue graph* K serves the purpose of the embeddings, consisting of the "outer vertices" (those connected to the rest of the graph) of L and R. These vertices, together with their edges to the rest of the graph, are preserved by the transformation; thus, a glue graph does not provide the same level of flexibility as an embedding.

The *parameterized* nature of this formalization means that the graphs L and R do not necessarily represent one subgraph specifically, but rather *graph patterns* that a matching instance can essentially be plugged into.

The *typed* nature of the graph means that there is a set of tuples $\mathcal{T} = \{(\sigma_1, \delta, \sigma_2), ...\} \subseteq \Sigma \times \Delta \times \Sigma$, so that

$$\forall (u, \delta, v) \in E_G : (\ell_G(u), \delta, \ell_G(v)) \in \mathcal{T},$$

defining precisely which types of nodes a given edge type δ may connect. Note that δ may not be unique in those tuples. The authors use a *type graph* [24, p. 257] $\mathcal{TG}$ instead of a set $\mathcal{T}$ which incorporates nodes and edges of all allowed combinations, and they check for type-correctness by verifying that a given graph is isomorphic to $\mathcal{TG}$, which is not required for the purpose of this thesis.

The relation between G and G' as defined above is formally described like this [24, p. 261]: "*G directly derives G' using π via oc_L and oc_R.*" Alternatively, the informal shorthand $\pi(G) = G'$ could be used.

Based on these definitions, Mens et al. continue to define formally under which circumstances a CPG is considered *well-formed*, how to prove that a refactoring preserves well-formedness, and the precise mechanics of the application of a refactoring to a graph.

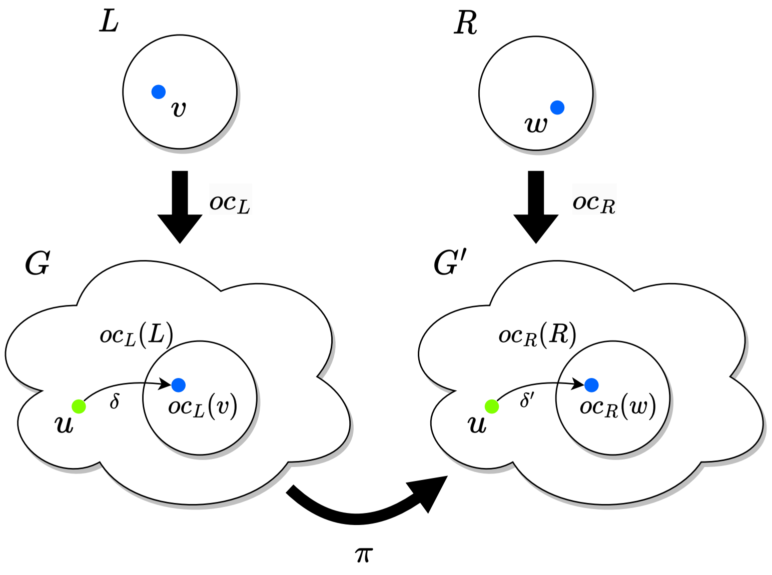


Figure 2.4.: Illustration of the relation of the original graph G, the left- and right-hand side L and R of a transformation π and their occurrences, the transformed graph G', and an embedded edge $((\delta, v), (\delta', w))$. Adapted from [24, p. 261].

3. Related work

In this section, we discuss recent contributions relevant to this thesis. First, we will take a look at extensions to token-based plagiarism detection that address new types of obfuscation attack schemes, input types, and intermediate representations. Then, we will cover a code transformation framework that aims to improve software quality.

3.1. Token-Based Plagiarism Detection

Several strategies to make token-based plagiarism resilient against a variety of attacks have been researched.

Resilience Against Specific Attacks. Krieg [20] describes approaches to defend against the introduction of dead variable declarations, unreachable code, and general insertion attacks like those used by MOSSAD. Sağlam et al. [38] use PDGs as an intermediate representation to detect dead code and counter *local confusion* [33, p. 1034] attacks, i.e., reordering of independent statements.

Niehues [25] presents a fully language-independent approach to combine adjacent token matches that are separated only by a few tokens, which is another effective way to counter insertion and reordering attacks, but also semantic-agnostic obfuscation attacks and AI-based obfuscation attacks.

Types of Submission Artifacts. Furthermore, recent contributions extend the area of plagiarism detection from code submissions to various other types of artifacts. Sağlam et al. introduce an approach for plagiarism detection on metamodels [40], and on tree-shaped models such as UML diagrams [37], with the latter approach claiming strong resilience against AI-generated obfuscation attacks. Strittmatter [44] describes an approach for plagiarism detection on state charts.

Token-Based Approach on Low-Level Instructions. Karnalim [18] presents an approach where tokens represent Java Bytecode instructions instead of structural elements of code. He argues that low-level instructions generalize over the syntactic sugar of a language, which makes a plagiarism detection tool immune against textual changes, like renaming, and changing comments or whitespace, but also against semantic-preserving replacement of instructions. For his evaluation, he collected a catalogue of over fifty plagiarism attack types and created an equal number of plagiarisms using only one attack type each. Comparing the effectiveness of the bytecode-based approach to two approaches based on source code, Karnalim concludes that the bytecode-based approach was the most effective.

A similar approach was evaluated by Heneka [16], who presents a language-independent approach to token-based plagiarism detection using LLVM IR as an intermediate representation. Heneka states that the language-specific approaches for Java and C++ were more effective at detecting plagiarism than the LLVM-based approach, concluding that the information lost in the process of the abstraction of low-level representations are indeed valuable for plagiarism detection. However, this abstraction step did prove to increase the resilience against obfuscation attacks.

Devore-McDonald and Berger [10] also argue that compiler optimization and comparison of assembly instructions are an effective countermeasure to dead code insertion attacks. They point out multiple drawbacks of this approach: Upon discovering a suspiciously similar pair of submissions, there is currently no technical aid that would map matching sequences of assembly instructions back to the source code, which makes it hard to manually trace these matches. Furthermore, this approach relies on the existence of a compiler that can produce a suitable representation of the submissions, which is unclear for programming languages that are typically compiled just-in-time.

Preprocessing. Novak [28] identifies various code snippets that occur commonly but contribute little functionality, thus diluting similarity measures in plagiarism detection. Examples of common code are getter and setter methods, as well as empty classes, functions, and blocks. Novak proposes to preprocess the code by removing common code, leaving more original code for comparison to generally decrease the similarity rating and the false positive rate. He also suggests other preprocessing techniques that are widely used in the literature, such as the removal of templates or base code supplied to students by the instructors, and the removal of comments from the code.

In a report by Simon et al. [42], the authors collect types of task-specific common code that *does* contribute functionality to the code, but no originality. Examples include the main method, input and output routines, base code that the instructors supply to the students, and code needed to initialize external libraries. Their study finds that the removal of these elements can increase the effectiveness of plagiarism detection.

3.2. Refactorings on Code Graphs

Cordy et al. [5] present TXL, a transformation system and DSL for defining and applying refactorings to source code. The transformations defined in TXL operate on a strictly syntactical level, so semantic information cannot be used in the preconditions. Thus, TXL would not have been a suitable transformation engine for this thesis.

Expanding further on TXL, Grant and Cordy [15] present Rust, an interactive interface that automatically detects and highlights parts of code suitable for refactoring. By selecting and applying refactorings, a range of typical code smells can incrementally be eliminated to increase the software quality.

4. Threat Model

A core benefit of token-based plagiarism detectors is that they are resilient against code modifications on a lexical level, i.e., a set of plagiarisms of a code file where identifiers, comments, strings, and formatting are changed still all result in the same token sequence. Even variable and field types can be exchanged for similar types if all accesses stay valid (e.g., replace a 32-bit integer with a 64-bit integer type). Thus, all these types of attacks pose no threat to the plagiarism detection.

By contrast, *structural* modifications to the code that do not change its observable behavior, which we will refer to as *refactorings*, are inherently bound to change the token sequence. The typical intention of such refactorings is to change a quality about the code, such as maintainability, but they are also a valid means to break token-based plagiarism detection.

Example. Any `for` loop can be refactored to a `while` loop (see 6.4.1), as demonstrated in the following schematic with a typical token representation:

```
for (<init>; <cond>; <iter>) {
    <body>;
}
```

```
FOR_STATEMENT;
VARIABLE_DECLARATION;
ASSIGNMENT;
ASSIGNMENT;
FOR_BLOCK_BEGIN;
...;
FOR_BLOCK_END;
```

```
<init>;
while (<cond>) {
    <body>;
    <iter>;
}
```

```
VARIABLE_DECLARATION;
ASSIGNMENT;
WHILE_STATEMENT;
WHILE_BLOCK_BEGIN;
 ...;
ASSIGNMENT;
WHILE_BLOCK_END;
```

The differences between the token sequences representing either control structure will likely break a matching token sequence, which is to say the refactored code version is unlikely to be recognized as a plagiarism of the original by a token-based plagiarism detection tool.

To counter a refactoring attack with a singular refactoring R, we could apply R to both the original and the plagiarism, so that the original becomes equal to the plagiarism (and the plagiarism remains the same); or we could try to restore the original by applying the inverse refactoring $\overline{R}$ of R to both the original and the plagiarism, so that the plagiarism becomes equal to the original (and the original remains unchanged). Either way, from two different versions we create two equal versions that a token-based plagiarism detection tool can easily identify as a potential case of plagiarism.

This thesis presents an approach that uses an extension of this idea to make a token-based plagiarism detection tool more resilient against typical refactoring attacks. In the real scenario, however, we are missing much of the knowledge that was available in the example above.

1. It is not known *which pairs of submissions* are originals and plagiarisms (unless current approaches already detect these plagiarisms).
2. It is not known *which kinds of refactorings* plagiarists might have used to create a plagiarism.
3. It is not known *which parts* of the original code might have been altered by a refactoring.

From 2., it is clear that we can never cover all types of refactorings. To get an approximation of the complete set, we need to define a set of refactorings that are likely to be used by our model adversary, and that are suitable for obfuscating a plagiarism from a token-based plagiarism detection tool.

From 3., it is clear that, given the set of all the code locations where a refactoring can possibly be applied, there is no valid method to select specific code locations to apply it to; instead, we need to apply the refactoring at all code locations possible. Thus, it is impossible in general to reconstruct the original code structure precisely. This, however, is not essential to the successful application of this approach; instead, we aim to create two equal versions from two different versions or—at least—two versions that are more similar than the submissions as they were submitted.

From 1., it is clear that it is essential that we must select the refactorings for our approach specifically with the risk of false positives in mind. While pairs of originals and plagiarisms (*OP-pairs*) shall become more similar to each other after refactoring them, pairs of unrelated originals (*OO-pairs*) should be made even *less* similar at best, but never more similar to an equal extent as the *OP*-pairs, or to a larger extent. To put it more briefly, the approach should make *OP*-pairs and *OO*-pairs more distinguishable by their similarity ratings than the current approach.

Now that we introduced the concept of a plagiarism generated by a refactoring attack, we can define a precise threat model. After that, the general idea of the defense mechanism is illustrated.

4.1. Threat Model Definition

The approach introduced in this work shall be evaluated by its effectiveness in defending against the following scenario:

- A plagiarist *P* obtains a complete, syntactically and semantically correct submission *S* created by a fellow student/participant *A*, the author, as a solution to a given programming task.
- *P* modifies *S* by, manually or especially automatically, applying refactorings to different portions of code.
  - We limit the refactorings to those that preserve the semantic of the code, so that the result of each application of a refactoring is again complete as well as syntactically and semantically correct.
  - The portion of code resulting from one application of a refactoring may again be a candidate for the same refactoring and/or other refactorings and may therefore be refactored multiple times.
- The result of all refactorings is a new submission *S'* that may be unrecognizable as a plagiarism of *S* when compared by a human or a state-of-the-art token-based plagiarism detection tool.
- *A* submits *S* and *P* submits *S'* for assessment and grading.

Refactorings considered in the threat model include:

- The semantic-preserving replacement of control structures,
- The extraction of variables (including class constants),
- The movement of class constants to other classes,
- The usage of API calls to replace implemented functionality,
- The insertion of dead code,
- The addition of *common code*, be it used or unused, and
- The semantic-preserving reordering of code elements.

On the other hand, we do *not* focus on plagiarism attacks that involve the use of large language models (LLMs). Also, we do not consider how the use of a refactoring on code will affect any of its qualities.

4.2. Distinction Against Related Work

The approach presented in this paper is distinct from existing work in the combination of three aspects.

Threat model. To the author's knowledge, strategies to counter refactoring attacks have not been subject to research yet. Related works have dealt with more specific attack schemes, like manipulation on the textual level (Moss[41], JPlag without normalization [33], Dolos [22]), reordering [38] or dead and unreachable code insertion [20]. Other works presented approaches that were immune to specific refactoring attacks because of the intermediary representation of the code but are vulnerable against refactoring attacks in general [21].

Usage of Code Graphs. In contrast to GPlag [21], this approach uses a code graph only as an intermediary representation of the code, but then tokenizes the code graph to a token list, which is subsequently used for comparison. Even though Liu et al. argue that isomorphism algorithms are much more efficient on PDGs than on general graphs, it is the quadratic number of comparisons in terms of the number of submissions which is bound to lead to a long runtime in real use cases. A-CFG [4], on the other hand, uses a CFG as the intermediary representation and metrics for comparison, which are not derived from the general structure of the submissions, but restricts itself to the order and number of API calls. The present approach aims to combine the power that comes with the rich information content of the CPG with the efficiency of runtime-optimized token-based comparison. Furthermore, it will run isomorphism detection only linearly often in the number of transformations and the size of the submissions.

Transformation of the Submissions. Other approaches apply removal of dead code or effectively discard dead code before comparison (GPlag [21], JPlag with token sequence normalization [20, 38]), but otherwise leave the general structure intact. In contrast, the present approach may alter, move, and/or remove major parts of the structure, while preserving their *semantic* content; the scale of these changes depends on the selection of transformations.

5. Graph-Based Structural Normalization

In Chapter 4, we established that a wide variety of refactorings can be used as a tool to create a plagiarism that current state-of-the-art plagiarism detectors may not be able to detect. The following chapter presents the first main contribution of this thesis: we describe the structure and workflow of a CPG transformation system as a framework to make token-based plagiarism detection resilient against refactoring attacks.

5.1. The Defense Mechanism – Overview

As discussed in the foundations (see Section 2.2.1), a token-based plagiarism detector transforms the code submissions into token lists, and then compares the token lists using a comparison algorithm. The tokens for a submission are created by traversing an intermediate representation of the code, typically the AST, and outputting tokens for a selected set of code elements that are deemed structurally relevant.

The approach presented in this thesis uses a *code property graph* (CPG), an AST extended by more edges which indicate various relations between nodes. Before the token list is created, the rich data of the CPG is used to detect specific structures in the code which are subsequently rearranged to perform refactoring transformations. These transformations intend to reverse refactorings that an attacker might have applied to the code by creating a *normalized representation* of the code.

Example. See Figure 5.1 for an example on how a refactoring may alter the token list. A new variable is extracted from a statement, and a variable reference is inserted at the original position of the extracted subexpression. This modification of the code also affects its token representation: the tokens of the subexpression move in front of the tokens of the statement in the token list, and a new token for the new variable declaration is added. Through these changes, a matching token sequence will likely be broken. As

```
return balance * power(1 + getRate(), time);     RETURN APPLY APPLY

double factor = 1 + getRate();                   VARDEF APPLY
return balance * power(factor, time);            RETURN APPLY
```

Figure 5.1.: The refactoring *Extract Variable* is effective in manipulating the token list, and thus can be used as an obfuscation attack.

extracted variables are often used one time only, as is the case in the example, a sensible normalization of the code could eliminate all single-use variables by inlining them if possible; the resulting normalized code would then be invariant under variable extraction attacks.

When integrated into a plagiarism detection tool, a CPG transformation system should incorporate a selection of transformations that strikes the balance between the effective action against refactoring attacks and the preservation of the semantic core of the code to limit the risk of false positives.

5.2. Workflow of a CPG Transformation System

Figure 5.2 shows the flowchart of a CPG transformation system. This section summarizes the entire process briefly, and the following sections cover each step in more detail.

The input to the CPG transformation system is a set of transformations, represented as pairs of a source and a target graph pattern each, and the code submission files.

(a) In a preparation step, the source and target graph patterns are compared to determine the individual operations required to build the target graph from the source graph.
```
transformationOperations = compare(sourcePattern, targetPattern)
```

(b) For each submission, a collective CPG is constructed from all submission files.
```
cpgRoot = createCpg(files)
```

(c) The structure of the CPG and the properties of the individual CPG nodes are compared to the source graph pattern of each transformation, yielding a list of matches.
```
matches = findMatches(sourcePattern, cpgRoot)
```

(d) From each match of a transformation, the transformation operations are instantiated with the concrete nodes of the match.
```
for (match in matches)
     concreteOperations = instantiate(transformationOperations, match)
```

(e) The transformation instance is applied to the subgraph involved in the match.
```
for (operation in concreteOperations) operation.apply()
```

(f) Finally, a linear order is determined on the CPG nodes, in which the nodes are then tokenized. The token list is the input to the subsequent comparison algorithm.
```
nodeList = linearize(cpgRoot)
tokens = tokenize(nodeList)
```

Next, we have a look at graph patterns for refactoring transformations, and which kinds of properties are necessary to check to ensure their validity.

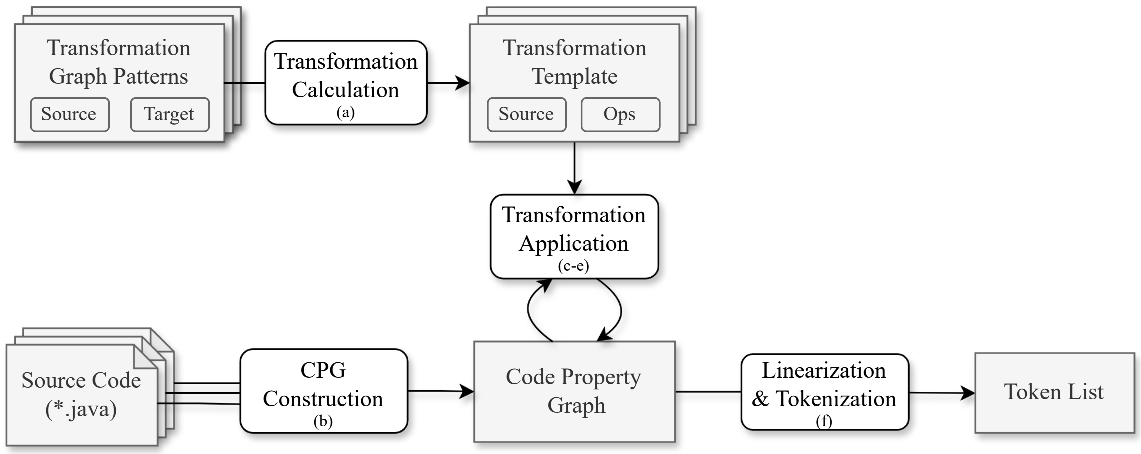


Figure 5.2.: A flowchart of the CPG transformation system, the core of the present approach.

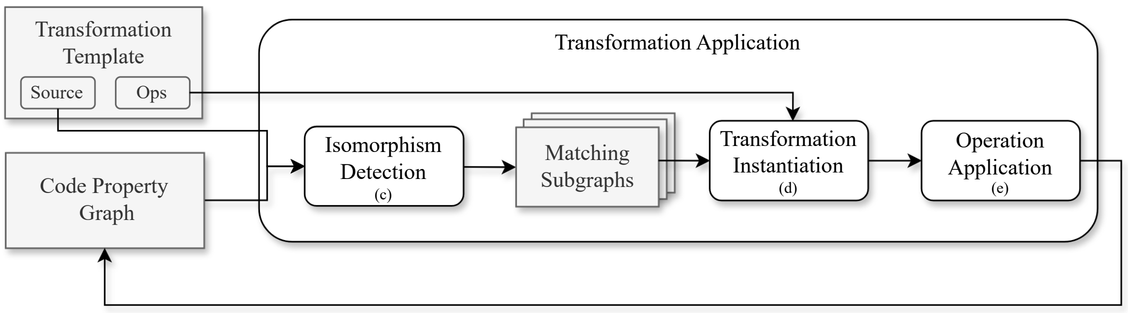


Figure 5.3.: A flowchart of the transformation application component.

5.3. Representation of Transformations

Following the example of Mens et al. [24], CPG transformations will be represented as a pair of *graph patterns* which represent the structure of a subgraph required to apply the transformation (the source graph pattern) and the structure that represents the same subgraph after the transformation (the target graph pattern). The source and target graph patterns serve two main purposes: The difference between the two graph patterns determines the *transformation operations* that form the transformation, and the source graph pattern is the template in the search for *matching subgraphs* of a concrete CPG to which the transformation shall be applied.

5.3.1. Overview

Just like the CPG is a set of nodes connected by directed edges, a graph pattern is a set of *node patterns* connected by asymmetric *relations*. A node pattern represents one CPG node and contains all the specifications needed to determine whether a given concrete node *matches* the node pattern. Before we examine the different elements more closely, we consider an example.

Figure 5.4 shows a source graph pattern and a target graph pattern representing the transformation *Move constant to only using class*. This transformation is supposed to revert the formation of *constant classes*, where a set of constants is moved from their respective original classes to a new class that contains no methods, only constants.

In text form, this is the list of requirements for a match:

- In a class *definingClass*, a constant must be defined, i.e., a static final field.
- The constant must be used at least once.

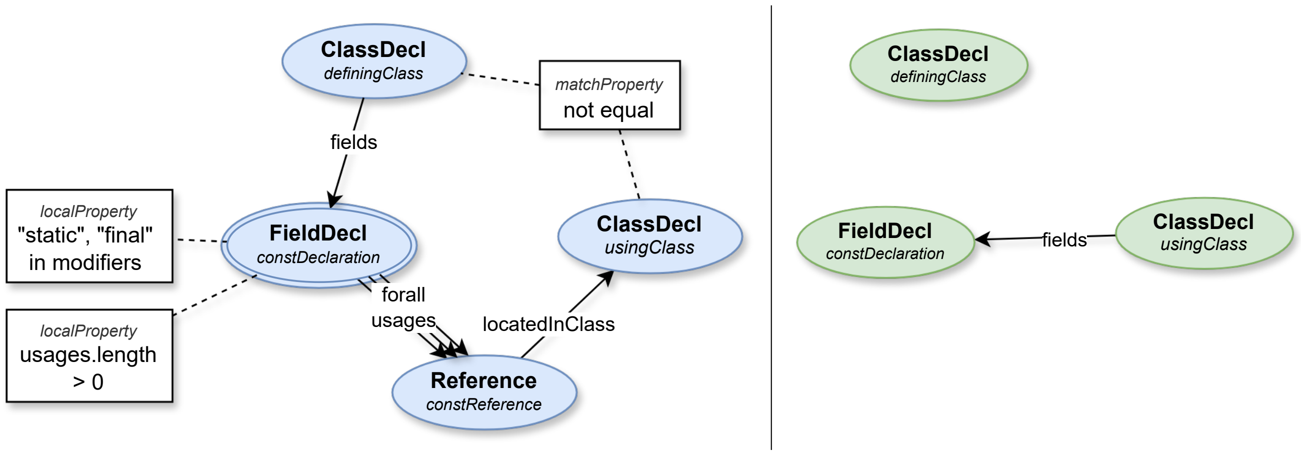


Figure 5.4.: The transformation *Move constant to only using class*, represented by a source and target graph pattern.

- The references to this constant must all be located in the same class *usingClass*.
- The *usingClass* must not be equal to the *definingClass*.

If all these requirements apply, the transformation moves the constant declaration from the *definingClass* to the *usingClass*, according to the target graph pattern.

Note that the relation of a class declaration to its field declarations is already present in an AST; however, to relate a declaration to its non-local references, the extensive analyses used in the construction of the CPG are necessary.

Note also that the only purpose of the *constReference* node pattern is for the description of a matching property; it is not involved in the transformation. For this reason, *constDeclaration* is marked as a *terminal node* for this transformation (illustrated by the double oval) indicating that all its related nodes should be preserved as they are. The transformation calculation will thus not recursively step into the related nodes of *constDeclaration*.

In the following subsections, the various categories of node pattern properties are described.

5.3.2. Local Node Pattern Properties

The most basic properties of a node pattern are *local*, i.e., the property can be evaluated with only the candidate node at hand. These are:

- the node type, e.g., a field declaration, a method parameter, a return statement.
- properties concerning attributes of the node, such as the identifier of a variable declaration, the value of a literal, or the modifiers of a method declaration.
- properties concerning the number of 1-to-n-related nodes (see below), e.g., a setter method should have exactly one parameter and one statement (an assignment); an unused variable should have exactly zero references to it.

5.3.3. Node Pattern Relations

In a CPG, there are various types of directed edges between nodes which express different types of relations, such as a parent-to-child relation in an AST, or a dependency relation of a DDG. We will use the term *relation* to generalize over CPG edges and any other kind of computable relation between nodes. Note that a relation type defines the type of nodes that it connects, e.g., the edge type *If condition* connects `if` statements to expressions.

We divide relations by their cardinality as follows:

- *simple* relations, or *1-to-1* relations, which relate at most one node to a reference node, e.g., the body of a method, the condition of an `if` statement, or the surrounding class definition of a statement.
  A simple relation may also be understood as a partial function r : `Node` $\rightharpoonup$ `Node`.

- *multi* relations, or *1-to-n* relations, which may relate any number of nodes to a reference node, e.g., the parameters of a method, the members of a class, the references to a variable.
  A multi relation may also be understood as a set-valued function $r : \texttt{Node} \longrightarrow \mathcal{P}(\texttt{Node})$.

From these relations, we can construct *relation properties* which are tuples (r, p) of a relation r and a node pattern p. For a *candidate node* n to satisfy the relation property (r, p), n must be related to another node n' with the relation r so that n matches p. We differentiate three types of relation properties:

- *simple* relation properties, or *1-to-1* relation properties
  $$n \vdash (r, p) :\Leftrightarrow r(n) \neq \texttt{null} \wedge r(n) \vdash p$$
  Example: The relation *locatedInClass* of the node pattern *constReference* to the node pattern *usingClass* in Fig. 5.4 is a simple relation property.
- *existential* relation properties, or *1-of-n* relation properties, where of all 1-to-n-related nodes, one matching node is sufficient to satisfy the matching property
  $$n \vdash (r, p) :\Leftrightarrow \exists n' \in r(n) : r(n) \vdash p$$
  Example: The relation *fields* of the node pattern *definingClass* to the node pattern *constDeclaration* in Fig. 5.4 is a 1-of-n relation property.
- *universal* relation properties, or *n-of-n* relation properties, where of all 1-to-n-related nodes, one mismatching node is sufficient to break the matching property
  $$n \vdash (r, p) :\Leftrightarrow \forall n' \in r(n) : r(n) \vdash p$$
  Example: The relation *usages* of the node pattern *constDeclaration* to the node pattern *constReference* in Fig. 5.4 is a n-of-n relation property.

Note that for simple relation properties and n-of-n relation properties, one iteration of the matching algorithm may only ever find one match, whereas a 1-of-n relation property may yield as many different matches as there are related nodes.

5.3.4. Match Properties

Match properties extend over more than one relation and may include any number of nodes from the match. Typical instances include cyclic graph patterns, i.e., cases where multiple different chains of relations must (or must not) lead to the same node. In Figure 5.4, for example, we require that the *definingClass* and the *usingClass* be unequal.

5.3.5. Roots of Code Property Graphs

Although CPGs inherently violate the properties of a tree, they still contain the AST structure of the represented code. When traversing or comparing CPGs, in many cases it makes sense to traverse the underlying AST and start at its root, which we will therefore also declare the *root node* of a CPG or any of its subgraphs. Analogously, graph patterns have a dedicated *root node pattern* that should be selected so that all relevant node patterns can be reached via transitive relations. In practice, it turned out that it is very convenient to allow graph patterns to have multiple root node patterns to restrict the graph pattern to the most relevant parts. See Section 6.3.1 for an example.

5.3.6. Wildcard Parent Nodes

In many transformations where a node shall be removed or replaced, that node can occur in many different contexts in the AST. For example, an expression node can occur as a method call argument, as the initialization value for a declaration, or as an operand to an operation. To gain flexibility regarding the context of a root node, we introduce the concept of the *wildcard parent node pattern* to graph patterns, serving as a placeholder pattern for the AST parent of a node. Unlike regular node patterns, wildcard parent node patterns do not specify their node type, nor the kind of edge that connects it to the proper root pattern (the *wildcard edge*). Again, Section 6.3.1 serves as an example.

5.4. Transformation Calculation

Given a source graph pattern and a sequence of transformation steps, one can construct the target graph by applying the transformation steps to the source graph pattern. On the other hand, to extract the transformation steps from a pair of source and target graph patterns is less trivial and requires an idea of what kinds of manipulations graph transformations should be able to perform.

First, to identify the pairs of node patterns that should be compared, we need to relate node patterns of the source and target graph pattern to each other. To that end, we assign *roles* to node patterns for a specific graph transformation. If a source node pattern and a target node pattern share the same role, they shall represent the same node of the CPG in every instance of the transformation. We call node patterns with the same role in their respective graph patterns *equivalent*.

I present a set of five types of transformation steps, called *operations*, which cover a wide variety of graph transformations for our scenario. Assuming that a node pattern m of the source graph S is compared to the pattern n of the target graph T, related to their parent node pattern p by the relation r, then

- If an equivalent of n does not exist in S, *create* an equivalent node of n and add it to the node set of S.
- If the roles of m and n are different, *replace* m by the equivalent node of n in S.
- If n is `null`, *remove* the edge from p to m in S.

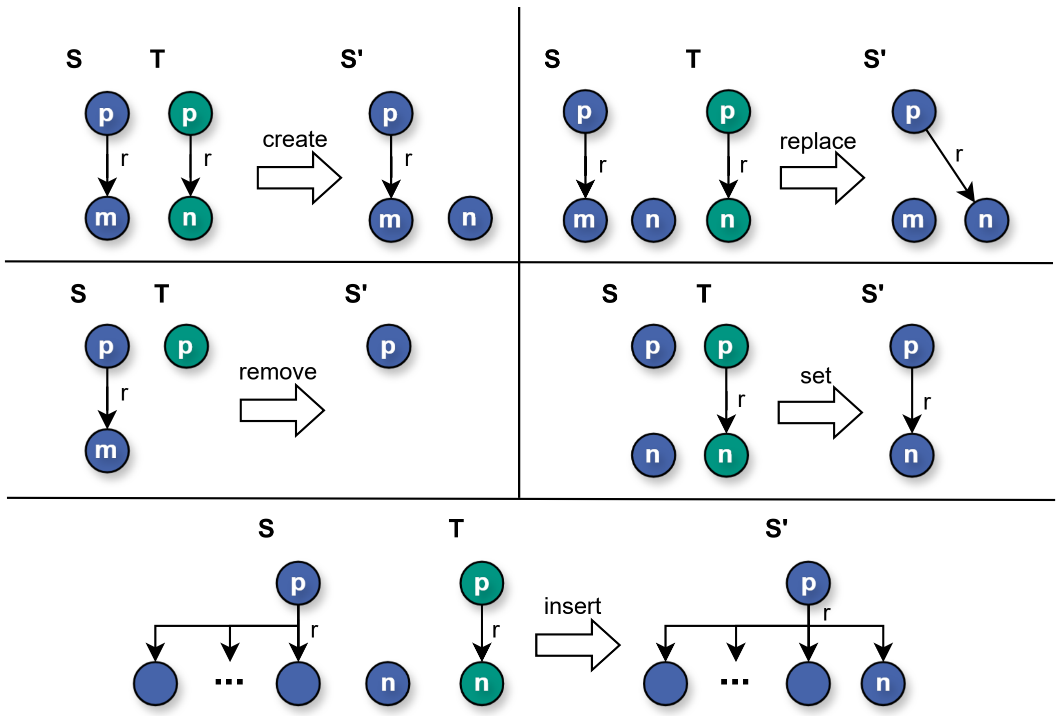


Figure 5.5.: Scheme of the five transformation operations.

- If *m* is `null` and *r* is a 1-to-1 relation, *set* the equivalent of *n* in *S* as the target of the relation of *p*.
- If *m* is `null` and *r* is a 1-to-n relation, *insert* the equivalent of *n* in *S* as a child of *p*.

The operations *set* and *insert* are different from each other in that insertion involves an index.

5.5. Pattern Matching Algorithm

The pattern matching algorithm, which we will also call the isomorphism detection algorithm, compares node patterns to concrete nodes, and creates a mapping of node patterns to nodes, called the *match*. The current node that is being compared to the node pattern will be referred to as the *candidate node*.

Just as in the transformation calculation (Section 5.4), the entry point of the graph pattern for the comparison is the (first) root. To avoid comparisons that can clearly not produce a match, we only initialize the comparison with nodes of the correct type, e.g., if the root node pattern is a `while` statement node pattern, then it is of no use to compare any nodes other than `while` statements to it.

For each pair of node pattern and candidate node encountered, the local properties are checked first. If the candidate node is not of the type specified by the node pattern, then the relation properties cannot exist, e.g., an assignment statement does not have any *fields* relations as required from the *definingClass* node pattern in Figure 5.4. If the local properties are satisfied, then the node pattern will be mapped to the candidate node in all current matches. Otherwise, the comparison for the current candidate node terminates.

Secondly, the relation properties are checked as described in Section 5.3.3, and for each relation property, the related node pattern is recursively compared to the related node of the candidate node. If a match already contains a mapping from a node pattern to a node `n`, and the current candidate node is not `n`, then it is a mismatch. If the mapped node and candidate node *are* equal, then the algorithm can step out of that comparison to avoid infinite loops.

Lastly, the match properties are checked against the current matches. It is only at this point, after we have encountered and matched all involved node patterns, that match properties *can* be evaluated. See Algorithm 2 in Appendix A.1 for a pseudo code listing of the isomorphism detection algorithm.

5.6. Node Management

Regarding the runtime of the CPG transformation system, traversing the CPG (or parts of it) repeatedly turned out to be especially costly. In most cases, this could be remedied by collecting and storing the relevant data in a single traversal, and subsequently extracting the needed data from this data instead of traversing the CPG again. This section describes three cases where in the design of various analyses or transformations, a large number of specific nodes needed to be accessed randomly, and how this requirement was dealt with, also considering the resulting computational efficiency.

5.6.1. Store Root Candidate Nodes by Class

In the process of isomorphism detection, we need to quickly find root candidate nodes *of a specific node type* for each transformation and each submission. A suitable data structure for this is a *tree map*, which stores key-value elements in a binary tree by a given order `<` on the keys. The structural invariant of the tree map is that for every element n, its left child element l, and its right child element r, $l \leq n \leq r$ holds. The tree map also supports operations to get sublists of elements in a range between two given keys.

For our purposes, we use node class objects as keys, and lists of nodes of the corresponding node types as values. To use a tree map on classes, we need to define a suitable order `<` on node classes.

We define the following order `<`, which uses lexicographic and hierarchic relations: Given a pair of node types `A` and `B`,

- If `B` is a (direct or transitive) subclass of `A`, then `A < B`.

- If `A` and `B` share the same immediate superclass, then order `A` and `B` lexicographically.
- Otherwise, there must exist a common transitive superclass `N` of `A` and `B`. Find the immediate subclasses `SupA`, `SupB` of `N` that are different superclasses of `A` and `B`, respectively, and order `A` and `B` according to the lexicographic order of `SupA` and `SupB`.

A sublist of node types ordered in this manner may look like this example, where the suffixes indicate the inheritance relations:

```
Node, Declaration, ClassDeclaration, FieldDeclaration,
      Expression, BinaryOperatorExpression, UnaryOperatorExpression,
      Statement, AssignStatement, IfStatement, WhileStatement.
```

In our tree map of node lists, the node lists will we stored in the order of their type. Given that tree maps are binary, a child node type will not be a subtype of its parent node type in general. In Figure 5.6, a tree map of integers shows how the order of the keys relates to the structure of the tree.

Now, we want to get all suitable root candidate nodes for comparison with a node pattern of a type `T` from the tree map. Those include not only the nodes of type `T` itself (which may be an abstract class), but all subtypes of `T`. To find and concatenate all these lists of nodes, we need to find the position of the smallest node type compatible with `T`, which is `T` itself, the position of the greatest subtype `T'` of `T`, and the lowest common ancestor `A` of `T` and `T'`; then, we collect the nodes of

- `T` and its right subtree,
- walking up along the edges towards `A` from `T`, all parent nodes that are reached from their left edge and their right subtrees,
- the lowest common ancestor node `A` of `T` and `T'`,
- walking down along the edges towards `T'` from `A`, all child nodes that are exited via their right edge and their left subtrees,
- `T'` and its left subtree.

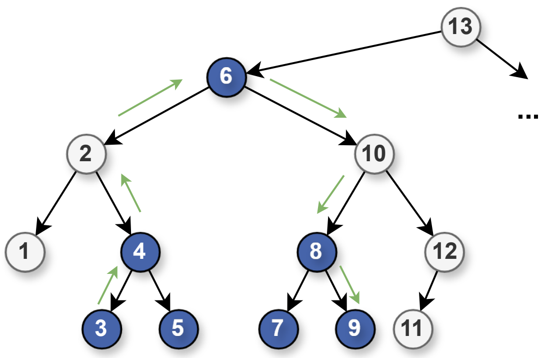


Figure 5.6.: A tree map with integer keys. The dark nodes represent the range between 3 and 9. The green arrows represent the traversal of the tree to find all nodes in that range.

This traversal is also depicted in Figure 5.6, where the equivalent of `T` is 3, `T'` is 9, and their lowest common ancestor `A` is 6.

After each transformation, we need to update this structure; more specifically, if a node was removed from the CPG, it must be deleted from the tree map, so that it can no longer be considered as a root node candidate.

5.6.2. Store AST Parents of Potential Root Nodes

In the isomorphism detection step, where the root node pattern shall be compared to suitable candidate nodes, we need to handle the case where the root node pattern is a wildcard parent pattern—these do not specify a node type, and so, virtually any node of the CPG may be considered a suitable candidate.

We solve this by storing a mapping of each CPG node to its AST parent (the *parent map*) and starting the isomorphism detection at the child of the wildcard parent pattern, the *proper root*, instead of the wildcard parent pattern itself. The wildcard parent and the wildcard edge can then be determined in constant time by a lookup in the parent map, completing the mapping of node patterns to nodes in the match.

After applying a transformation, the potential changes need to be reflected in the parent map. These updates can easily be integrated into each transformation operation.

5.6.3. Relative Order of Nodes

The DFG analysis step detects pairs of nodes that must remain in order relative to each other during linearization, which is represented in the CPG by additional DFG edges. Whenever such a pair is determined, the direction of the relation is not immediately clear, especially considering that these relations can occur across the boundary of a loop. To determine the relative order of the nodes, we can reuse the parent map with a small extension: The depth of each node is added to the parent map entries. Algorithm 1 shows a pseudo code listing for the algorithm.

The calculation of the EOG distance (line 15) involves traversing the EOG, which is exactly what we are trying to circumvent when trying to reduce the computational effort. However, most cases can be covered with the linear-time case in line 13, and in the remaining cases, at least one node is usually rather close to the parent node, e.g., in the condition of a loop statement, or in an argument of a method call.

Algorithm 1 Relative Node Order Algorithm

```
 1: function RelativeOrder(node1, node2)                    ▷ returns signed integer
 2:     parent1 ← node1
 3:     parent2 ← node2

 4:     while $depth(parent1) \neq depth(parent2)$ do
 5:         find deeper parent reference and replace it by its parent
 6:     end while

 7:     while parent1 and parent2 are not siblings do
 8:         parent1 ← $parent(parent1)$
 9:         parent2 ← $parent(parent2)$
10:     end while

11:     parent ← $parent(parent1)$               ▷ common AST parent of node1, node2
12:     if parent is a block block then
13:         return $block.childIndexOf(parent1) - block.childIndexOf(parent2)$
14:     else
15:         return $eogDistance(parent, parent1) - eogDistance(parent, parent2)$
16:     end if
17: end function
```

5.7. Order of Transformation Application

To obtain a truly deterministic normalization of the CPG, the transformation step would have to be treated as a fix-point procedure: As long as any transformation can be applied, apply it. This requires checking for new matches of each transformation an extensive number of times. For the sake of runtime optimization, we will define an order on the different transformations that should produce a close approximation of the fixpoint procedure. This order has a major influence on the outcome, so it must be defined carefully.

This section describes two kinds of dependencies that occur between transformations that need to be addressed in the transformation stage.

5.7.1. Dependent Transformations

Given a pair T_1, T_2 of transformations, we call T_2 *dependent* on T_1 if

- as long as T_1 still has matches, the precondition of T_2 cannot hold, or
- the application of T_1 may create additional matches for T_2.

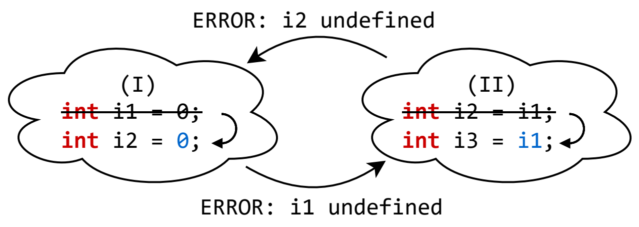
Note that the first condition implies the second. If one or both conditions hold, T_1 should be applied before T_2, or else matches for T_2 may be left untransformed.

Example. We consider the transformations T_1: *Remove Unused Constant*, which removes a `static final` field `CONST` from a class `C` if there are no references to `CONST` in the CPG, and T_2: *Remove Empty Class*, which removes the class declaration of a class `C` from the CPG if `C` has no members. It is apparent that a class `C` that contains an unused class constant `CONST` cannot be empty, thus the first condition holds. After T_1 is applied to `C`, `C` contains one field less than before. If the field removed by T_1 was the last member of `C`, then `C` is now empty and should match the source graph pattern of T_2, thus the second condition holds. If T_2 was to be applied before T_1, potential new matches of T_2 after T_1 would be missed.

5.7.2. Overlapping Instances of the Same Transformation

In the transformation step for a given transformation T, all current matches for T are collected before the first transformation is applied. This can lead to an invalid CPG if different matches share nodes. Consider the sequence of statements shown in Figure 5.7.

```
int i1 = 0;
int i2 = i1;
int i3 = i2;
```



(I)
~~int i1 = 0;~~
`int i2 = 0;`

(II)
~~int i2 = i1;~~
`int i3 = i1;`

Figure 5.7.: For the code snippet on the left, two overlapping matches are found for the transformation *Inline Single-Use Variable* (right).

A transformation T: *Inline Single-Use Variable* (see Section 6.3.1), which replaces the only reference to a variable by its assigned value, will find two matches in these statements and output the following transformation instances:

(I) replace the initial value of `i2` by `0` and remove the variable declaration of `i1`.

(II) replace the initial value of `i3` by `i1` and remove the variable declaration of `i2`.

Regardless of which transformation instance we apply first, we will remove a declared variable that is still referenced in the other transformation operation, so the resulting graph will be semantically incorrect. To deal with this, we have to validate each match right before the corresponding transformation instance is applied. If any invalidated matches are found, they are discarded, and a flag is set to re-run the isomorphism detection in case new matches have emerged in place of the invalidated matches.

5.8. Graph Linearization and Tokenization

After all transformations are complete, the last step of the CPG transformation system pipeline is to extract a token list from the CPG as the input to the comparing mechanism.

To this end, a visitor needs to traverse the CPG in a deterministic order and create tokens for selected node types. As discussed before, in AST-based approaches, the order is typically depth-first-search order along the AST. In our CPG-based approach, there are several options at different code levels that in and of themselves may contribute to the normalization.

Submission level. A submission contains a number of files. The order in which the files appear in the token list has no influence on the result, as matching token sequences cannot span across multiple files.

File level. A code file contains a number of top-level elements (TLEs), like classes, records, enumerations, or interfaces, and matches across multiple of these elements are allowed. It may be beneficial to try to normalize the order of these elements, but considering (1) that each of these elements is likely to produce enough tokens to reach the minimal token match length and the comparison mechanism is already resilient against reordering on a larger scale, and (2) that declaring multiple TLE per file is discouraged and may even cause compile-time errors[1], the deeper levels should get more attention than the file level.

Class level. A Java class (or record, enumeration, interface, etc.) contains declarations for fields, constructors, methods, and inner TLEs. At this level, reordering starts to have a major effect on token matches. These elements can easily be reordered by category, e.g., instance fields first, then static fields, class constants, instance methods, static methods, and then inner TLEs, possibly also sorted by category. The elements of each category should be sorted recursively.

Fields and Constants. The number of tokens produces by field declarations depends largely on their initial value. If there are fields with token-producing initial values, like method calls, then the order of the field declarations plays a role. The field declarations may be sorted by their length, either measured by the number of tokens or CPG nodes in the respective AST subtree.

Methods. Of the class-level elements, the methods are likely to produce the largest number of tokens in total, so matches across method declarations may play a key role, especially if there are many methods of rather small size. The order by size approach used for fields may also be applied for methods.

Another interesting option is to order methods by their rank in the *call graph*, i.e., a depth-first search is conducted from the main method over the method calls, approximating the order in which the methods might be called for the first time in a hypothetical run. However, the construction of the call tree is a considerable effort, and when there are multiple overriding implementations of the same method in different subclasses, then the call tree will become only a conservative approximation of the run-time behavior.

A third option is to precalculate the token sequence of the methods and sort the methods by comparing their token sequence. Using this strategy, we increase the likelihood that corresponding methods are matched even if their token sequence is shorter than the

[1]See Java Language Specification, Java SE 17 Edition, last visited on 11.05.2024

minimum token match. However, this leads to a higher similarity even between pairs of original submissions.

Blocks. The statements of a block, be it a method block or a block of a control statement, have the greatest potential to break matching token sequences when they are reordered, as statements are very diverse in structure. However, starting at this level, dependent elements occur whose order may strongly influence the semantic of the code. These dependencies thus strictly limit the degree of freedom when reordering statements.

Sağlam et al. [38] discuss reordering tokens as a means to normalize the order of statements, and the constraints that must be respected in the process. To briefly summarize, they categorize tokens into a two-dimensional matrix based on (1) the degree of freedom to which they may to be permutated, and (2) their contribution to the program behavior. The first factor determines the partial order on the tokens: Tokens with *fixed order* must remain in position relative to all other tokens; different accesses to variables must remain in order relative to one another; and the boundaries of loops must be upheld, even if a token may occasionally be moved from the end of the loop to the start or vice versa. For the second factor, the concept of a *critical token* [38, p. 5] is introduced, meaning a token that represent code that directly contributes to the visible behavior of the program, including method calls, class declarations, and control structures. This is the equivalent of what we will refer to as an *essential statement* in this thesis. Essential statements are used to determine *dead* code, i.e., all statements that have no data flow towards any essential statements. For the statements that are neither essential nor dead (or *irrelevant*), we will use the term *relevant*.

Deviating from the work of Sağlam et al., our reordering algorithm will operate on nodes instead of tokens, and the scope of the reordering will be individual blocks (excluding statements of inner blocks), which should severely limit the number of statements which are considered for insertion at any one time. Furthermore, control structures will only be treated as essential if they contain essential statements in their inner blocks; otherwise, the complete control structure may very well be irrelevant code. Lastly, we relax the restriction that control statements are in fixed order: Statements may be allowed to swap position with control statements if there are no data dependencies between the two statements. This allows for more freedom while reordering while still maintaining the correctness of the resulting order.

A potential topological order of the statements can be determined as follows:

- Remove any loops from the data dependency graph by discarding all edges that point backwards.
- Out of the list of statements in the block `b`, select and remove those that have no successors in the data dependency graph (DDG), add them into the list `ready` and sort them by their category as follows:
  1. jumping nodes (`return`, `continue`, `break`)
  2. non-essential relevant nodes
  3. essential nodes

- While `ready` is not empty:
  - Remove the first statement `s` from `ready` and insert it at the top of the block `b`.
  - Reorder the inner blocks of `s` recursively, if there are any.
  - Determine the statements that have become ready now that `s` is inserted and add them into their respective category in the `ready` list.

This procedure keeps dependent statements close together, and if a loop contains an increment statement for the loop variable, it is placed at the end of the block, if possible (as it is non-essential and relevant). Moreover, dead statements are eliminated.

To complete the deterministic order of the statements, the individual categories of `ready` (jumping, relevant, essential) must also be sorted. The order by AST subtree size or by number of tokens discussed above can be applied here as well.

Statements. Traditionally, the elements of individual statements are ordered by their appearance in depth-first search on the AST. In this work, they will be ordered by their appearance in the EOG instead, which approximates the order in which the individual nodes are evaluated at runtime. This is also the approximate order in which individual elements of a compound statement might be extracted, which mitigates such attacks (if inlining of single-use variables is not applied already).

Example. Table 5.1 shows two equivalent code snippets, where in the second snippet, various parts have been extracted to proper statements. This causes a great deal of permutation in the AST-based tokenization, but no changes to the EOG-based tokenization.

Code snippet	AST-based tokenization	EOG-based tokenization

`for (int i = 0; i < 9; i++) {`
`    double d = sqrt(i);`
`    System.out.println(++d);`
`}`
	`FOR VARDEF ASSIGN`
	`VARDEF APPLY`
	`APPLY ASSIGN`
		`VARDEF FOR`
		`APPLY VARDEF`
		`ASSIGN APPLY`
		`ASSIGN // i++`

`int i = 0;`
`for (; i < 9; i++) {`
`    double d = sqrt(i);`
`    d++;`
`    System.out.println(d);`
`}`
	`VARDEF`
	`FOR ASSIGN`
	`VARDEF APPLY`
	`ASSIGN`
	`APPLY`
		`VARDEF`
		`FOR`
		`APPLY VARDEF`
		`ASSIGN`
		`APPLY`
		`ASSIGN // i++`

Table 5.1.: Linearization of different versions of equivalent code map to the same token list if traversed along the EOG, but different token lists if traversed along the AST.

This concludes the system description of the CPG transformation system. As it is only a framework, it has no transformations built in. To complete the approach, we need to define a set of code transformations suitable for the purpose of code normalization. Let us now take a look at our proposed selection of transformations, which is the second core contribution of this thesis.

6. CPG Transformations for Refactoring Obfuscation Resilience

In this chapter, we illustrate a set of thirteen CPG transformation schemes, each of which is intended to add to the normalization of code graphs while preserving their visible behavior. It is only when these transformations are integrated into the CPG transformation system (see Chapter 5) that it becomes an effective approach to detect plagiarisms with high resiliency against refactoring attacks. The transformation selection represents the second core contribution of this thesis.

As discussed in Chapter 4, no selection of transformations can ever cover all possible refactoring attacks. Therefore, the transformations described here shall be understood as a solid base that may be extended by any user as they see fit.

We divide the transformations into four categories: Transformations that *remove elements* exclude elements from the CPG which are regarded as irrelevant in terms of the unique behavior of the code. Transformations *moving members* determine elements that are not defined in the same class where they are used and move these elements closer to the code that references them. Transformations *inlining elements* revert the extraction of code elements to new variables, fields, or methods. Finally, transformations that perform *semantically equivalent replacement* address the possibility of two valid possibilities to express the same semantic and ensure that only one of these possibilities persists in the code by transforming one option to the other.

The statement reordering and dead code elimination, which are transformations in their own right, require extensive analysis in order to determine where they can be applied, and so do not fit the scheme used by the CPG transformation graph, even though they use the same transformation operations. See Section 5.8 for more information about these transformations.

Each transformation will be presented in three ways:

- The source and target graph patterns (including the syntactic context needed),
- A schematic code fragment that shows the effect of the transformation on the affected code, and
- A token list fragment that shows the effect of the transformation on the token list.

As is common for the visualization of graph transformations, the left side (in blue) represents the state before the transformation, or phrased differently, the precondition, and the right side (in blue) shows the state after the transformation, the postcondition.

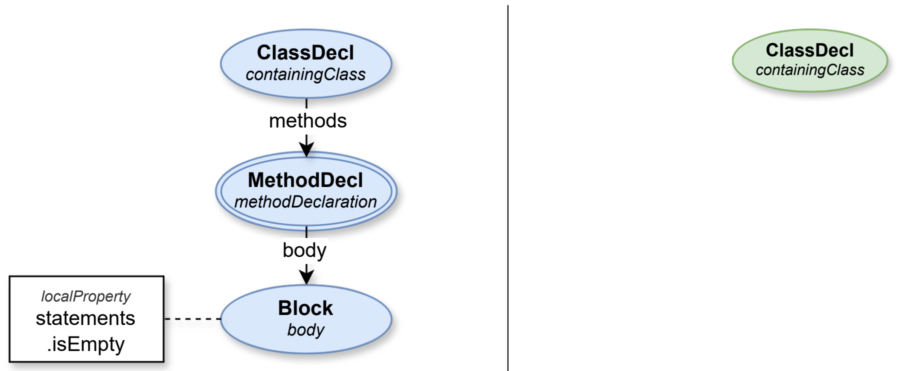
6.1. Removing Elements

These transformations remove parts of the code that does not represent code relevant to the program behavior. Among other possibilities, such code may be introduced by dead code insertion attacks, as a byproduct of other transformations, or as common code, which we can neglect for the tokenization.

6.1.1. Empty Methods and Constructors

Empty methods and constructors can occur rightfully in code, e.g., to override the non-empty method of the superclass; in the context of similarity detection, however, they are considered common code [28] and can therefore be discarded so that more unique parts of the code are emphasized in the comparison. This transformation removes empty methods and constructors from classes (or similar structures), which removes $3 + \#param$ tokens from the token list.

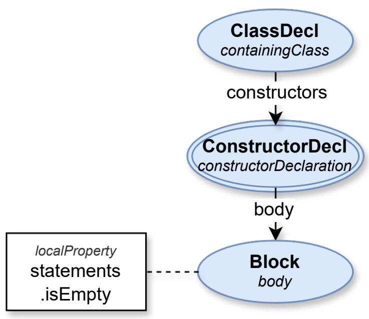
Note that methods with a non-`void` return type cannot be empty, as they need to have a `return` statement. Trivial non-`void` methods are covered in Section 6.1.3.

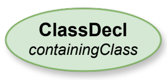


`public void myMethod() {`
`}`
	`// removed`

`METHOD_DECLARATION`
`PARAM*`
`METHOD_BODY_START`
`METHOD_BODY_END`
	`<none>`

Figure 6.1.: Illustration and schema for the transformation *Remove empty method*.





```java
public MyClass(String name,
               int id, ...) {
}
```

```java
// removed
```

```
CONSTRUCTOR_DECLARATION
PARAM*
CONSTRUCTOR_BODY_START
CONSTRUCTOR_BODY_END
```

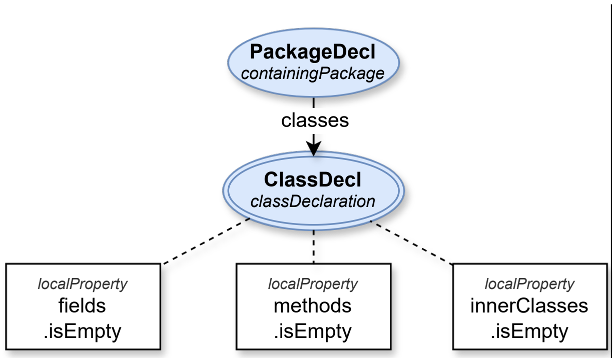
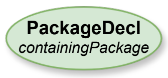
```
<none>
```

Figure 6.2.: Illustration and schema for the transformation *Remove empty constructor*.

6.1.2. Empty Classes

Empty classes introduce no new properties or behavior that would distinguish instances from their supertype, and so there seems to be no compelling reason they should appear in code. At least, instances of empty classes have different compatibility properties in generic types and can be identified using explicit type checking. In the context of similarity detection, empty classes are considered common code [28] and can thus be discarded to avoid meaningless matches, removing three tokens from the token list.

The transformation defines empty classes to be classes with no fields, methods, or inner classes.

 PackageDecl containingPackage classes ClassDecl classDeclaration localProperty fields .isEmpty localProperty methods .isEmpty localProperty innerClasses .isEmpty	

`public MyClass {`
`}`
	`// removed`

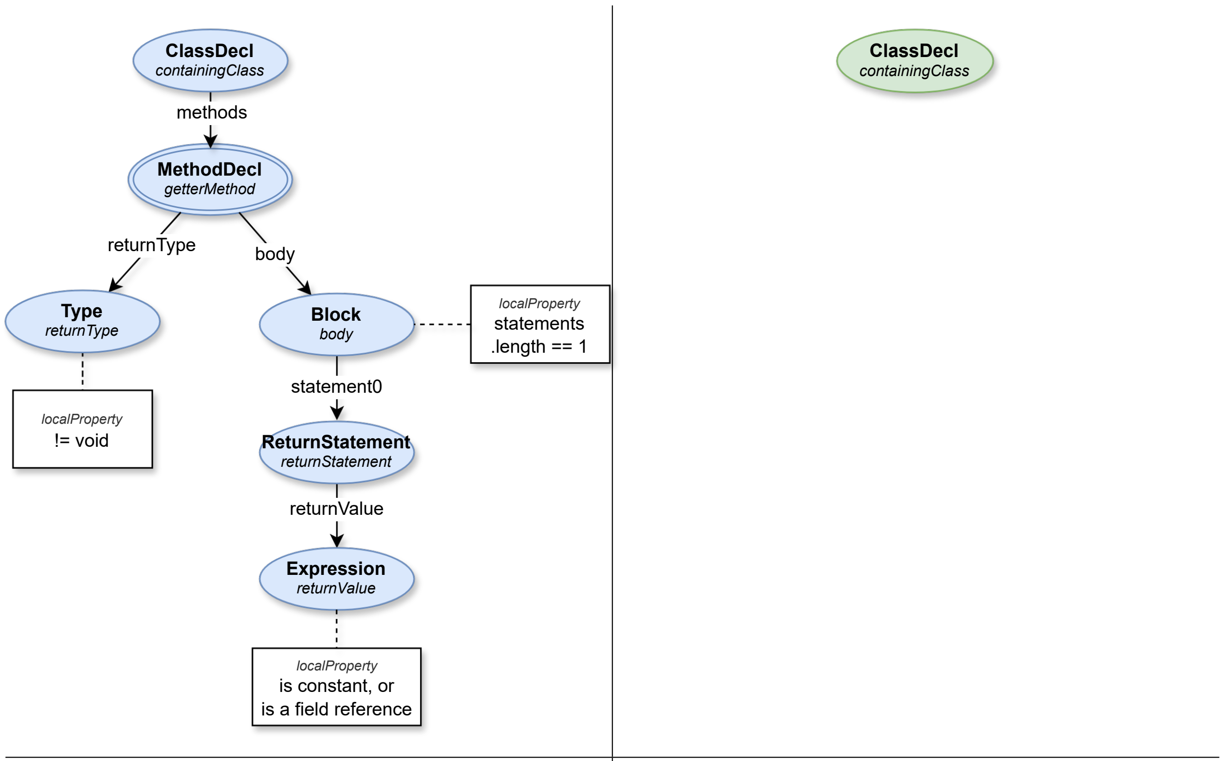
`CLASS_DECLARATION`
`CLASS_BODY_START`
`CLASS_BODY_END`
	`<none>`

Figure 6.3.: Illustration and schema for the transformation *Remove empty class.*

6.1.3. Getter Methods

Getter methods regularly appear in large numbers in object-oriented code, many of which share the same structure. Although their use is encouraged for the sake of data encapsulation, in similarity detection, they cause a large number of matches that are not considered meaningful. As *common code* [27], they can be discarded, removing at least four tokens from the token list.

Note that calls to the removed getter method are kept, which would create invalid references in the code. In the CPG, this causes no problems.



```
public String getName() {
    return this.name;
}
```

```
// removed
```

```
METHOD_DECLARATION
PARAM*
METHOD_BODY_START
RETURN
METHOD_BODY_END
```

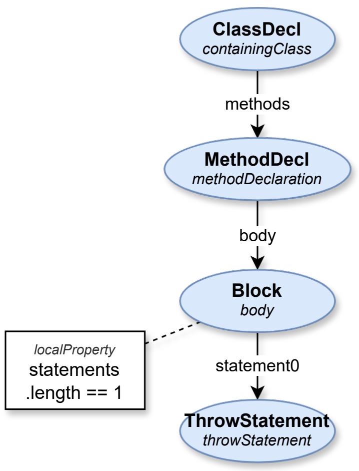
```
<none>
```

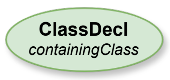
Figure 6.4.: Illustration and schema for the transformation *Remove getter*.

6.1.4. Unsupported Methods and Constructors

Methods and constructors that immediately throw an exception when called, which we will call *unsupported*, are not explicitly listed as a form of *common code* by Novak [27] or Simon et al. [42], but as these methods are evidently not supposed to be called, they do not represent part of the program behavior and can therefore be omitted. This transformation removes unsupported methods and constructors from the CPG, which removes at least 4+#*param* tokens for the method declaration, and possibly some tokens for the construction of the `Exception` object.

Note that the example code snippets and tokens in Figures 6.1 and 6.2 show only one possible example instance. Any method or constructor whose body contains only a throw statement will be matched by this transformation.





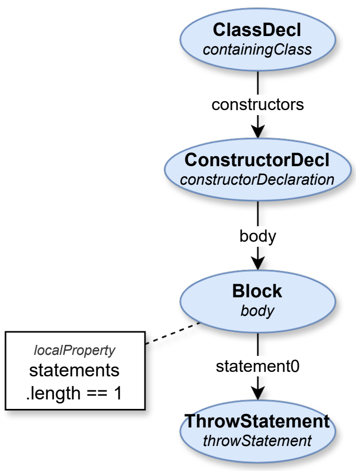
```
public void myMethod(String name,
                     int id, ...) {
    throw new MyException("Error!");
}
```

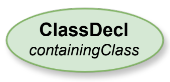
```
// removed
```

```
METHOD_DECLARATION
PARAM*
METHOD_BODY_START
NEW_OBJECT
THROW
METHOD_BODY_END
```

```
<none>
```

Figure 6.5.: Illustration and schema for the transformation *Remove unsupported method.*





```
public MyClass(String name,
               int id, ...) {
    throw new MyException("Error!");
}
```

```
// removed
```

```
CONSTRUCTOR_DECLARATION
PARAM*
CONSTRUCTOR_BODY_START
NEW_OBJECT
THROW
CONSTRUCTOR_BODY_END
```

```
<none>
```

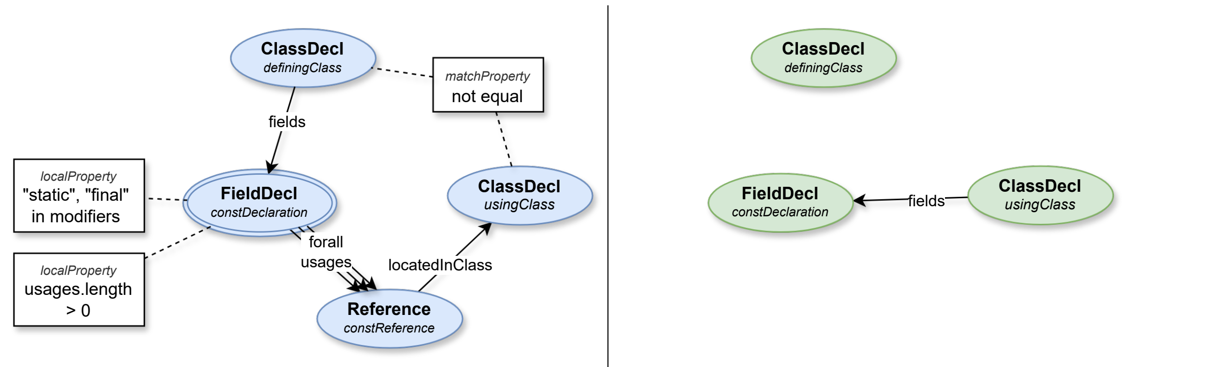
Figure 6.6.: Illustration and schema for the transformation *Remove unsupported constructor*.

6.2. Moving Members

This category of transformations moves members of classes to other classes where they are most relevant, according to a usage analysis. As of now, only one transformation of this category was selected. Another related example is *auxiliary methods*, which, much like constants, are commonly collected in *utility classes*, even if these auxiliary methods are ultimately used in other classes only.

6.2.1. Moving Constants from Constant Classes

This transformation is designed to counter the formation of constant classes, i.e., special classes with the only purpose to act as a central storage for constants. The transformation moves a constant declaration if all of its usages are located in the same other class. This causes at least one token to move within the token list.



```
public class MyConstants {
    public static final String MESSAGE
        = "Yes(Y) or No(N)";
}
public class MyClass {
    void printMessage() {
        output.print(MESSAGE);
    }
}
```

```
public class MyConstants {
    // removed
}
public class MyClass {
    public static final String MESSAGE
        = "Yes(Y) or No(N)";

    void printMessage() {
        output.print(MESSAGE);
    }
}
```

```
CLASS_DECLARATION
FIELD_DECLARATION
...
CLASS_DECLARATION
...
```

```
CLASS_DECLARATION
...
CLASS_DECLARATION
FIELD_DECLARATION
...
```

Figure 6.7.: Illustration and schema for the transformation *Move constant to only using class.*

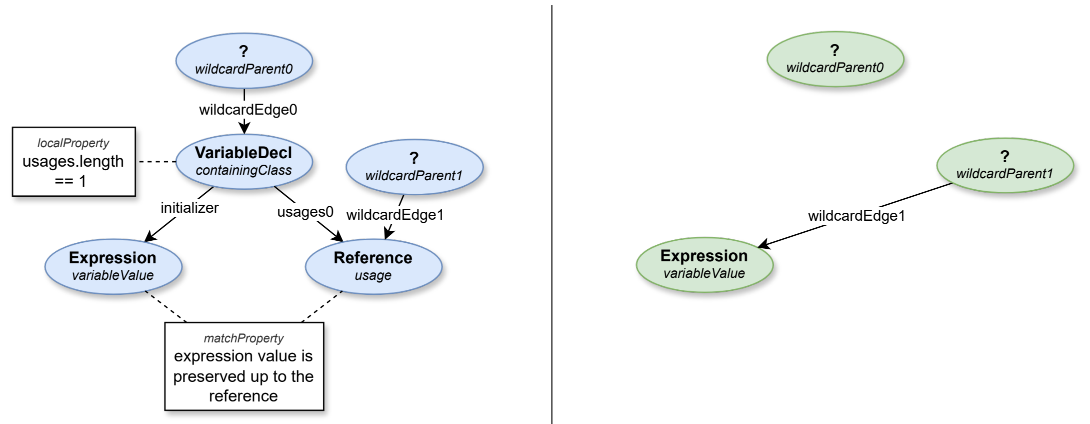
6.3. Inlining Elements

Inlining transformations reverse the extraction of elements to new named constructs, e.g., subexpressions to variables, class members to new classes. The respective tokens are moved within the token list, therefore extraction refactorings constitute an effective attack against token-based plagiarism detectors.

6.3.1. Single-Use Variables and Constants

The extraction of a subexpression to a new variable or constant constitutes a Level 3 plagiarism, see Table 2.1 in Section 2.1. Such an extraction refactoring often produces a single-use variable or constant. Thus, in order to balance the risk of information loss and the potential benefit of this transformation for the similarity detection, we opt to inline variable declarations only if they are used exactly once. In this case, the transformation replaces the variable/constant reference by its value and removes the variable/constant declaration, removing one token and moving the expression tokens.

In the case of variables, this transformation only preserves the semantic of the code if all subexpressions can be guaranteed to have the same value in the position of the usage as in the position of the former variable declaration. To determine whether this is the case, a data flow analysis is required.



```
double singleUse = expression;
// ...
return singleUse * otherTerm;
```

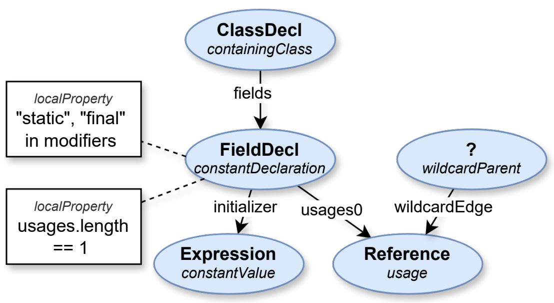
```
return expression * otherTerm;
```

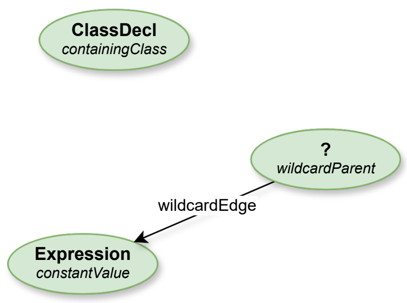
```
<variable value>
VARIABLE_DECLARATION
...
<variable usage>
```

```
...
<variable value>
<variable usage>
```

Figure 6.8.: Illustration and schema for the transformation *Inline single-use variable.*

In the case of class constants, it may occur that the expression assigned the constant is not constant per se, i.e., if evaluated at different instances during execution, their value may vary. However, as this can be seen as an edge case, and for the sake of not being overly restrictive, the transformation does not verify that the value of the constant is truly constant.





```
public static final
      double singleUse = expression;
// ...
return singleUse * otherTerm;
```

```
return expression * otherTerm;
```

```
<constant value>
FIELD_DECLARATION
...
<field usage>
```

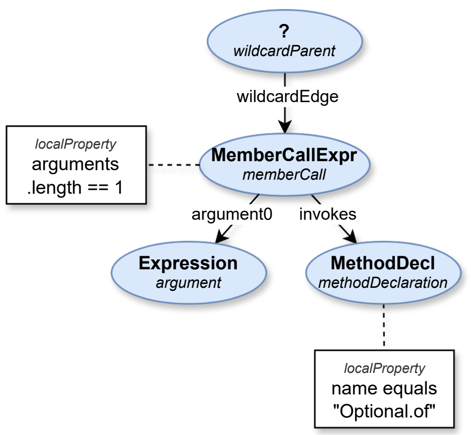
```
...
<field value>
<field usage>
```

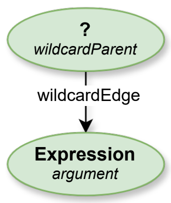
Figure 6.9.: Illustration and schema for the transformation *Inline single-use constant*.

6.3.2. Unwrapping Optional Values

This refactoring replaces an optional value (an `Optional` object) by its wrapped value. This is accomplished with two distinct transformations: The first transformation replaces calls to `Optional.of(expression)` by the proper `expression`, which removes one token from the token list. The second transformation replaces calls to `Optional::get()` by the expression on which the method is called, also removing one token. Both methods are identified with the fully qualified name.

While these two transformations cannot transform all functionality of the `Optional` interface, they are sufficient to counter simple attacks where a variable is changed to an `Optional` object containing its value. As a potential extension, calls to the methods `Optional::isPresent` and `Optional::isEmpty` can be replaced by `null` checks.





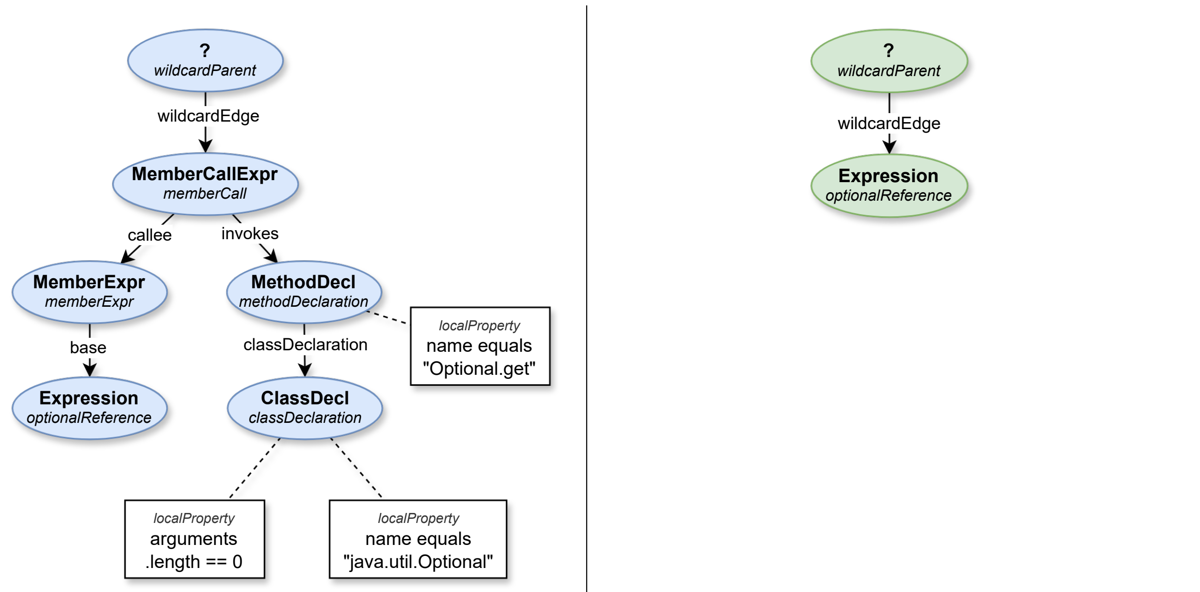
```
Optional<Type> maybeValue
                    = Optional.of(expression);
```

```
Type realValue = expression;
```

```
<expression value>
METHOD_CALL
VARIABLE_DECLARATION
```

```
<expression value>
VARIABLE_DECLARATION
```

Figure 6.10.: Illustration and schema for the transformation *Unwrap Optional.of()*.



`return maybeValue.get();`	`return realValue;`

`METHOD_CALL`
`<expression value>`
	`<expression value>`

Figure 6.11.: Illustration and schema for the transformation *Unwrap Optional.get()*.

6.4. Semantically Equivalent Replacement

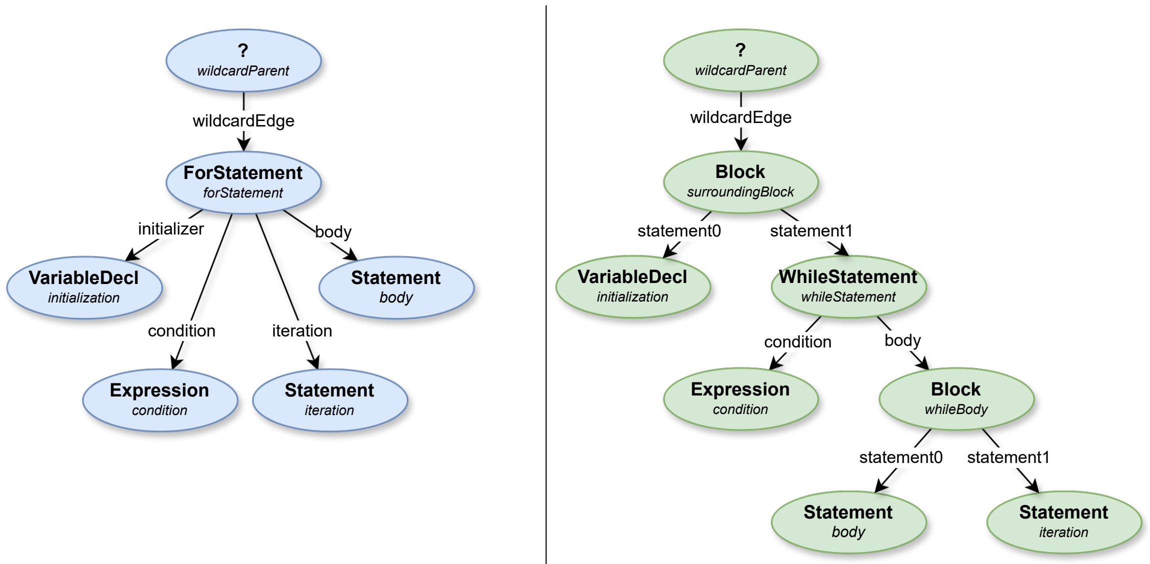
Unlike the previous categories, when performing semantic-preserving transformations, the result is not necessarily less code, but other code that has indistinguishable visible program behavior.

6.4.1. For Loop to While Loop

According to the research of Karnalim [18], exchanging equivalent loop types is a common type of Level 5 plagiarism. A classical example is the transformation of a `for` loop to a `while` loop, which follows a standard pattern. While equally possible, it is more complex to go the opposite way and construct a `for` loop from a `while` loop, as a meaningful loop variable must first be identified.

This transformation replaces a `for` statement with a block containing the loop variable initialization and a `while` statement. The `while` statement uses the condition of the `for` statement, and the increment statement is moved to the end of the `while` block. This changes the type of three tokens from types specific to `for` loops to types specific to `while` loops, and moves the token of the iteration statement to the very end of the `while` block token.

Note that the additional surrounding block around the `while` statement is necessary to keep the semantic correct. This is because the loop variable becomes undefined after the end of the `for` statement, and so, it must also become undefined after the `while` statement, which can be accomplished with the additional block. Without it, identifiers may clash if a variable with an equal identifier is declared later in the method, for example, by another loop statement.



```
for (int i = 0; i < iMax; i++) {
    // for body statements
}
```

```
{
    int i = 0;
    while (i < iMax) {
        // for body statements
        i++;
    }
}
```

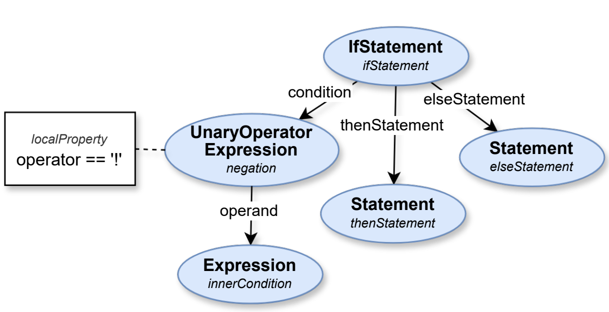
```
VARIABLE_DECLARATION
FOR_STATEMENT
FOR_BODY_START
<body statements>
FOR_BODY_END
ASSIGN
```

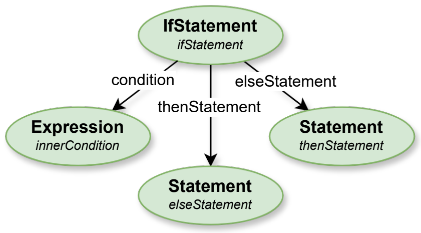
```
VARIABLE_DECLARATION
WHILE_STATEMENT
WHILE_BODY_START
<body statements>
ASSIGN
WHILE_BODY_END
```

Figure 6.12.: Illustration and schema for the transformation *For Statement to While Statement.*

6.4.2. Negated If-Else Condition

Current IDEs offer to automatically invert the condition of an `if` statement and exchange its *then* and *else* blocks if that is possible. This transformation reverts the swap if the condition is surrounded by a negation operator, and removes the negation operator. As the negation operation is not represented in the token list, only the tokens of the *then* and *else* blocks swap position.





```
if (!expression) {
    // then statements
} else {
    // else statements
}
```

```
if (expression) {
    // else statements
} else {
    // then statements
}
```

```
<condition>
IF_STATEMENT
THEN_BLOCK_START
<then statements>
THEN_BLOCK_END
ELSE_BLOCK_START
<else statements>
ELSE_BLOCK_END
IF_BLOCK_END
```

```
<condition>
IF_STATEMENT
THEN_BLOCK_START
<else statements>
THEN_BLOCK_END
ELSE_BLOCK_START
<then statements>
ELSE_BLOCK_END
IF_BLOCK_END
```

Figure 6.13.: Illustration and schema for the transformation *Invert negated If condition.*

7. Evaluation

In this chapter, we want to evaluate the novel CPG transformation system in order to assess whether it accomplishes its two major design goals: We aimed to improve the general resilience of token-based plagiarism detection against obfuscation attacks while preserving its performance and reliability. To this end, the evaluation was set up as follows:

Ten data sets of submissions with plagiarisms were collected. The original submissions are real-world submissions by university students, from which other authors have created seven plagiarisms with multiple different attack schemes. Three new data sets with a total of 450 new plagiarisms were generated using automatic refactoring obfuscation attacks.

As a representative of the state of the art to compare our approach to, we chose JPlag, as it is widely used and open-source (see Section 2.3.2). To evaluate the effects of the respective linearization and tokenization of the approaches, we disabled all other components of both approaches and ran them on the data sets. This was to ensure that the observed effects of the CPG approach would not be biased by the token selection and linearization method.

Next, the CPG approach and JPlag were run on all ten data sets with their full optimization capacity. The results were accumulated sorted by the attack scheme used, so that the quality of the plagiarism detection could be compared separately for each attack scheme. As a measure for the plagiarism detection quality, we used the average distinctiveness of the similarity of plagiarisms and originals. During both runs, the processing time of each submission was recorded.

The results show that the CPG approach is considerably more effective at distinguishing refactoring plagiarisms from originals. For all other attack types, the distinctiveness of plagiarisms and originals of the CPG approach matched those of JPlag.

In the next sections, the individual components of the evaluation are described. Then, we look at the results and discuss our conclusions. Finally, we consider potential threats to validity.

7.1. Goal-Question-Metric Plan

With the design of the CPG transformation system and a selection of transformations, we intend to make token-based plagiarism detection more resilient against the serious issue of obfuscation attacks. As an early step in the design phase, a Goal-Question-Metric (GQM) plan [2] was set up in order to link the achievement of this goal to concrete metrics.

The individual questions and metrics of the GQM plan address various functional and non-functional aspects of our primary objective. Given that the CPG normalization is the core novelty of this approach, we will assess its impact most carefully, based on four different obfuscation attack schemes.

G.1 Make plagiarism detection more resilient against refactoring obfuscation attacks

Q.1.1 How does the **linearization** by the CPG approach affect the quality of plagiarism detection?

M.1.1.1 Compare the similarity metrics of JPlag and the CPG approach with **no** normalizations enabled.

Q.1.2 How does the **normalization** by the CPG approach affect the quality of plagiarism detection?

M.1.2.1 Compare the similarity metrics of JPlag and the CPG approach with **all** normalizations enabled.

G.2 Preserve the performance of JPlag

Q.2.1 How does the **generation** of the CPG and **linearization** affect the runtime of the CPG approach?

M.2.1.1 Compare the runtime of JPlag and the CPG approach with **no** normalizations enabled.

Q.2.2 How does the application of the **transformations** affect the runtime of the CPG approach?

M.2.2.1 Compare the runtime of JPlag and the CPG approach with **all** normalizations enabled.

M.2.2.2 Compare the runtime of the CPG approach **with all and without any** normalizations enabled.

Q.2.3 How reliable is the CPG approach?

M.2.3.1 Compare the failure rate of the dedicated language modules and the CPG approach with **all** normalizations enabled.

7.2. Data Sets

This section presents the data sets from which plagiarisms were generated for the evaluation.

PROGpedia Task 19 and 56. The PROGpedia data set [30] contains the submissions to sixteen real-world programming tasks by undergraduate computer science students at the University of Porto, submitted in Java, Python, C++, and C. For the evaluation, the Java submissions for task 19 and task 56 were used, as they contained the greatest number of valid solutions, and the solutions had more LOC than other tasks.

In task 19, the students designed an analysis framework for social networks, represented as graphs, which determines isolated groups of related people, their members, size, and the total number of groups. The format of input and output data was defined precisely. The data set contains 244 correct submissions by 175 students written in C, Python, and Java, including 69 resubmissions. Sağlam et al. [38] prepared a clean subset, which only contains the final Java submissions and no plagiarisms. The resulting data set contained $n = 27$ single-file submissions ranging from 40 to 345 LOC, with a mean of 131 LOC and a median of 106 LOC.

In task 56, the students were assigned an optimization problem where they had to find the minimal spanning tree of a set of nodes and determine the total length of its edges. Again, the format of input and output data was given. The data set contains 108 correct submissions by 89 students written in C, C++, and Java, including 19 resubmissions. Sağlam et al. [38] prepared a clean subset, which only contains the final Java submissions and no plagiarisms. The resulting data set contained $n = 28$ single-file submissions with a range of 40 to 316 LOC, with a mean of 84.6 LOC and a median of 77 LOC.

The TicTacToe Data Set. The TicTacToe data set consists of submissions for a mandatory exercise task of the introductory programming course for undergraduate computer science students at Karlsruhe Institute of Technology (KIT) in Java. The task was to implement a tic-tac-toe game, which should be playable interactively via a CLI both against a second human player and the computer. The data set contains 738 submissions written in Java. Sağlam et al. [38] prepared a clean subset, which contains no plagiarisms, and only submissions that compile successfully. The resulting data set contained $n = 626$ submissions with a range of 33 to 662 LOC, with a mean of 236 LOC and a median of 225 LOC. Each submission contained up to 16 files, with a mean of 3.8 files and a median of 3 files.

7.3. Generated Plagiarisms by Attack Scheme

For the evaluation, nine plagiarized data sets were used that incorporated one of four specific obfuscation attack schemes each. We base the evaluation on this diverse data set in order to ensure that the CPG approach preserves the effectiveness of JPlag on all attack schemes, and to ensure the validity of our results. First, we consider dead statement insertion attacks, which used to be a severe vulnerability of token-based plagiarism detection exploited by MossAD, until this vulnerability was addressed by token sequence normalization [38]. Next, we include two types of attacks supported by the use of LLMs, which are an accessible way to generate plagiarisms with little effort or skill. Finally, we take a look at combined refactoring attacks, which create a structurally different, semantically equivalent version from a submission without creating dead code.

This section presents the different plagiarized versions of the data sets used for evaluation. For each plagiarized data set, we give the number of original submissions o and the number of plagiarized submissions p.

7.3.1. Insertion Attacks

For his bachelor's thesis [3], Brödel created the data sets *prog19-insert* ($o = 27, p = 27$) and *prog56-insert* ($o = 28, p = 28$) with his tool *JPlag-Gen*. Much like the automatic plagiarism generation tool MossAD, JPlag-Gen iteratively inserts random statements into code blocks, and compiles and runs the intermediate result to check whether the insertion preserved the semantic of the submission. For each of the 27 and 28 submissions, the respective *insert* data sets contain the original and one plagiarism, labeled with the index of the original submission. The plagiarisms created by Brödel are available via the additional material for the paper complementing his bachelor's thesis [39]. Using JPlag-Gen, Niehues created 51 plagiarisms for the TicTacToe data set (*tictactoe-insert*, $o = 626, p = 51$) [26].

7.3.2. LLM-Generated Attacks

Submission Obfuscation Attacks. For his master's thesis [25], Niehues created the data sets *prog19-obfuscated* ($o = 27, p = 75$), *prog56-obfuscated* ($o = 28, p = 75$), and *tictactoe-obfuscated* ($o = 626, p = 75$) by submitting original submissions to ChatGPT with different prompts to create derived plagiarisms with similar semantics, but different representation in code. The different prompts were phrased specifically so that the resulting plagiarisms would look vastly different to traditional token-based plagiarism detection compared to the originals. For each of the base data sets, five original submissions were selected, and fifteen plagiarisms were created from each of these submissions, which makes 75 in total.

Fully Generated Submissions. Additionally, Niehues presents the *FullyGenerated* ($o = 626, p = 10$) data set where ChatGPT was supplied with the original specification of the *tictactoe* task as it was handed out to the students, but no code to build upon, and was prompted to create a new solution for the task. This process was repeated ten times.

7.3.3. Refactoring Transformation Attacks

To be able to evaluate the resilience of the CPG transformation system against refactoring attacks, a new automatic plagiarism generation tool was implemented, which we will call *PlagGen*. PlagGen was designed specifically so that the transformations used to generate the plagiarisms would not be based on the same mechanics as the transformation system that is supposed to revert these same transformations. It uses the Inria Spoon API for Java Source Code Analysis and Transformation [32] to parse Java code submissions, iteratively apply transformations to them, and generate code from the transformed AST. The following transformations were used in combination:

- *Wrap in Optional*: Variable declarations are boxed into an `Optional` object, i.e., assignments use `Optional.of` to box the assigned value, and variable reading references use `Optional.get` to unbox the value.
- *Extract Variable*: A random set of expressions is selected, each of which is extracted to a new variable and their original occurrence replaced by a reference to that variable.
- *Extract Constant Class*: A random set of expressions is selected, each of which is extracted into a class constant of a new class in a separate file, and their original occurrence replaced by a reference to that constant.
- *Swap If-Else*: The conditions of `if` statements are inverted, either by negating it with a negation operator around the condition, or by replacing the equality operator `==` by the inequality operator `!=` and vice versa. Also, the then and else statements are swapped and, if necessary, wrapped into a block statement.
- *Insert Unsupported Constructor*: A new constructor, which immediately throws a runtime exception, is added at a random position in the class body.
- *Insert Unsupported Method*: A new method, which immediately throws a runtime exception, is added at a random position in the class body.
- *Insert Getter*: A new method, which immediately returns a field value, is added at a random position in the class body.

The individual generated plagiarisms are distinct from each other in that a random number generator decides which of all possible transformation instances are applied, using a suitable application rate for each transformation.

With PlagGen, fifteen submissions were randomly selected from the *prog19*, *prog56*, and *tictactoe* data sets, and ten plagiarisms generated for each of them. The results are the new data sets *prog19-transfromed* ($o = 27, p = 150$), *prog56-transformed* ($o = 28, n = 150$), and *tictactoe-transformed* ($o = 626, p = 150$).

7.4. Approaches Used for Comparison

Four approaches were used in the various stages of the evaluation.

Linearization Evaluation: Approaches without optimizations. To evaluate the influence of the linearization and tokenization, the AST-based default configuration of JPlag ("JPlag") was compared to the CPG-based approach with all transformations deactivated ("CPG"). This ensured that the influence of the linearization and token selection could be examined in isolation, without interference of transformations, reordering, or other optimizations.

Resilience Evaluation: Approaches with optimizations. To evaluate the resilience of the CPG approach, JPlag with token sequence normalization enabled ("JPlag-N") was compared to the CPG-based approach will all transformations and reordering enabled ("CPG-N").

For each of these approaches, the minimum token match size was set to 9, the default value of JPlag for Java submissions.

7.5. Results

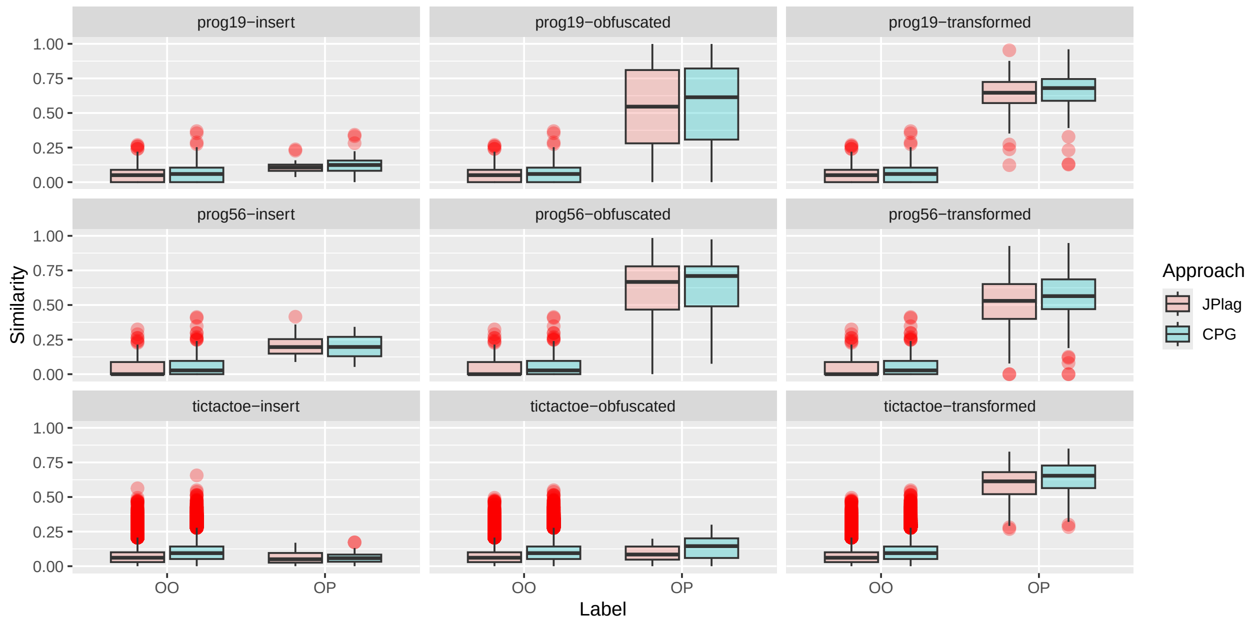
For the evaluation of the plagiarism detection quality, the results will be presented in two ways: First, the distribution of similarity values of all submission pairs will be presented in a boxplot categorized by the data sets and the approaches used, as well as the relation between the compared submissions: Pairs of originals and plagiarisms (of that original) will be labeled as *Original vs. Plagiarism* or *OP* for brevity, and pairs of unrelated originals will be labeled as *Original vs. Original* or *OO*.

Secondly, key statistics of the distribution will be presented in a table categorized by approach and relation between the compared submissions. The rows labeled *Diff* show the statistics of the element-wise difference between the similarities from the different approaches, which (apart from the mean) is different from the difference of the statistics from the different approaches.

To motivate how we measure the plagiarism detection quality, we assume the following scenario: an instructor sets out to detect plagiarisms in a submission set. They use a token-based plagiarism detection tool as a basis for the examination of suspicious pairs. The result is presented to the instructor as a list of the pairwise similarities, and a histogram. As it is infeasible to check all $O(n^2)$ pairs of submissions for plagiarism, the instructor opts to examine only the most similar pairs.

In this scenario, the correct classification of the unlabeled originals and plagiarisms depends on the extent to which pairs of originals and plagiarisms are *distinctly more similar* to each other compared to pairs of originals. Thus, in the evaluation on our labeled data sets, we calculate statistical measures of the original pairs m_{OO} and pairs of originals and plagiarisms m_{OP} and use the difference $m_{OP} - m_{OO}$, the distinctiveness of plagiarisms among originals, as an indicator for the plagiarism detection quality of each approach.

7.5.1. Linearization



Approach	Mean	Median	IQR	Q3
JPlag-N	0.0697	0.0611	0.0711	0.1006
CPG-N	0.1016	0.0942	0.0904	0.1423
Diff	0.0319	0.0291	0.0603	0.0603

Figure 7.1.: Evaluation of the approaches "JPlag" and "CPG" (see Section 7.4) on all data sets.

Figure 7.1 shows the results of running JPlag with no optimization and the CPG approach reduced to CPG construction, linearization, and tokenization, without any transformations or reordering. This isolates the effects of the linearization from those of further optimizations; the latter are examined in 7.5.2.

Q.1.1 How does the **linearization** by the CPG approach affect the quality of the plagiarism detection?

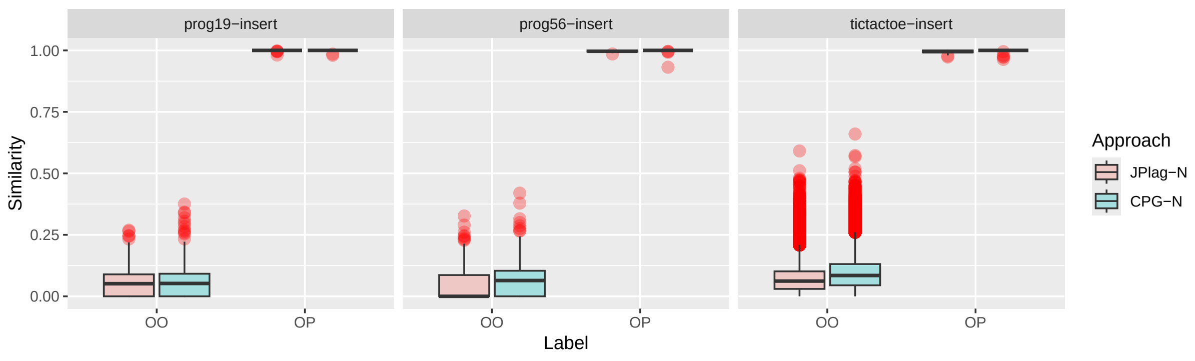
A.1.1 With an average and median difference of three percentage points, the linearization shows no major effect on the similarity rating; instead, a rather stable offset of three percentage points is introduced that should have little influence on how well *OP* pairs can be distinguished from *OO* pairs. Thus, if the CPG approach *with* transformations and reordering produces different results than JPlag, where the difference goes beyond the offset, we can confidently attribute this difference to the transformation and reordering instead of the linearization and tokenization process.

7.5.2. Transformations

This section examines the effectiveness of the approaches on specific obfuscation attacks. We evaluate the approaches on each attack scheme by taking into account only those plagiarisms that were generated using that particular attack.

7.5.2.1. Insertion Attacks

Figure 7.2 shows the result of running JPlag with optimizations and the full CPG approach with all transformations and reordering on the data sets plagiarized with insertion attacks.



— Originals vs. Originals (OO) —				
Approach	**Mean**	**Median**	**IQR**	**Q3**
JPlag-N	0.0700	0.0619	0.0716	0.1015
CPG-N	0.0929	0.0846	0.0863	0.1310
Diff	0.0229	0.0225	0.0579	0.0535

— Originals vs. Plagiarisms (OP) —				
Approach	**Mean**	**Median**	**IQR**	**Q3**
JPlag-N	0.9960	0.9964	0.0058	1.0000
CPG-N	0.9978	1.0000	0.0000	1.0000
Diff	0.0018	0.0024	0.0053	0.0053

— Distinctiveness OP vs. OO —			
Approach	**Mean**	**Median**	**Q3**
JPlag-N	0.9260	0.9345	0.8985
CPG-N	0.9049	0.9154	0.8690
Diff	-0.0211	-0.0201	-0.0482

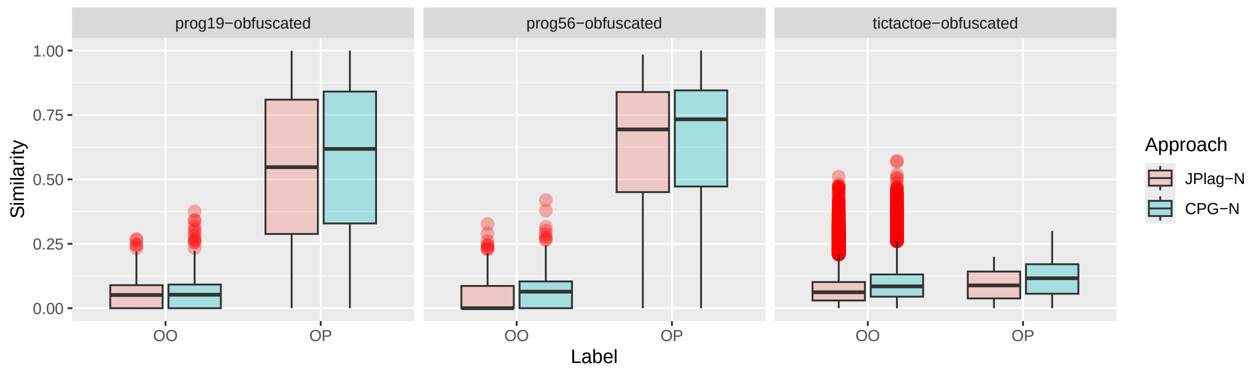
Figure 7.2.: Evaluation of the approaches “JPlag-N” and “CPG-N” (see Section 7.4) on data sets with insertion attack plagiarisms.

Q.1.2 How does the **normalization** by the CPG approach affect the quality of the plagiarism detection?

A.1.2 The results show that the similarity distributions on all three data sets are very similar between the JPlag and CPG approaches. Both approaches produce substantial disambiguation gaps between *OO* and *OP* pairs. Overall, the average distinctiveness difference is -2.1 percentage points, which can be attributed to the tokenization offset and the near-perfect results of both approaches. Thus, we claim that the CPG approach is **equally suitable** to detect insertion attacks as the JPlag approach with token sequence normalization.

7.5.2.2. LLM Obfuscation Attacks

Figure 7.3 shows the result of running JPlag with optimizations and the full CPG approach with all transformations and reordering on the data sets plagiarized with LLM-based obfuscation attacks.



— Originals vs. Originals (OO) —				
Approach	**Mean**	**Median**	**IQR**	**Q3**
JPlag-N	0.0700	0.0619	0.0716	0.1015
CPG-N	0.0928	0.0846	0.0862	0.1309
Diff	0.0228	0.0225	0.0579	0.0535

— Originals vs. Plagiarisms (OP) —				
Approach	**Mean**	**Median**	**IQR**	**Q3**
JPlag-N	0.4485	0.4416	0.6426	0.7632
CPG-N	0.4733	0.4625	0.6318	0.7831
Diff	0.0248	0.0194	0.0862	0.0682

— Distinctiveness OP vs. OO —			
Approach	**Mean**	**Median**	**Q3**
JPlag-N	0.3785	0.3797	0.6617
CPG-N	0.3804	0.3779	0.6522
Diff	0.0020	-0.0031	0.0147

Figure 7.3.: Evaluation of the approaches “JPlag-N” and “CPG-N” (see Section 7.4) on data sets with LLM-obfuscated plagiarisms.

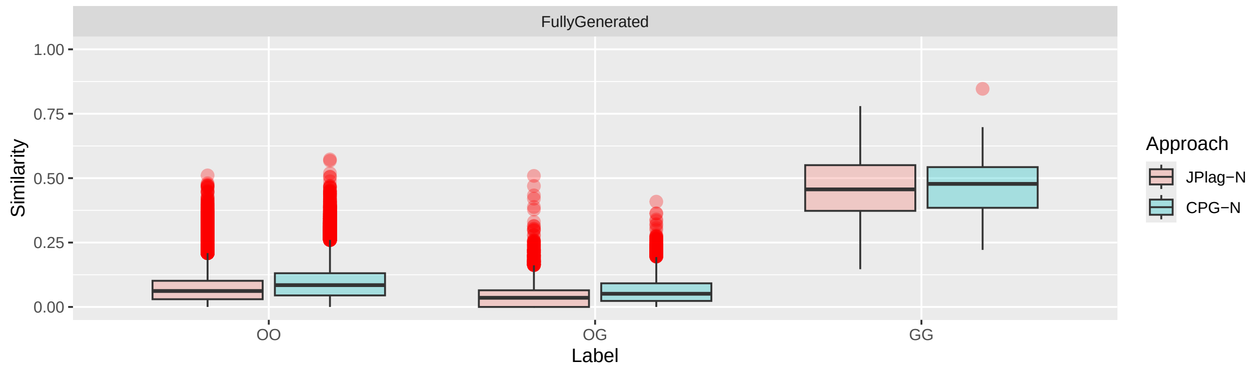
Q.1.2 How does the **normalization** by the CPG approach affect the quality of the plagiarism detection?

A.1.2 The results show that the similarity distributions on all three data sets are very similar between the JPlag and CPG approaches, if we disregard the offset. Overall, with an average distinctiveness difference of 0.2 percentage points, we claim that the CPG approach is **equally suitable** to detect LLM-based obfuscation attacks as the JPlag approach with token sequence normalization.

A more detailed investigation showed that the normalization of neither approach alters the submissions to a substantial degree, which is to be expected as neither of them considers LLM-based obfuscation attacks explicitly in their threat model.

7.5.2.3. LLM Plagiarism Generation Attacks

Figure 7.4 shows the result of running JPlag with optimizations and the full CPG approach with all transformations and reordering on the data sets that contain fully LLM-generated submissions. For this evaluation specifically, we call human submissions *originals* and LLM-generated submissions *generated*, which extends to the labels of pairs.



— Originals vs. Originals (OO) —				
Approach	**Mean**	**Median**	**IQR**	**Q3**
JPlag-N	0.0701	0.0619	0.0716	0.1016
CPG-N	0.0929	0.0847	0.0861	0.1309
Diff	0.0229	0.0226	0.0579	0.0535

— Originals vs. Generated (OG) —				
Approach	**Mean**	**Median**	**IQR**	**Q3**
JPlag-N	0.0422	0.0356	0.0647	0.0647
CPG-N	0.0612	0.0512	0.0683	0.0917
Diff	0.0190	0.0079	0.0427	0.0427

— Generated vs. Generated (GG) —				
Approach	**Mean**	**Median**	**IQR**	**Q3**
JPlag-N	0.4663	0.4564	0.1774	0.5504
CPG-N	0.4672	0.4777	0.1579	0.5428
Diff	0.0009	-0.0076	0.1373	0.0617

— Distinctiveness OG vs. OO —			
Approach	**Mean**	**Median**	**Q3**
JPlag-N	-0.0279	-0.0263	-0.0369
CPG-N	-0.0317	-0.0335	-0.0392
Diff	-0.0038	-0.0147	-0.0108

Figure 7.4.: Evaluation of the approaches “JPlag-N” and “CPG-N” (see Section 7.4) on data sets with LLM-generated plagiarisms.

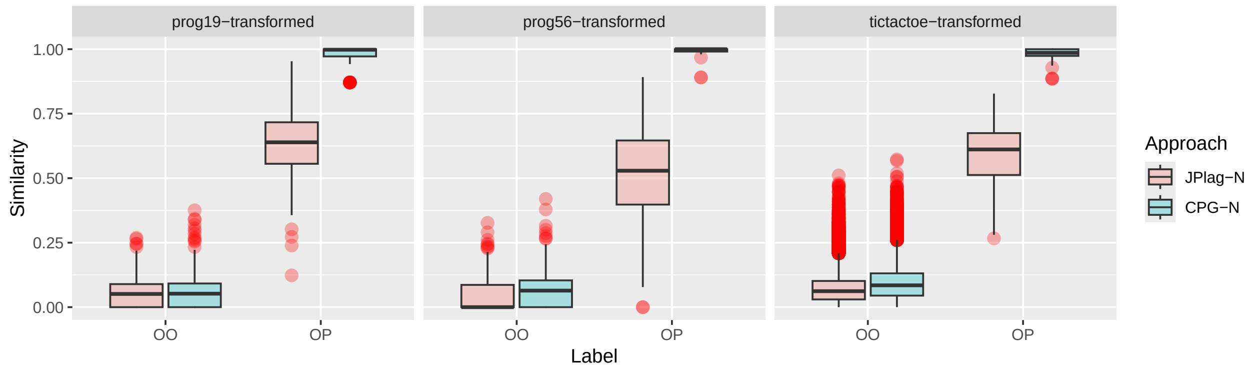
Q.1.2 How does the **normalization** by the CPG approach affect the quality of the plagiarism detection?

A.1.2 On average, *OG* pairs are not distinct from *OO* pairs in either approach. Indeed, *OG* pairs are less similar to each other than *OO* pairs. With a distinctiveness difference of -0.38 percentage points, we claim that the CPG approach is **equally suitable** to detect LLM-generated submissions as the JPlag approach with token sequence normalization.

As noted by Niehues [25], the generated submissions are remarkably similar to each other. For this particular data set, the average *GG* pair would rank among the most similar pairs of *OO* and *OG* pairs. Again, this is true for both approaches.

7.5.2.4. Refactoring Attacks

Figure 7.5 shows the result of running JPlag with optimization and the CPG approach with all transformations and reordering on the data sets plagiarized through automatic refactoring attacks.



— Originals vs. Originals (OO) —				
Approach	**Mean**	**Median**	**IQR**	**Q3**
JPlag-N	0.0700	0.0618	0.0716	0.1015
CPG-N	0.0928	0.0846	0.0862	0.1308
Diff	0.0228	0.0225	0.0579	0.0535

— Originals vs. Plagiarisms (OP) —				
Approach	**Mean**	**Median**	**IQR**	**Q3**
JPlag-N	0.5786	0.5956	0.2008	0.6838
CPG-N	0.9854	1.0000	0.0190	1.0000
Diff	0.4068	0.3919	0.1949	0.4940

— Distinctiveness OP vs. OO —			
Approach	**Mean**	**Median**	**Q3**
JPlag-N	0.5086	0.5338	0.5823
CPG-N	0.8925	0.9154	0.8692
Diff	0.3840	0.3694	0.4405

Figure 7.5.: Evaluation of the approaches “JPlag-N” and “CPG-N” (see Section 7.4) on data sets with plagiarisms generated through automatic refactoring transformations.

Q.1.2 How does the **normalization** by the CPG approach affect the quality of the plagiarism detection?

A.1.2 With a median similarity of 100 percent and mean similarity of more than 98 percent between originals and plagiarisms, the results show that the CPG approach is able to revert nearly all transformations used to create the plagiarisms and, in most cases (227 of 450), restore the exact token sequence of the normalized original. The average distinctiveness difference of the CPG approach is 38 percentage points greater than that of JPlag with normalization enabled, which of course was not designed to counter refactoring attacks. Yet again, the offset of two percentage points shows in the similarity between original pairs.

Note that the distinctiveness difference between the approaches can be arbitrarily increased by creating even more obfuscated plagiarisms by applying the same transformation to the submission more times, and that the plagiarism generator was designed specifically to use only precisely those transformations that the CPG approach would be able to counter. In real use cases, we expect that plagiarisms use combined attacks, not all of which will be covered by the implemented transformations.

While the CPG approach transforms every match that it can find, there are multiple constellations that may cause some matches to be left untransformed.

- To limit runtime, the CPG is matched against the source graph pattern of each transformation one or two times only. As transformations may result in new matches for other transformations, some matches may be left untransformed. The order of the transformations was chosen carefully so that untransformed matches should occur as an exception, but this remains a vulnerability of the implementation.
- In the course of the *transformed* data sets generation, PlagGen uses simple rules to determine where a refactoring may be applied. In some cases, these refactorings may turn out to inadvertently change the semantic, in which case the CPG approach will deliberately not transform them back.
- As the CPG library is still relatively new, there are still some edge cases where inconsistencies occur, which essentially makes the plagiarism generator more difficult to beat.

Despite these challenges, the implementation was able to counter the generated attacks.

7.5.3. Nonfunctional Properties

Runtime. Figure 7.6 shows the runtime of the approaches on the different data sets in milliseconds. These measurements are taken before and after the call to the approaches, respectively. Overall, the median of the CPG approach with transformations and reordering runtime is between 86 and 257 percent of the runtime of JPlag, which may amount to a difference of just several minutes for a large submission set.

Looking into the results in more detail, the runtime of the CPG approach shows to be highly dependent on the structural complexity of the code. On data sets with linear, short code, it may even outperform the AST-based approach. On more complex data sets, however, the CPG approach is slower than JPlag on average, and extreme outliers occur in the CPG approach which take more than two orders of magnitude more time than the upper quartile. An investigation into these outliers shows that the DFG analysis needed for the linearization on block level (Section 5.8) is likely to blame, as its performance depends heavily on the number of paths through a method. As some methods in the tictactoe data sets reached a cyclomatic complexity of more than 150[1], the runtime of the DFG analysis becomes unreasonable. Thus, an option was added to skip the DFG analysis of a method if its cyclomatic complexity exceeds 120, which is still an excessive value. Being a fixpoint analysis on graphs, the DFG analysis surely leaves room for further optimization.

[1]To put this value into relation: Watson and McCabe suggest setting a limit cyclomatic complexity of 10 as a code quality requirement and claim that this value has "significant supporting evidence." [45, p. 15]

Note also that there is a considerable initialization runtime overhead for both approaches, which was eliminated by running them on a small sample before the proper evaluation run.

Impact of the Transformations on the Runtime. To evaluate the runtime of the transformations, the evaluation procedure was repeated with the DFG analysis and reordering deactivated.

Q.2.2 How does the application of the transformations affect the runtime of the CPG approach?

A.2.2 The results showed that in this configuration, the transformations caused a runtime overhead of 15 to 25 percent. This is a small overhead compared to the DFG analysis. On all data sets except the *tictactoe* data sets, the CPG approach without DFG analyses is faster on average than the AST-based approach.

Reliability. During the evaluation runs, failures of the approaches were recorded to evaluate the reliability of the CPG approach.

Q.2.3 How reliable is the CPG approach?

A.2.3 All runtime failures could be eliminated. However, the reliability of the CPG approach depends on whether all language features used in the submissions are covered in each step of the pipeline. As no exhaustive test has been run, it would not be accurate to claim perfect reliability for the CPG approach.

– prog19-insert –

Approach	Mean	Min	Q1	Median	Q3	Max	Sum	Failures
Java	110	98	106	109	112	137	5923	0
Java-N	100	93	96	97	98	243	5390	0
CPG	89	34	55	74	100	279	4813	0
CPG-N	183	61	103	141	189	985	9855	0

– prog19-obfuscated –

Approach	Mean	Min	Q1	Median	Q3	Max	Sum	Failures
Java	104	94	98	103	107	136	10656	0
Java-N	99	93	95	97	102	118	10096	0
CPG	95	18	64	81	108	363	9683	0
CPG-N	136	30	82	117	161	655	13779	0

– prog19-transformed –

Approach	Mean	Min	Q1	Median	Q3	Max	Sum	Failures
Java	113	100	107	110	114	246	20003	0
Java-N	103	93	98	100	105	148	18182	0
CPG	76	33	60	75	90	187	13396	0
CPG-N	146	59	112	132	160	705	25787	0

– prog56-insert –

Approach	Mean	Min	Q1	Median	Q3	Max	Sum	Failures
Java	107	101	103	106	108	136	5964	0
Java-N	110	94	101	103	106	272	6144	0
CPG	62	26	45	53	65	250	3483	0
CPG-N	120	47	85	107	130	451	6745	0

– prog56-obfuscated –

Approach	Mean	Min	Q1	Median	Q3	Max	Sum	Failures
Java	99	94	96	97	99	146	10238	0
Java-N	97	92	94	96	97	120	10007	0
CPG	62	26	48	55	73	240	6366	0
CPG-N	92	46	73	83	104	333	9498	0

– prog56-transformed –

Approach	Mean	Min	Q1	Median	Q3	Max	Sum	Failures
Java	111	97	105	109	113	203	18706	0
Java-N	104	94	97	98	101	293	17434	0
CPG	74	25	47	62	74	403	12459	0
CPG-N	132	52	96	108	126	415	22234	0

– tictactoe-insert –

Approach	Mean	Min	Q1	Median	Q3	Max	Sum	Failures
Java	124	97	116	121	128	225	84222	0
Java-N	105	94	100	102	106	251	71329	0
CPG	146	24	98	125	166	2892	99137	0
CPG-N	377	40	178	242	367	5246	255935	0

– tictactoe-obfuscated –

Approach	Mean	Min	Q1	Median	Q3	Max	Sum	Failures
Java	102	94	98	100	104	141	68394	0
Java-N	99	92	96	98	100	164	66745	0
CPG	137	25	94	122	158	1925	92214	0
CPG-N	305	32	157	211	311	4470	204422	0

– tictactoe-transformed –

Approach	Mean	Min	Q1	Median	Q3	Max	Sum	Failures
Java	126	95	116	122	132	207	97742	0
Java-N	101	94	98	100	102	210	78732	0
CPG	166	31	104	135	187	2799	128655	0
CPG-N	402	41	186	257	391	5385	311921	0

– FullyGenerated –

Approach	Mean	Min	Q1	Median	Q3	Max	Sum	Failures
Java	109	103	104	106	108	138	1090	0
Java-N	99	94	96	100	103	104	993	0
CPG	59	53	54	57	63	72	592	0
CPG-N	124	105	113	125	128	157	1239	0

Figure 7.6.: Statistics of the runtime in milliseconds, and number of failures during the evaluation.

7.6. Discussion

In this section, we summarize what we have learned from the evaluation.

Results of the Evaluation. The results of the evaluation suggest that the CPG approach does improve the quality of plagiarism detection. While it shows very similar results to JPlag with token sequence normalization for insertion, reordering, LLM-obfuscated and LLM-generated attacks (albeit with a relatively stable offset of two to three percentage points), on refactoring attacks, it outperforms JPlag by 38 percentage points on average. As discussed before, this considerable distinction value could be increased arbitrarily by transforming the original even more; likewise, it decreases if transformation attacks are included that the CPG transformation system cannot deal with yet.

The increase in plagiarism detection quality comes at the cost of a (median per-submission) runtime of up to 250 percent of JPlag. Submissions with exceedingly complex control structures may take more than 100 times longer than the average submission runtime, but a limit of the cyclomatic complexity can be adjusted accordingly, so that any submission exceeding that limit can be reviewed manually.

While no failures occurred during the final run of the evaluation, we cannot guarantee perfect reliability. The correct function of the approach depends on covering all kinds of different language features, some of which may not have appeared in any of the data sets used for testing and during the evaluation.

Automatic Generation of Refactoring Plagiarisms. During the implementation of PlagGen, it became apparent that it is equally challenging to create plagiarisms that preserve the semantic of the original as it is to preserve the semantic in the plagiarism detection transformations. For example, an expression cannot be extracted to a variable if it is the case expression of a case statement, an assignment, the target of an assignment, the condition of a loop statement, a `null` reference, among many other cases. Similarly, an expression cannot be extracted into a constant if it contains a method call, a variable reference, or if it is a type reference. As a last interesting example, the transformation *Wrap in Optional* was used on a variable that was later the target of a shorthand assignment operation, where it is read *and* written. The transformation could only be applied after the shorthand assignment was transformed into the long form, where the reading and writing accesses to the variable are separate.

7.7. Threats to Validity

This section discusses how the validity of the results could have been compromised, and how these threats have been dealt with.

7.7.1. Threats to Internal Validity

Test data generation. The *transformed* data sets, which incorporate plagiarisms created through refactoring attacks, were all generated by the automatic plagiarism generation tool PlagGen. If the data sets would have been created by the same mechanism that is used to detect the plagiarism, this could have introduced a bias. To avoid this, PlagGen was designed deliberately using different libraries to apply refactorings, so that no part of the CPG transformation system could be reused. The original submissions that were selected to create plagiarisms from were determined by a random number generator, so no bias could arise from the selection.

Selection of existing data sets. Other than the *transformed* data sets, the *insert*, *obfuscated* and *FullyGenerated* data sets were used, as they were used by Sağlam et al. [38] and Niehues [25], which ensured that the data sets are suitable for such an evaluation and that the results are comparable between the different approaches.

Influence of the rest of the pipeline. Apart from the transformations, the other steps of the pipeline may have also affected the similarity rating. However, it was demonstrated that the CPG linearization only introduced an offset in the range of two to three percentage points to the similarity rating, which should not affect how clearly pairs of originals and plagiarisms could be distinguished from pairs of unrelated originals. The multiple reordering mechanisms clearly do influence the similarity rating. However, as we compared the CPG approach to JPlag with normalization, which also applies reordering, we still determined the effect of the transformations accurately. The token selection also may influence the similarity rating. To mitigate the risk of bias from the token selection, the token selection of JPlag for Java was closely replicated.

7.7.2. Threats to External Validity

Scale of the effect. As a concept, a CPG transformation system can increase the quality of plagiarism detection *if* refactoring attacks were used that are covered in the current transformation selection. In practice, where instead of pure refactoring attacks, combined attacks are more likely to be used, and refactorings may be used that are currently not selected, the distinctiveness of plagiarisms among originals may not be as high as in the evaluation.

Language independence. The present approach was only evaluated on submissions in Java; thus, we cannot claim that the results transfer to other languages. This is despite the fact that the CPG as an intermediary representation accepts other languages such as

C++, Python, or Ruby; also, the transformation engine should generally work on other languages. However, some of the transformations themselves may not be valid in other languages, especially if they refer to specific API elements. For these language-specific transformations, code in another language will never match, which is detrimental to the effectiveness of the transformation selection. For example, a transformation may expect that a method declaration is located inside a class declaration, as is usually the case in Java. In Python and C++, this is not a requirement.

It remains to be examined in future research whether the approach is equally effective in other programming languages.

Plagiarism Detection Tool. While the tokenization and interface of the CPG pipeline were designed to work with JPlag, the core part of the approach works independently from the chosen plagiarism detection tool. Thus, other token-based plagiarism detection tools should be able to use an adaptation of its implementation with little effort.

7.7.3. Threats to Construct Validity

Approach of the evaluation. The evaluation was planned and conducted according to the Goal Question Metric approach [2]. This ensures that the metrics presented in the evaluation are not selected or adapted at will once the evaluation is carried out with the intention to favorably manipulate the results.

Meaningful metrics. The method of comparing similarity ratings between labeled originals (false positives) and originals vs. plagiarisms (true positives) has been used throughout the research on code plagiarism and its detection, for example, by Devore-McDonald [10], Sağlam [40], Krieg [20], Brödel [38], and Niehues [25].

7.7.4. Threats to Reliability

Reproducibility of the results. A reproduction package was published [23] to ensure that the results of the evaluation are reproducible. It includes the following:

- JPlag, including an implementation of the CPG approach (code base and runnable JAR file),
- The plagiarized *prog19*, *prog56*, and *FullyGenerated* data sets used for the evaluation,
- The complete results in CSV format,
- The R script used to process the results, and all graphics and tables that it generated.

The *tictactoe* data sets could not be published due to data protection restrictions.

8. Limitations and Future Work

Several factors still limit the effectiveness of the present approach. In this chapter, potential extensions to the CPG approach are discussed that may be investigated and evaluated in further research to address these limitations.

Language Independence. Due to time constraints, the CPG approach was evaluated only on submissions written in Java. Still, it can parse submissions in various other languages, such as C++, Python, and Ruby. This raises multiple questions:

- Are the transformations of the current CPG approach effective on submissions in other languages?
- Which transformations could be added to the CPG approach to make it similarly effective on input in other languages as for input in Java?
- Is the CPG approach effective in detecting plagiarisms across programming languages?

This research requires suitable testing data in the respective programming languages.

Limitations of Static Analysis. It is known from Rice's theorem [35] that static analysis can ultimately only ever approximate the runtime behavior of a program. Plagiarists may find ways to exploit this fact by designing refactoring attacks in such a way that excessive analysis will be required to counter them, although we can assume that such an attack takes considerable effort and knowledge if done manually.

A form of analysis that particularly addresses properties of runtime behavior using static analysis is *abstract interpretation* [8]. The insights gained from abstract interpretation could be used to eliminate dead control flow statements, where the condition is not a constant expression, but still constantly `true` or `false` at runtime.

Limitations by the Selection of Transformations. Due to its special-purpose nature, the CPG approach is only able to counter the precise refactorings that were considered during its design. Any other refactoring may still prove to be quite an effective attack against the CPG approach, at least to a similar extent that it would be against state-of-the-art approaches. This is the case with the AI-generated attacks: as the CPG approach yields no significant increase in similarity compared to JPlag with the token sequence normalization approach, it is safe to assume that ChatGPT uses refactorings which are not yet covered by the current transformation selection.

Thus, we can increase the effectiveness of the approach by extending the catalogue of refactorings further.

Limitations by the Conservative Nature of the Refactorings. During the conception of the approach, a great concern was that the transformations might alter the submissions in such an aggressive manner that the false-positive rate increases to an extent which would render the approach ineffective. With this in mind, the CPG approach was designed specifically so that transformations would not change the semantics of the submitted code (except for common code removal). This means that if a plagiarist changes the semantics by even so little as to move a variable declaration in front of its parent block, this refactoring cannot be reverted by the current CPG approach. A variant of this attack, *Declare all variables at the beginning of source code*, is described by Karnalim [18, p. 65].

In this specific case, a specialized analysis could determine the latest position in the method where a variable declaration could be placed (e.g., in the innermost block that contains all variable usages, immediately before the statement or block that contains the first reference to the variable). Other similar cases should also be investigated.

API Semantics. Structural elements of code have a well-defined semantic, so we can easily transform equivalent representations of code to each other using different elements. However, if the inner workings are hidden behind API calls, then a structural analysis gives little insight about the behavior of the code and how the code might be represented in more basic terms. As was done with the `Optional` class, we could deal with more API elements through specialized transformations.

9. Conclusion

When students plagiarize code by other authors and try to obfuscate the plagiarism, these obfuscation attacks can often be interpreted as sequences of refactorings. In this thesis, we examined an approach to make token-based plagiarism detection resilient against refactoring attacks by transforming the submissions to a normalized structure.

To this end, a CPG transformation system was implemented that calculates the atomic operations on the CPG necessary to conduct a transformation, finds subgraphs that are isomorphic to the source graph pattern of a transformation, and applies these operations while preserving the invariants of the graph, as well as the semantic of the code. The transformed CPG is linearized and tokenized to a token list, which can be compared effectively by algorithms like Greedy String Tiling with Karp-Rabin Matching.

More than a dozen transformations were implemented, covering a wide variety of refactorings, including, but not limited to the insertion of *common code* and dead code, the extraction of new variables and constants, the formation of constant classes, the equivalent replacement of control statements, and the negation of `if-then-else` conditions.

The CPG approach was evaluated on data sets that contained plagiarisms obfuscated by insertion attacks or refactoring attacks, as well as submissions created by an LLM. The results suggest that the CPG approach shows effectiveness on insertion attacks and LLM-based attacks on-par to JPlag with token sequence normalization enabled, and near-perfect effectiveness on the refactoring attacks that it was designed to counter. As we will learn more from future research about the refactorings which occur in manually and automatically generated plagiarisms, we are confident that the CPG approach will continue to be extended and is only just at the beginning of its potential resilience.

Bibliography

[1] Brenda S. Baker. “On Finding Duplication and Near-Duplication in Large Software Systems”. In: *Proceedings of 2nd Working Conference on Reverse Engineering*. 1995, pp. 86–95. DOI: 10.1109/WCRE.1995.514697.

[2] Victor R. Basili, Gianluigi Caldiera, and Hans Dieter Rombach. “The Goal Question Metric Approach”. In: *Encyclopedia of Software Engineering* (2002). Ed. by Rini van Solingen. DOI: https://doi.org/10.1002/0471028959.sof142.

[3] Moritz Brödel. “Preventing Automatic Code Plagiarism Generation Through Token String Normalization”. BA thesis. Department of Informatics, Karlsruhe Institute of Technology, Apr. 2023. DOI: 10.5445/IR/1000165371.

[4] Dong-Kyu Chae et al. “Software Plagiarism Detection: A Graph-based Approach”. In: *CIKM ’13. Proceedings of the 22nd ACM international conference on Information & Knowledge Management* (Oct. 27, 2013–Nov. 1, 2023). San Francisco, CA, USA, Oct. 2013, pp. 1577–1580. DOI: 10.1145/2505515.2507848.

[5] James R. Cordy et al. “Source Transformation in Software Engineering Using the TXL Transformation System”. In: *Information and Software Technology* 44.13 (2002), pp. 827–837. DOI: 10.1016/S0950-5849(02)00104-0.

[6] Georgina Cosma. “An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis”. PhD thesis. Department of Computer Science, University of Warwick, July 2008. URL: http://wrap.warwick.ac.uk/3575/.

[7] Georgina Cosma and Giovanni Acampora. “A Fuzzy-based Approach to Programming Language Independent Source-Code Plagiarism Detection”. In: *FUZZ-IEEE. 2015 IEEE International Conference on Fuzzy Systems*. 2015. DOI: 10.1109/FUZZ-IEEE.2015.7337935.

[8] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[9] Neil Davey et al. “The Development of a Software Clone Detector”. In: *International Journal of Applied Software Technology* 1.3/4 (1995), pp. 219–236. URL: http://hdl.handle.net/2299/617.

[10] Breanna Devore-McDonald and Emery D. Berger. “MOSSAD: Defeating Software Plagiarism Detection”. In: *ACM Programming Languages* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428206.

[11] Martin Dick et al. “Addressing Student Cheating: Definitions and Solutions”. In: *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*. ITiCSE-WGR ’02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 172–184. DOI: 10.1145/960568.783000.

[12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and its Use in Optimization”. In: *ACM Transactions on Programming Languages and Systems*. Vol. 9. 1987, pp. 319–349. DOI: 10.1145/24039.24041.

[13] Jinan A.W. Fiaidhi and Scott K. Robinson. “An Empirical Approach for Detecting Program Similarity and Plagiarism Within a University Programming Environment”. In: *Computers & Education* 11.1 (1987), pp. 11–19. DOI: 10.1016/0360-1315(87)90042-X.

[14] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. 2nd ed. with contributions by Kent Beck. Pearson Education, Inc., 2019. ISBN: 978-0-13-475759-9.

[15] Scott Grant and James R. Cordy. “An Interactive Interface for Refactoring Using Source Transformation”. In: *REFACE ’03. 1st international Workshop on Refactoring: Achievements, Challenges, Effects*. 2003, pp. 30–33. URL: https://research.cs.queensu.ca/home/cordy/Papers/REFACE-WCRE03-RUST.pdf.

[16] Niklas R. Heneka. “Software Plagiarism Detection on Intermediate Representation”. BA thesis. Department of Informatics, Karlsruhe Institute of Technology, Oct. 2023. DOI: 10.5445/IR/1000168422.

[17] Susan Horwitz, Thomas Reps, and David Binkley. “Interprocedural Slicing Using Dependence Graphs”. In: *ACM Transactions on Programming Languages and Systems* 12.1 (Jan. 1990), pp. 26–60. DOI: 10.1145/77606.77608.

[18] Oscar Karnalim. “Detecting Source Code Plagiarism on Introductory Programming Course Assignments Using a Bytecode Approach”. In: *2016 International Conference on Information & Communication Technology and Systems (ICTS)* (Oct. 12, 2016). 2016, pp. 63–68. DOI: 10.1109/ICTS.2016.7910274.

[19] Sangujun Ko, Jusop Choi, and Hyoungshick Kim. “COAT: Code ObfuscAtion Tool to evaluate the performance of code plagiarism detection tools”. In: *2017 International Conference on Software Security and Assurance (ICSSA)*. 2017, pp. 32–37. DOI: 10.1109/ICSSA.2017.29.

[20] Pascal Krieg. “Preventing Code Insertion Attacks on Token-Based Software Plagiarism Detectors”. BA thesis. Department of Informatics, Karlsruhe Institute of Technology, Sept. 2022. DOI: 10.5445/IR/1000154301.

[21] Chao Liu et al. “GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis”. In: *KDD ’06. Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Aug. 20–23, 2006). Philadelphia, PA, USA, 2006, pp. 872–881. DOI: 10.1145/1150402.1150522.

[22] Rien Maertens et al. “Dolos: Language-Agnostic Plagiarism Detection in Source Code”. In: *Journal of Computer Assisted Learning* 38.4 (2022), pp. 1046–1061. DOI: 10.1111/jcal.12662.

[23] Robin Maisch. *Reproduction Package for "Preventing Refactoring Attacks on Software Plagiarism Detection through Graph-Based Structural Normalization"*. Version 1.0. Zenodo, May 2024. DOI: `10.5281/zenodo.11182830`. URL: `https://doi.org/10.5281/zenodo.11182830`.

[24] Tom Mens et al. “Formalizing Refactorings with Graph Transformations”. In: *Journal of Software Maintenance and Evolution: Research and Practice* 17.4 (2005), pp. 247–276. DOI: `10.1002/smr.316`.

[25] Nils Niehues. “Intelligent Match Merging to Prevent Obfuscation Attacks on Software Plagiarism Detectors”. MA thesis. Department of Informatics, Karlsruhe Institute of Technology, Nov. 2023. DOI: `10.5445/IR/1000167446`.

[26] Nils Niehues. *Reproduction package for: Intelligent Match Merging to Prevent Obfuscation Attacks on Software Plagiarism Detectors.* Version v1. Nov. 2023. DOI: `10.5281/zenodo.10149535`.

[27] Matija Novak. “Effect of Source-Code Preprocessing Techniques on Plagiarism Detection Accuracy in Student Programming Assignments”. PhD thesis. Faculty of Organization and Informatics, University of Zagreb, 2020. URL: `https://repozitorij.unizg.hr/islandora/object/foi:4787`.

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

[29] Matija Novak, Mike Joy, and Dragutin Kermek. “Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review”. In: *ACM Transactions on Computing Education* 19.3 (May 2019). DOI: `10.1145/3313290`.

[30] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. *PROGpedia.* Version 1.0.1. Dec. 2022. DOI: `10.5281/zenodo.7449056`.

[31] Junhyun Park et al. “An Efficient Technique of Detecting Program Plagiarism Through Program Slicing”. In: *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing.* Vol. 790. Springer International Publishing, 2019. Chap. 13, pp. 164–175. DOI: `10.1007/978-3-319-98367-7_13`.

[32] Renaud Pawlak et al. “Spoon: A Library for Implementing Analyses and Transformations of Java Source Code”. In: *Software: Practice and Experience* 46 (2015), pp. 1155–1179. DOI: `10.1002/spe.2346`. URL: `https://hal.archives-ouvertes.fr/hal-01078532/document`.

[33] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. “JPlag: Finding plagiarisms among a set of programs”. In: *Journal of Universal Computer Science* 8.11 (Mar. 2002), pp. 1016–1038. DOI: `10.3217/JUCS-008-11-1016`.

[34] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. “Software Clone Detection: A systematic review”. In: *Information and Software Technology* 55.7 (2013), pp. 1165–1199. DOI: `10.1016/j.infsof.2013.01.008`.

[35] Henry G. Rice. “Classes of Recursively Enumerable Sets and Their Decision Problems”. In: *Transactions of the American Mathematical Society.* Vol. 74. 2. American Mathematical Society, 1953, pp. 358–366. DOI: `10.2307/1990888`.

[36] Chanchal Kumar Roy and James R. Cordy. *A Survey on Software Clone Detection Research.* Tech. rep. 2007-541. Ontario, Canada: School of Computing, Queen’s University at Kingston, Sept. 2007. URL: `http://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf`.

[37] Timur Sağlam et al. “Automated Detection of AI-Obfuscated Plagiarism in Modeling Assignments”. In: *ICSE-SEET ’24. 46th International Conference on Software Engineering: Software Engineering Education and Training.* 131. ACM, Apr. 2024. DOI: `10.1145/3597503.3639192`.

[38] Timur Sağlam et al. “Detecting Automatic Software Plagiarism via Token Sequence Normalization”. In: *ICSE ’24. Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.* 131. Apr. 2024. DOI: `10.1145/3597503.3639192`.

[39] Timur Sağlam et al. *Supplementary Material for "Detecting Automatic Software Plagiarism via Token Sequence Normalization".* Version v1. Dec. 2023. DOI: `10.5281/zenodo.10430322`.

[40] Timur Sağlam et al. “Token-Based Plagiarism Detection for Metamodels”. In: *MODELS ’22. Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings.* Oct. 2022, pp. 138–141. DOI: `10.1145/3550356.3556508`.

[41] Saul David Schleimer, Daniel Shawcross Wilkerson, and Alex Aiken. “Winnowing: Local Algorithms for Document Fingerprinting”. In: *2003 ACM International conference on Management of Data* (2003), pp. 76–85. DOI: `10.1145/872757.872770`.

[42] Simon et al. “Choosing Code Segments of Exclude from Code Similarity Detection”. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education.* June 2020. DOI: `10.1145/3437800.3439201`.

[43] Simon et al. “Negotiating the maze of academic integrity in computing education”. In: *Proceedings of the 2016 ITiCSE working group reports.* 2016, pp. 57–80. DOI: `10.1145/3024906.3024910`.

[44] Jonas Strittmatter. “Token-based Plagiarism Detection for Statecharts”. BA thesis. Department of Informatics, Karlsruhe Institute of Technology, Apr. 2023. DOI: `10.5445/IR/1000165276`.

[45] Arthur H. Watson and Thomas J. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. NIST Special Publication 500-235.* Computer Systems Laboratory, National Institute of Standards and Technology, 1996. URL: `http://www.mccabe.com/pdf/mccabe-nist235r.pdf`.

[46] Mark Weiser. “Program Slicing”. In: *IEEE Transactions on Software Engineering* SE-10.4 (July 1984). DOI: `10.1109/TSE.1984.5010248`.

[47] Michael J. Wise. “String Similarity via Greedy String Tiling and Running Karp-Rabin Matching”. Department of Computer Science, University of Sydney. Dec. 1993. URL: https://www.researchgate.net/profile/Michael_Wise/publication/262763983_String_Similarity_via_Greedy_String_Tiling_and_Running_Karp-Rabin_Matching/links/59f03226aca272a2500141f4/String-Similarity-via-Greedy-String-Tiling-and-Running-Karp-Rabin-Matching.pdf.

[48] Fabian Yamaguchi et al. “Modeling and Discovering Vulnerabilities with Code Property Graphs”. In: *2014 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA, 2014, pp. 590–604. DOI: 10.1109/SP.2014.44.

A. Appendix

A.1. Isomorphism Detection Algorithm – Pseudo Code

Algorithm 2 CPG Isomorphism Detection Algorithm

```
function COMPARE(node, pattern, matches)
    finished ← $\{match \in$ matches $\mid match[$pattern$] =$ node$\}$
    invalid ← $\{match \in$ matches $\mid match[$pattern$] \neq$ node$\}$
    open ← $\{match \in$ matches $\mid$ pattern $\notin match.keys\}$

    if node violates local properties specified by pattern then
        return finished                          ▹ No new matches from this comparison
    end if

    open ← $\{match[$pattern $\mapsto$ node$] \mid match \in$ open$\}$
                                                 ▹ Map pattern to node in all open matches

    for all relations $r$ specified by pattern do
        if $r$ is a simple (1:1) relation then
            if node has a related node $related$ via $r$ then
                open ← COMPARE($related$, $r.pattern$, open)
            else
                open ← $\emptyset$                      ▹ No new matches from this comparison
            end if

        else if $r$ is an existential (1-of-n) relation then
            $newMatches$ ← $\emptyset$
            for all related nodes $related$ of node via $r$ do
                $newMatches$ ← COMPARE($related$, $r.pattern$, open) ∪ newMatches
            end for
            open ← $newMatches$
                                       ▹ open is now non-empty iff any related node matches

        else if r is a universal (n-of-n) relation then
            for all related nodes $related$ of node via $r$ do
                open ← COMPARE($related$, $r.pattern$, open)
            end for
                                       ▹ open is now non-empty iff all related nodes match
        end if
    end for

    for all match properties $p$ specified in pattern do
        open ← $\{match \in$ open $\mid match$ satisfies $p\}$
    end for

    return finished ∪ open
end function
```