Karlsruhe Institute of Technology

Software Design and Quality

Preventing Code Insertion Attacks on Token-Based Software Plagiarism Detectors

Bachelor's Thesis of

Pascal Krieg

at the Department of Informatics
KASTEL – Institute of Information Security and Dependability

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Timur Sağlam
Second advisor:	M.Sc. Sebastian Hahner

26. May 2022 – 26. September 2022

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

This document is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0): https://creativecommons.org/licenses/by/4.0/deed.en

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, Semptember 26th, 2022

. . . .

(Pascal Krieg)

Abstract

Some students tasked with mandatory programming assignments lack the time or dedication to solve the assignment themselves. Instead, they might plagiarize a solution by slightly modifying their peer's code. To counteract this, there exist numerous tools that assist in detecting this kind of plagiarism. These tools can be used by instructors to identify plagiarized programs among the set of all submissions. The most used type of plagiarism detection tool is token-based plagiarism detectors. They are resilient against many types of obfuscation attacks, such as renaming variables or whitespace modifications. However, they are susceptible to the insertion of code that does not affect the program flow or result.

The current working assumption was that the successful obfuscation of plagiarism takes more effort and skill than solving the assignment itself. This assumption was broken by automated plagiarism generators, which exploit this weakness. This work aims to develop mechanisms against code insertions that can be directly integrated into existing token-based plagiarism detectors. For this, we first develop mechanisms to negate the negative effect of many types of code insertion. Then we implement these mechanisms prototypically into a state-of-the-art plagiarism detector. We evaluate our implementation by running it on a dataset consisting of real student submissions and automatically generated plagiarism. We show that with our mechanisms, the similarity rating of automatically generated plagiarism increases drastically. Consequently, the plagiarism generator we use fails to create usable plagiarisms.

Zusammenfassung

Manchen Studierenden fehlt die Zeit oder Arbeitsbereitschaft, ihre Programmieraufgaben selbst zu lösen. Stattdessen plagiieren sie Abgaben ihre Kommilitonen, indem sie deren Code übernehmen und leicht verändern. Um dem vorzubeugen, existieren Programme, die beim Finden von Plagiaten unterstützen. Die geläufigste Art dieser Plagiatserkenner sind Token-basierte Plagiatserkenner. Diese sind resistent gegen viele Verschleierungsversuche, wie beispielsweise Variablenumbenennungen oder Umformatierung des Quelltextes. Sie sind jedoch generell anfällig gegen das Einfügen von Codezeilen, die den Programmfluss und das Ergebnis nicht beeinflussen.

Die bisherige Annahme war, dass das erfolgreiche Verschleiern von Plagiaten mehr Zeit und Können erfordert, als die Aufgabe selbst zu lösen. Automatisierte Plagiatsgeneratoren brechen diese Annahme, indem sie die Anfälligkeit gegen Codeeinfügungen ausnutzten, um automatisiert Plagiate zu erstellen. Ziel dieser Arbeit ist es, Mechanismen zu finden, die in bestehende Token-basierte Plagiatserkenner integriert werden können, um die Resilienz gegen Codeeinfügung zu verbessern. Dazu entwerfen wir zunächst Mechanismen, die den negativen Effekt vieler Codeeinfügungen auf die Plagiatserkennung reduzieren können. Anschließend implementieren wir diese Mechanismen prototypisch in einem modernen Token-basierte Plagiatserkenner. Wir evaluieren unsere Implementierung anhand eines Datensatzes aus echten Programmierabgaben und aus Plagiaten, die wir automatisch generiert haben. Damit zeigen wir, dass die Verwendung unserer Mechanismen die gemessene Ähnlichkeit automatisch generierter Plagiate stark erhöht. Dadurch ist der von uns verwendete Plagiatsgenerator nicht mehr in der Lage, verwendbare Plagiate zu erstellen.

Contents

Abstract i

Zusammenfassung iii

1. Introduction 1

2. Foundations 3
2.1. Plagiarism . . . 3
2.2. Software Plagiarism Detectors . . . 3
2.3. Token-Based Software Plagiarism Detectors . . . 4
2.4. JPlag . . . 5
2.4.1. Tokenization . . . 6
2.4.2. Sequence Comparison . . . 6
2.4.3. Limitations . . . 7

3. Threat Modelling 9
3.1. Automated Code Insertion . . . 9
3.2. Mossad . . . 9
3.2.1. Method of Creating Plagiarism . . . 10
3.2.2. Applicability to JPlag Rather Than MOSS . . . 11
3.3. Threat Model . . . 11
3.4. Role of Manual Inspection for Plagiarism Detection . . . 12

4. Defense Mechanism 15
4.1. Deleting Unused Variables . . . 15
4.2. Deleting Tokens in Unreachable Code . . . 16
4.2.1. Naive State Machine . . . 17
4.2.2. Limitations and Addition for Scanned Languages . . . 18
4.3. Directly Reverting Token Insertions . . . 20
4.3.1. Naive Algorithm . . . 21
4.3.2. Key Considerations . . . 22

5. Implementation 23
5.1. Implementing the Approaches in JPlag . . . 23
5.1.1. C++ Scanner Extensions . . . 23
5.1.2. Core Algorithm Extension . . . 25
5.2. Framework for Automatically Evaluating JPlag . . . 26
5.2.1. Configuration . . . 27
5.2.2. Running the Framework . . . 27

6. Evaluation **29**
6.1. Methodology . . . 29
6.1.1. Dataset . . . 29
6.1.2. GQM Plan . . . 30
6.2. Effect on Plagiarism Detection . . . 32
6.2.1. Effect of Using All Mechanisms . . . 32
6.2.2. Impact of the Specific Mechanisms . . . 33
6.2.3. Generic Token Filtering Impact on Non-Plagiarisms . . . 36
6.3. Effect on New Plagiarism Generation by Mossad . . . 37
6.4. Threats to Validity . . . 39

7. Related Work **41**

8. Future Work **43**

9. Conclusion **45**

Bibliography **47**

A. Appendix **51**

List of Figures

2.1. Code Insertion Effect on Matches 5
2.2. Internal Structure and Dataflow in JPlag 6

3.1. Schematic Dataflow of Mossad 10

4.1. Unreachable Code Detection State Machine for Parsed Languages 17
4.2. Unreachable Code Detection State Machine for Scanned Languages . . . 19
4.3. Improved Unreachable Code Detection State Machine for Scanned Languages 20
4.4. Basic Concept of Generic Token Filtering Algorithm 21
4.5. Token Example Motivating Window Padding 22

5.1. UML Representation of the State Machine 26

6.1. Plagiarism Similarity Using All Mechanisms 32
6.2. Plagiarism Similarity Using Only Single Defense Mechanisms 33
6.3. Mechanism Contributions after Deleting Unused Variables 34
6.4. Generic Token Filtering Impact Depending on Original Similarity 35
6.5. Similarity Change of Non-plagiarisms Depending on LOC 36
6.6. Mossad Iterations Required for Plagiarism Creation 38
6.7. Mossad Execution Time for Plagiarism Creation 38

List of Tables

6.1. GQM-Plan 31
A.1. Dataset 51

1. Introduction

When assigned programming exercises, many students lack the time or the knowledge to solve these assignments by themselves. Instead, they sometimes collude in groups or directly plagiarize their solutions, even if neither is allowed. To avoid detection, students usually try to obfuscate their plagiarised submissions through techniques like renaming, reordering, or even exchanging similar control structures like for and while loops.

Today, multiple systems for detecting plagiarisms among student submissions exist, such as JPlag [14], MOSS [1], or Sherlock [25]. Until recently, the working assumption was that fooling these plagiarism detection systems required more work and knowledge than actually doing the assignment without plagiarising [7, p. 138:1]. However, Devore-McDonald and Berger [7] challenged this working assumption by introducing Mossad, an automatic and non-deterministic system for generating multiple plagiarisms based on a single correct solution [7]. Although this software is not available to the public, similar tools could be developed and released to students in the future. This motivates the need for further research and improvements to existing plagiarism detection systems in order to prevent such obfuscation attacks.

In this bachelor's thesis, we analyze how Mossad creates plagiarisms that exploit those weaknesses. Based on this analysis, we design three defense mechanisms that aim to reduce the attack vector on token-based plagiarism detectors. The first two mechanisms remove unused variables and some unreachable sections during tokenization. These mechanisms can be adapted for different languages, but our main focus is on the C and C++ programming languages because Mossad only works for those languages. Our third mechanism is a language-independent prefiltering algorithm for the comparison algorithm.

To empirically validate the effectiveness of our implementation, we evaluate it by running the original version of JPlag and our modified version on a custom-made dataset and compare their key metrics. This dataset consists of real, unplagiarized submissions, plagiarisms created using common obfuscation techniques, and plagiarisms created by Mossad. While we design these improvements for JPlag and its preexisting features, like visually comparing suspicious code, we designed them in a generic way. This way, they can be implemented in similar plagiarism detectors as well.

The main contribution of the thesis is presenting different mechanisms that can be included in existing plagiarism detectors. Additionally, we evaluate our defense mechanisms based on a prototypical implementation in JPlag. Further contributions include a framework for automatically evaluating modifications to JPlag and a Mossad wrapper that allows multi process creation of plagiarism for faster dataset creation.

2. Foundations

This chapter defines the key terminology we use in the later chapters. It also explains the foundations required to understand our defense mechanisms, their prototypical implementation, and their evaluation. First, we define the term plagiarism in Section 2.1, both in general and for software in particular. Then, in Section 2.2 we give an abstract overview of software plagiarism detectors, including the different types. In Section 2.3, we go into more detail about token-based plagiarism detectors and their fundamental weakness. Lastly, in Section 2.4, we introduce *JPlag*, the software plagiarism detector we extend in our implementation and on which we base our evaluation.

2.1. Plagiarism

Plagiarism is "the act of taking the writings of another person and passing them off as one's own [2]." In general, this definition is broad but often used in the context of academic texts lacking proper citation. This thesis focuses on software plagiarism, which limits the definition above to using other people's source code or binaries without permission or attribution. One example of this is taking code from open-source repositories without respecting their license. In this case, the source code of the plagiarized source can be located anywhere, and the source code of the potential plagiarism is usually not available at all. This type of plagiarism is relevant primarily in the industrial sector [32, 3].

Another type of software plagiarism is found in programming courses, where many students try to solve the same, well-defined assignment and submit their solution. Students who lack time, skill, or dedication to solve the assignment themselves might take parts or even the entirety of another student's solution and submit it as their own. We limit our thesis to this kind of software plagiarism. We will refer to the task given to students as *assignment* and their respective solutions to one assignment as *submissions*. Plagiarized submissions take their plagiarized code sections from what we refer to as the *originals*. Students might try to obfuscate their plagiarism by changing aspects of a plagiarized submission. The goal is to escape detection from plagiarism detectors. We will call every single change made to the plagiarism a *mutation* and the process of mutating a plagiarism to escape detection *plagiarism obfuscation*.

2.2. Software Plagiarism Detectors

Software that can detect plagiarism is called a plagiarism detector. Like there are different types of plagiarism, there are corresponding types of plagiarism detectors. We limit this thesis to software plagiarism detection, but note that considerable effort has gone into researching the detection of academic plagiarism [11].

As mentioned above, a key differentiating factor is the necessity for access to the source code of the potential plagiarism. When no source code is available, plagiarism detection has to be performed on the programs binaries. This can be done by performing a *static analysis* or a *dynamic analysis*. During static analysis, a set of features is extracted out of the binaries and then compared to the possible originals. An example feature is the sequence of calls to the operating system APIs. For dynamic analysis, the program is executed, and the runtime behavior is inspected. However, the execution can change between environments, and there is no guarantee about the coverage of possible execution paths [3].

When the complete source code is available, the source code is usually tokenized and then compared. Sometimes, the source code or the tokenization result is filtered to achieve better results. The tokenization steps goal is to abstract away easily changeable source code aspects, for example, whitespace, comments, or variable names. This means that ideally, the result of the tokenization step is mostly invariant during a plagiarism attempt. The concrete tokenization method and the corresponding comparison algorithm vary between plagiarism detectors. Examples of tokenization include source code token sequences, low-level token sequences, abstract syntax trees, or program dependence graphs. The most common one is the tokenization into source code tokens [16]. In this thesis, we refer to software plagiarism detectors that utilize this method as *token-based plagiarism detectors*. JPlag [14] and MOSS [1], two commonly used plagiarism detectors, both fall under this category.

2.3. Token-Based Software Plagiarism Detectors

As previously said, token-based plagiarism detectors generally utilize tokenization followed by comparison. The tokenization usually occurs by parsing or scanning the source code and creating a token sequence, in which each token corresponds to a syntactic concept of the language. Easily changeable semantic information, like variable names, is abstracted away. In a limited capacity, these tokens can also include unchangeable semantic meaning. One example is differentiating braces belonging to a class definition, an if statement or other language concepts requiring braces [14, 22, 23].

The comparison algorithm assigns a similarity score to a token sequence pair and is usually based on string matching [14, 31, 16] or document fingerprinting [24]. These algorithms try to find identical subsequences shared by both token sequences. The similarity score is then calculated from the relative share of tokens that could be matched to the other sequence. In order to avoid random matches, identical subsequences have to be of a certain length in order to be regarded. Otherwise, commonly used code like iterating over array values and reassigning them would be detected as a match. The exact mechanism for preventing these short matches differs depending on implementation. Additionally, the value for this minimum length of a match depends on the concrete tokenization and target language. For this thesis, we will refer to this parameter as *minimum token match*, which is the term JPlag uses internally [14].

The existence of a minimum token match reveals a fundamental weakness of token-based plagiarism detectors: They are susceptible to code insertion attacks [17]. In the

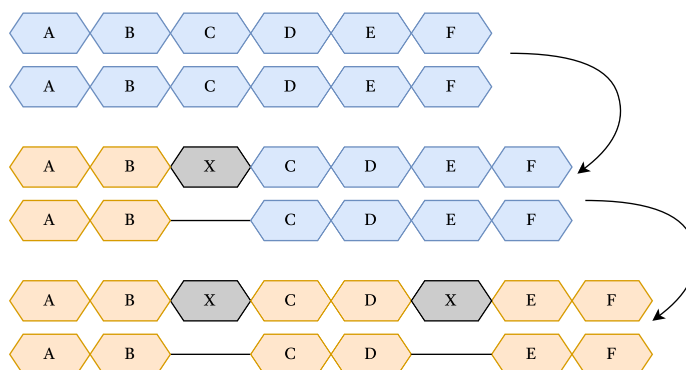

Figure 2.1.: Schematic impact of how code insertions can decrease the similarity of two submissions. Assuming a minimum token match of three, the attacker can prevent every match by decreasing the length of identical subsequences.

context of this thesis, we define code insertion attacks as plagiarism obfuscation attempts by inserting short parts of code into the plagiarism. This additional code does not change the execution behavior of the code, and we will also refer to the additional tokens resulting from the mutation as inserted tokens. While a single inserted token simply splits the match into two, inserting many tokens can cause the length of the matches to fall below the minimum token match. As a result, an otherwise matching subsequence is ignored, and the overall similarity is lowered. Figure 2.1 illustrates this issue with an example minimum token match of three. The gray tokens represent a token resulting from code insertion, the blue tokens represent a token match, and the orange tokens represent unmatched tokens. The inserted tokens prevent every match and lower the similarity to 0%. Other comparison algorithms try to combat this issue, for example, the *smith waterman algorithm* [26] used by Nichols et al. [21] in their work. Initially designed for matching common molecular subsequences, this algorithm considers gaps that are created by tokens inserted in the other sequence. However, increasing the amount of inserted tokens evens out this advantage [7].

2.4. JPlag

Developed in 1996 at the University of Karlsruhe, JPlag [14] provides an open-source solution for locally checking a set of software for plagiarism. Independent studies have proven JPlag's high effectiveness at detecting common plagiarism techniques [5]. JPlag can be built and run entirely locally, which eliminates privacy concerns. As of September 2022, JPlag supports multiple languages, including C, C++, C#, Java, Python 3 [14]. By operating on the parse tree, JPlag is resistant to many obfuscation techniques by design, for example, whitespace insertion or changes in comments. JPlag's uses the same two main phases we described in the previous section [22, p. 5ff], as it is visible in figure Figure 2.2.

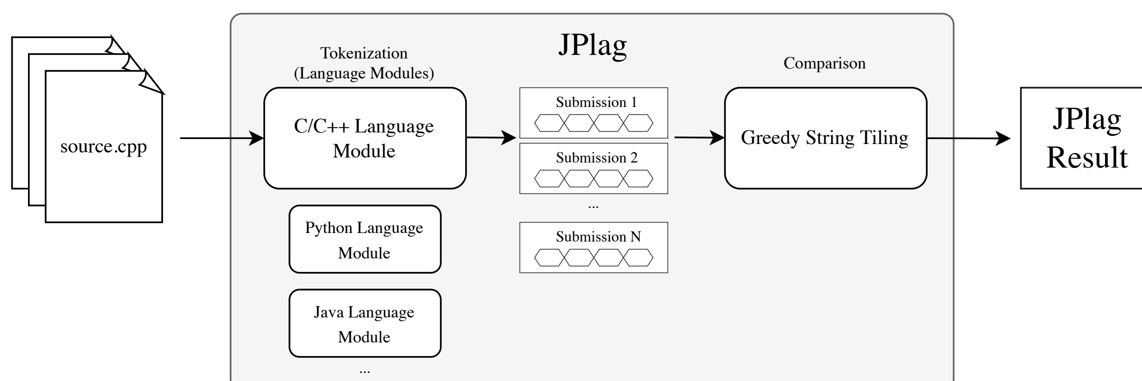

Figure 2.2.: Internal structure and dataflow in JPlag. JPlag can be called using a CLI or as a Java Library. Therefore, the output is either a directory containing JSON files or a Java Object.

2.4.1. Tokenization

JPlag parses or scans the input and creates a token sequence for each submission. This phase is highly language-dependent, and each language module can use whatever method is most suited to create the token sequence. For example, the Java module uses the JavaC API, the C/C++ module uses a JavaCC grammar, and the C# module uses an ANTLR grammar. In parsed languages, JPlag tries to maintain some level of semantic information, like differentiating between ends of methods or ends of classes. For the most part, behavioral details that mainly describe the operational semantics are abstracted away, so the tokens resemble the structure of a program. For example, each method call results in one *apply* token, while information, like parameters, is ignored. This prevents many common obfuscation techniques, like parameter reordering. Each language module can decide what types of tokens there are and how the source code is transformed into these tokens. Unfortunately, the design decisions behind the token selection for each module are not well documented. Although each token type resembles a syntactic concept of the language, like assignments or loops, the semantic meaning of each token is irrelevant for the next phase. Because of this, we refer to these tokens as *language-independent tokens*.

2.4.2. Sequence Comparison

JPlag iterates over each possible submission pair and tries to cover one token sequence with sections of the other using a modified version of *Greedy String Tiling* [31]. An important parameter for this algorithm is the previously discussed *minimum token match*, which is set differently for each supported language. This value holds the minimum amount of tokens that must be identical to mark it as a match. This prevents detecting very short identical sections, like multiple variable declarations, which is common in programs and does not indicate plagiarism. The *minimum token match* plays a crucial role in attacking JPlag.

2.4.3. Limitations

It is important to clarify that JPlag is not a black box that classifies software as either genuine or plagiarism. Depending on the assignment and size of the submissions, a certain degree of overlap is expected. However, it provides a tool to detect anomalies of higher than average code similarity and manually inspect them. JPlag helps instructors by displaying the suspicious code and coloring matching sections. They can then manually inspect and compare the student's code to confirm plagiarism. Additionally, plagiarisms taken from sources outside the submission set, like GitHub and other online sources, can not be detected.

3. Threat Modelling

The process of creating and detecting plagiarism can fundamentally be classified as an attacker-defender scenario. It is therefore essential to define the attacker's capabilities for the remainder of this thesis. In the following chapter, we define the primary type of attack we aim to defend against. We then introduce *Mossad* [7], a tool that students can potentially use to create plagiarisms that escape detection by common software plagiarism detectors. Based on Mossad, we define our threat model, explaining how students plagiarize, how they could use Mossad, and what limitations apply in the context of this thesis.

3.1. Automated Code Insertion

As mentioned in Section 2.3, token-based plagiarism detectors are generally susceptible to code insertion attacks [17]. Previously, these attacks had to be done manually and could require significant time investment [7]. Recent work [7] has shown that it is possible to automate the process of creating a plagiarism only by utilizing code insertion. This type of obfuscator is, although simple, highly effective against token-based plagiarism detectors because of their inherent weakness against code insertion attacks. We define the qualities of a plagiarism obfuscator based on code insertions as follows:

- The obfuscator can insert an arbitrary amount of lines at arbitrary locations.
- The obfuscator can check if its current variant is semantically equivalent to the original.
- The obfuscator has access to the current similarity score of the current variant and the original.
- The obfuscator has access to an arbitrary but fixed set of lines it can insert.

3.2. Mossad

Mossad is a program transformation framework introduced by Devore-McDonald and Berger [7] in their paper "Mossad: Defeating Software Plagiarism Detection" in 2020. Mossad allows the creation of one or multiple plagiarisms from one original. MOSS [1], another token-based software plagiarism detector we introduced briefly in Chapter 2, can not detect this plagiarism. Additionally, the authors claim that the fabricated plagiarisms are unsuspicious during a manual inspection. Although it was developed to attack MOSS, it is also effective against JPlag. Section 3.2.2 further elaborates on why Mossad can be used against JPlag with minimal modifications.

3.2.1. Method of Creating Plagiarism

Because it was developed to attack MOSS, the main idea is to disrupt the hashes created in MOSS' algorithm. Mossad achieves this by inserting new lines of code at regular intervals. However, this has a similar effect on JPlag, because it splits up matching sections and keeps the length of the sections below the *minimum token match*. These new lines are taken from within the submission or provided in an external file and do not affect the program's results.

In practice, Mossad uses an iterative approach: During Initialization, Mossad compiles the source file to object code using Clang with high optimization enabled. The generated object code is later used to ensure the semantic equality of the original and the generated plagiarism. Mossad then uses a copy of the source file as the starting point. Every source file line is added to the *pool*, which is used for generating mutations. Users can use an *entropy file* to provide custom additions to the pool or allow Mossad to add successful mutations to the entropy file for future use.

During the *generation* phase, Mossad selects lines from the *pool* and performs basic semantic checks. These checks ensure the line can be used without causing a compiler error and speed up the next step. After selecting a line, it is sent to the *mutation module*. Mossad inserts the selection into copies of the current variant at random locations and checks if the resulting program is still equivalent. To check whether the variant's program semantics are different, Mossad compiles the variant to object code using *Clang* [20] with high optimization enabled. When high optimization is enabled, Clang tries to find code that has no effect on the execution or result of the program and omits it from the object code. Mossad utilizes this to ensure semantic equality by comparing the originals object code generated in the initialization phase to the object code of the current mutation. If both are not equal, Mossad assumes the mutation has changed the semantics and discards the mutation.

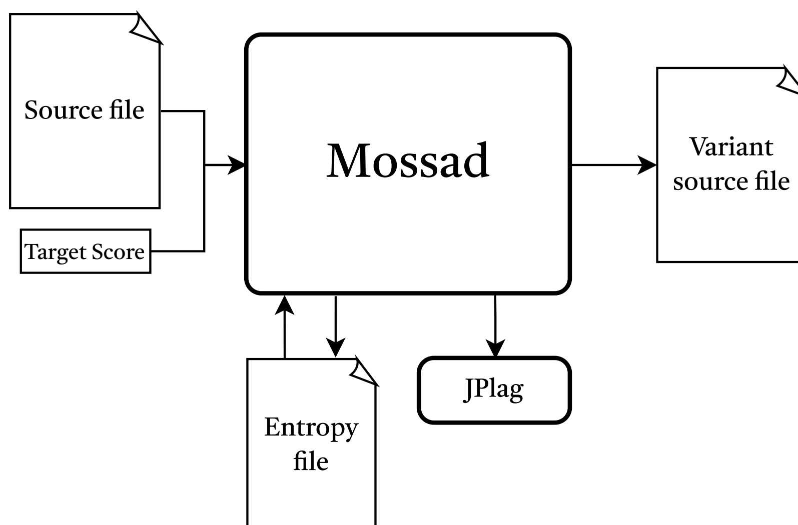

Figure 3.1.: The basic dataflow of Mossad (Simplified version of Figure 3 in [7]). This schematic depicts the ingoing and outgoing data for the JPlag variant of Mossad.

For semantics preserving variants, Mossad then checks if the target MOSS score has been reached by calling the MOSS web service. If this is not the case, the current variant is used in the next iteration. If the target MOSS score is reached, Mossad outputs the current variant and terminates [7], as MOSS can no longer sufficiently distinguish the generated plagiarism from real solutions.

Figure 3.1 shows the data flow when using Mossad. Mossad receives one original submission source file and a target similarity score. While it runs, lines are read from and written to the entropy file, and after every mutation JPlag is called to determine the current similarity score. After Mossad terminates, the current variant has a lower similarity score than the target when comparing it to the original file.

3.2.2. Applicability to JPlag Rather Than MOSS

Although Mossad was created to defeat MOSS, it attacks a weakness that exists in both MOSS and JPlag similarly. While the inserted tokens disrupt the hashes in MOSS, they interrupt matching sequences in JPlag and reduce their length below the *Minimum Token Match*. We have already illustrated this weakness in Section 2.3. Mossad does not regard the explicit tokens corresponding to the inserted lines. Instead, the effectiveness of the random mutations is checked by calling MOSS and checking if the similarity is below the preset threshold. The mutations done by Mossad are, therefore, identical against MOSS and JPlag.

To adapt Mossad for JPlag, we only need to exchange the similarity check call. Devore-McDonald and Berger [7] already provide an adapted version for JPlag 2.11, which we modified to work on newer JPlag versions.

3.3. Threat Model

We base our new threat model on the following threat model created by Devore-McDonald and Berger [7, 138:7]:

Threat Model

- "Students are incentivized to minimize the time spent on assignments while maximizing their grade. Therefore, students will not spend more time attempting to defeat the plagiarism detector than actually doing the assignment."
- "Students have arbitrary access to peers' assignment solutions or solutions found online. Students may use these solutions as the basis for their plagiarized assignment solution."
- "In addition, all solutions are also available to the instructor; attacks that depend on outsourcing (hiring someone to write an entirely new program that is not available online) are to our knowledge beyond the scope of existing plagiarism detection tools, since the original program is unavailable."

- "Course staff relies primarily on software plagiarism detection, beyond the lightweight manual inspection described below."
- "Course staff will only examine student code for spot-checking or for those assignments that receive high similarity scores with another assignment. [...]" [7, 138:7]

In the context of this thesis, we assume that the students have access to Mossad or a similar plagiarism obfuscator utilizing code insertion. We, therefore, extend the threat model to include:

- Students have access to Mossad or other software that performs automatic obfuscation attacks using code insertion.
- Students have access to our modified version of JPlag and know how to use it with Mossad. As a result, Mossad knows if the plagiarisms it creates are detected by our modified version of JPlag. Mossad can then keep mutating the variants until JPlag can not detect them anymore.

Additionally, we set the following limitations:

- Students cannot construct complex entropy files and mainly rely on entropy automatically introduced by other submissions. We assume that students would take longer to construct entropy files than to solve the assignment themselves.
- Students only use Mossad to create their plagiarism. Creating a Mossad plagiarism out of pre-existing plagiarism or further obfuscating the result of Mossad is outside the scope of this thesis.
- Students can remove lines introduced by Mossad that reduce readability without decreasing the similarity. Mossad inserts more lines than are necessary to reduce the similarity below the threshold, which can raise suspiciousness during manual inspections.

3.4. Role of Manual Inspection for Plagiarism Detection

A survey conducted by Devore-McDonald and Berger [7, 138:6-7] indicates that many respondents only manually inspect submission pairs that receive high similarity by plagiarism detectors. This is supported by the fact that the amount of submission pairs is equal to $\binom{N}{2} = \frac{N!}{2(N-2)!} = \frac{N(N-1)}{2}$, where N is the total amount of submissions. This quadratic scaling means that even for relatively small courses, the number of submission pairs can quickly reach thousands. For example, 50 submissions mean a total of 1225 combinations. We can therefore assume that plagiarism that does not receive a high similarity rating is not detected during spot checking.

As stated above, it is essential to recognize that software plagiarism detectors can not classify student submissions as plagiarism with absolute certainty. Whenever a pair of student submissions receive a high similarity value, instructors must take further steps to

investigate the case. This should always include a manual inspection of the source code in question. JPlag provides a user interface for directly comparing the sections recognized as similar. Should the instructor find that the submissions contain identical or similar sections, he or she can request the students in question to defend their code.

Because of this, our primary goal is to maximize the similarity of Mossad plagiarism, so instructors are likely to check them manually. A small number of false positives can be discarded and merely increase the required effort. This means we have to keep the precision high enough to make a manual inspection of all high similarity pairs feasible.

4. Defense Mechanism

Resilience against obfuscation attacks means that the obfuscation does not decrease the similarity of a plagiarism pair. In order to improve resilience against the attacks defined in Chapter 3, our primary goal is to remove tokens from plagiarism token sequences that belong to inserted lines. We can do this in two different ways:

- Remove or ignore tokens during parsing. Because this action requires the language's explicit token types and semantic meaning, this is *language specific*.
- Ignore tokens during the comparison of two submissions. Ideally, this can be performed on generic token types, meaning the semantic meaning of a token is irrelevant, and only differentiation between types is required. This limitation makes this *language independent*.

The following chapter describes two common types of code insertions and mechanisms to counteract them. We found that these code insertions make up a large share of automatically inserted lines of code. Additionally, we propose an algorithm that iteratively checks if two token sequences can be made equal by deleting one short token subsequence. This algorithm aims to delete code insertions for arbitrary programming languages.

4.1. Deleting Unused Variables

Deciding if a variable is unused is trivial in specific cases but very complex in other cases. Variables that are only declared or written to but never read from have no impact on the execution and result of a program and are, therefore, always unused. However, more complex cases are hard or impossible to decide. A selection of these cases includes:

- Public class members that are not read directly but are part of the class's public interface.
- Private class members that are only read in an access method.
- Class members could be read in derived classes that are not available to the plagiarism detector.
- Variable reads that are located in unreachable code. The problem of classification of *unreachable* is described in section Section 4.2.
- Variables that are read, but their contained value does not affect program execution or result. A simple example is adding an unused variable to another unused variable.

Considering these complex cases would exceed the scope of this thesis. Instead, we limit our scope to declared but unread local variables. Local variables are strictly limited in access to their local scope, which means that all read accesses to the variable must be contained within the local scope. Therefore, checking for the mere existence of read access is straightforward. The reachability of the read access is much more complex. For example, Chen, Gansner, and Koutsofios [4] elaborate on the complex relations that can occur in C++ code and the difficulty of determining if a statement is executed. Therefore, we restrict this thesis to the existence of read access. Future research can lift this restriction to detect more sophisticated plagiarism.

Because the insertion of unused variables and the resulting check for semantic similarity are easy to perform automatically, mutations of this kind occurred very frequently when we ran Mossad. For a valid mutation, the declaration must only be in a valid location and not shadow an existing variable with the same name.

The term *shadowing* refers to the situation in which a variable is declared in an inner scope, although there already exists a variable in an outer scope with the same name. The value of the wider scoped variable is then inaccessible within the smaller scope, and the execution behavior changes [28, p. 158f].

We suspect that single variable insertions with ambiguous names like *count*, *status*, or common single-letter names like *n*, *k*, or *i* are easily missed during manual inspection. We base this suspicion on the study by Devore-McDonald and Berger [7]. In their experiment, they presented plagiarism created by Mossad to programming instructors and asked for any suspicious findings. Since we found that a large share of Mossad's mutations is insertions of unused local variables, we conclude that this type of code insertion is easily missed during manual inspection. It is, therefore, crucial to prevent declarations of unused variables from affecting plagiarism detection. How tools can implement this defense mechanism is described in Chapter 5.

4.2. Deleting Tokens in Unreachable Code

Another way of adding lines that do not change semantics is inserting *dead* or *unreachable* lines of code. In general, code is defined as "unreachable if there is no control flow path to it from the rest of the program" [6, p. 382]. This is commonly the case when the propagation of parameter values guarantees that a conditional branch is never taken. A rarer and more obvious occurrence is code located after guaranteed jumps, like `return` or `break` statements. [6, p. 382]

In contrast, "dead code refers to computations whose results are never used" [6, p. 382]. This includes computations whose only usage lies within unreachable code. Both concepts are, therefore, closely coupled. When classifying code as dead, it is important to consider side effects like exceptions or raised signals. Any side effect can impact the execution or result of the program; thus, any code that includes side effects is not dead, even if the computational result is not used directly. [6, p. 382]

These concepts are commonly used in compilers when performing *code compaction*, which is the act of reducing the size of the resulting executables. This can allow the program to run on lower-memory devices [6, p. 379]. In the context of software plagiarism,

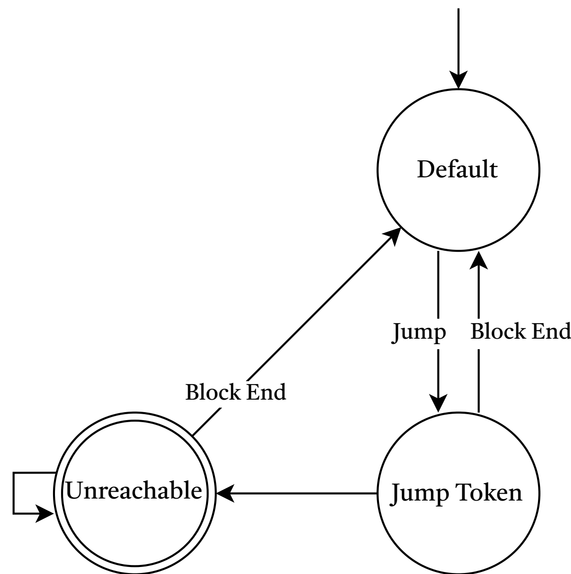

Figure 4.1.: Naive state machine for detecting tokens after guaranteed jump tokens.

dead and unreachable code can either reduce the relative similarity by increasing the total amount of tokens or even break up continuous tokens to prevent a match. Because unreachable code will never be executed, all valid code insertions in the unreachable section will yield a semantically equal mutation.

4.2.1. Naive State Machine

In this section, we will introduce a state machine that detects tokens inserted after guaranteed jumps in linear time on the token level. Although this state machine is designed for C++, it can be adapted for other languages that do not prohibit tokens after guaranteed jumps, for example, Python. Based on JPlag's implementation, we also assume that the C++ code is only scanned and not fully parsed. This state machine is much simpler for fully parsed language that introduces more semantic information into the token level. This means our assumption is not a restriction but a generalization.

We construct our state machine by starting with a naive state machine that can detect code insertions after guaranteed jumps but is prone to exploits. Figure 4.1 shows a representation of this naive state machine. The state machine uses token type categories instead of explicit token types to keep generality because token types vary between implementations or languages. *Jump* includes all guaranteed jump tokens, like `return`, `break`, `throw`, or `continue`. *Block End* refers to tokens that indicate the end of a code block. This can be a closing brace, or should the tokenizer support full parsing, a token indicating the closing of this specific block.

We can use the state machine by starting in the initial state and reading each token in the sequence sequentially. Should a token lead into the accepting state, this token should

be deleted, and the state machine resumed for the next token without leaving the current state.

Since our state machine expects *Block End* types of tokens to leave the *unreachable* state, it is susceptible to large blocks of inserted code containing `if`, `while`, or other control structures that can contain explicit code blocks. Modifications to our formal state machine can not fix this issue directly because it cannot count the depth of nested code blocks within the unreachable section. A simple fix we can apply in practice is introducing a *depth counter*. This counter is initialized as 0 when entering the *unreachable* state. Every *Block Begin* token increases the depth counter by one, and every *Block End* token decreases the counter by one. Whenever the state machine is in the *unreachable* state and encounters a *Block End*, it checks the value of the *depth counter* and bases the state transition on it. Should the value be 0, then it simply transitions like usual. Should the value be higher than 0, then the state machine remains in the *unreachable* state, the token is deleted, and the machine continues. We leave this modification out of our graphical state machine representations, as the concept of variables theoretically does not exist for state machines.

4.2.2. Limitations and Addition for Scanned Languages

Many C-like languages like Java, C#, and C++ allow the omission of braces in control structures when the block only contains a single statement. This is usually not an issue in parsed languages because end-of-block tokens are possible without braces. Scanned languages, however, present the issue that they can only create tokens from explicit source code. This is especially dangerous because the end of blocks is not always represented as a token and can be missed. The code example shown in Listing 4.1 shows a situation where the naive state machine wrongly deletes tokens.

```
int main()
{
    int a = 10;
    if (a == 0)
        return;
    // All tokens below are wrongly deleted until
    // the end of the function
}
```

Listing 4.1: Code example illustrating an exploit in C++ when using the naive state machine illustrated in Figure 4.1

A first idea for fixing this issue might be to check whether the `if` is followed by an opening brace. If not, the token after the `if`-token could be expected as a single statement in the implicit code block. This, however, is not viable because an arbitrary amount of tokens can belong to the condition. Listing 4.2 illustrates the issue with two code examples that yield the same token sequence when scanned. This proves that the tokens do not contain enough semantic information to differentiate both situations fully. The lack of semantic information about the start and ends of code blocks is caused by the assumption that a scanner instead of a parser for tokenization.

```
int main()
{
    int a = 10;
    if (a == (b = c))
        return;

    // reachable code
}
```

```
int main()
{
    int a = 10;
    if (a == 0)
        b = c;
    return;
    // unreachable code
}
```

Listing 4.2: Two code examples can result in equal tokens in scanned languages, depending on tokenization implementation.

To safeguard against this exploit, we add the necessity for explicit starts and ends of blocks. Figure 4.2 shows the change to our state machine. We introduce a *Block Beginning* state that is traversed whenever a block starts. This state ensures that only blocks with explicit start and end tokens are regarded for unreachable code sections. A negative side effect is that the state machine can remain in the *Block Beginning* state, even if the corresponding code is directly after a jump token. As a result, unreachable code, like shown in Listing 4.3, will not be filtered out of the resulting token sequence. This situation is only resolved when the next *Block Beginning* or *Block End* token is encountered since these tokens ensure that the block without braces has ended.

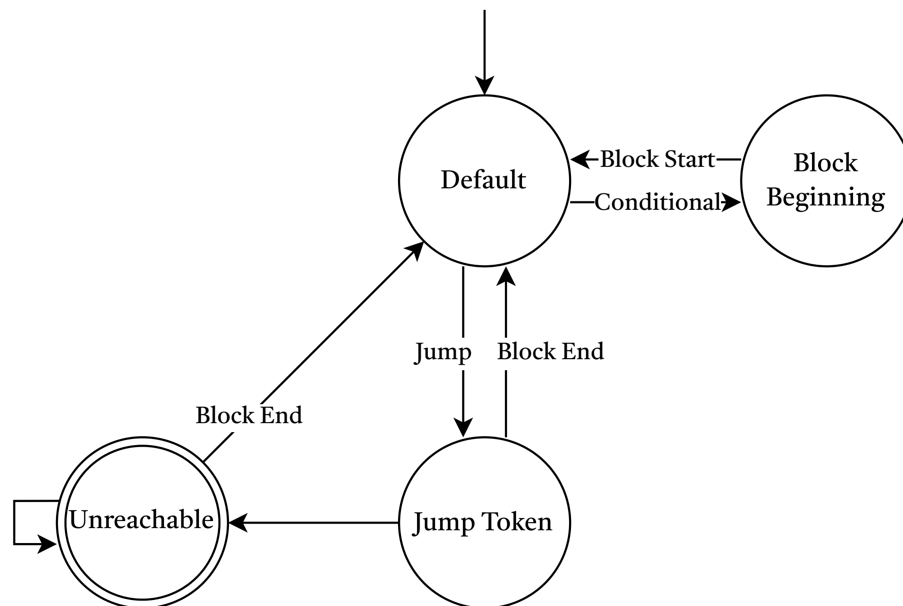

Figure 4.2.: State machine that fixes the token deletion exploit of the naive implementation of Figure 4.1 as shown in Listing 4.2

```
int main()
{
    int a = 10;
    if (a == 0)
        a = 5;
    return;
    // The following tokens are wrongly not deleted
    int n = 0
}
```

Listing 4.3: Code example that contains an undetected inserted token.

There are cases where the blocks' ends are explicit without bracing, even in scanned languages. One specific example we regularly encountered in C++ were inserted lines after `breaks` in switch statements. Because the `case` keyword acts as block start and end, we add a state to include this as an edge case. Figure 4.3 depicts the full state machine.

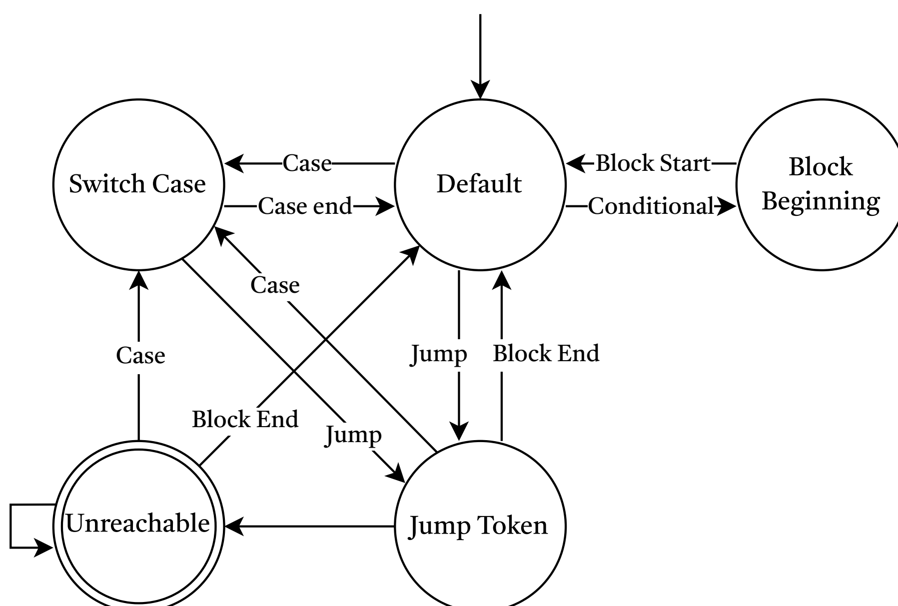

Figure 4.3.: The final state machine.

4.3. Directly Reverting Token Insertions

As mentioned above, the critical weakness of Token-Based Detectors is that they require long identical sequences of tokens to detect matches. As a result, even single token insertions break up two otherwise identical sections. If the resulting match length is below a certain threshold, it is discarded and not recognized as plagiarism. This is especially dangerous because minimal changes can result in an extra token depending on the tok-

enization step. Carefully placed, this can reduce the similarity of a submission pair down to 0% without considerably affecting readability.
A practical example of this is the usage of fully qualified names in C++. We found that in the original tokenization implementation of JPlag 3.0.0, using explicit namespaces (`::`) resulted in *scope* tokens. Consequently, `std::cout` and `cout` differ after tokenization, so manually adding or removing explicit namespaces is equivalent to token insertion or deletion. We will not further discuss the effects of tokenization implementations and token types in this thesis, but we mention it in Chapter 8 as future work. This nevertheless motivates the need for directly addressing this fundamental weakness.

4.3.1. Naive Algorithm

To detect short but arbitrary token insertion like this, we examined the viability of an algorithm that tries to revert the insertion directly. We do this by comparing two token sequences and looking if the deletion of a short coherent subsequence makes the two sequences equal. The explicit token types are irrelevant; we only need to be able to discern different token types, which makes this algorithm applicable to every language. Figure 4.4 depicts two token sequences and shows the basic concept of the algorithm, which is explained in the following. The second sequence was created by inserting a token marked with **x** into the first token sequence.

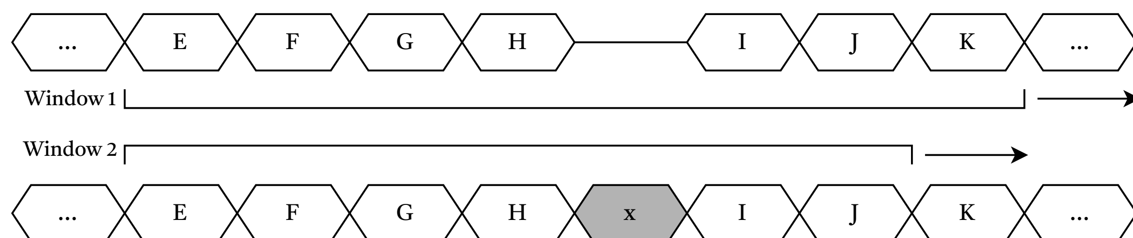

Figure 4.4.: Basic concept of using running windows for detecting similar sequences.

We start by trying to find inserted tokens in the first sequence and will then swap the sequences and rerun the algorithm. To detect and revert this insertion, we use two different running windows, one for each token sequence. We iterate the first window over the first token sequence. We iterate the second window over the second sequence for every first window position. Both windows have a set length, and their position on their respective token sequence is incremented with a predefined increment value. We will refer to these configurable values as *Window Length* and *Window Increment*.
For every window combination, we search for inserted tokens in the entire window. We do this by iterating over the tokens in both windows simultaneously and checking for equality. After the first occurrence of a difference in token types, we skip and count tokens in the first window until the token types are identical again. We limit the length of skipped tokens to a maximum value called *maximum insertion length*. Should the token mismatch be longer than this value, or should multiple mismatching sections occur, we skip to the next second window. Otherwise, we delete the mismatching section.

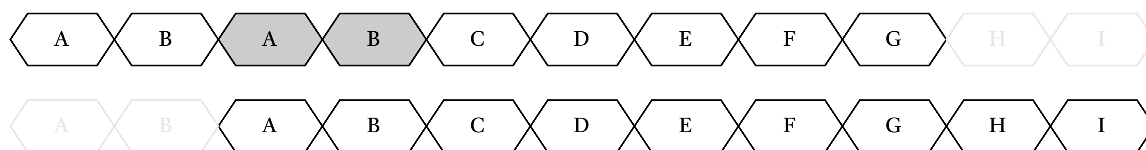

Figure 4.5.: Two slightly offset windows with repeating subsequence at the beginning. The gray tokens are wrongly detected as code insertion.

4.3.2. Key Considerations

When thinking about this algorithm, one quickly discovers edge cases that prohibit the expected function. In the following, we will explain edge cases that must be considered when implementing this algorithm.

When dealing with two identical token sequences and windows that offset by *n*, where *n* is limited by the *maximum insertion length*, the first *n* tokens will be detected as insertion and Therefore removed. This occurs multiple times and ultimately prevents the detection of verbatim copies. It is, therefore, essential to ignore tokens equivalent to the *maximum insertion length* at the beginning and end of the window. Furthermore, certain situations where the edges of the sequences have repeating subsequences, as shown in Figure 4.5, can cause this phenomenon to extend beyond the *maximum insertion length*. Therefore, we introduce padding at the start and end of the window to reduce the frequency of this occurring. While this reduces the likelihood of a false positive, it also increases the required space between token insertions. Although we cannot eliminate the risk of false-positive deletions, since a single token difference of otherwise identical sequences might even occur on semantically completely different code, we evaluate the frequency with which this occurs in Chapter 6.

5. Implementation

To evaluate our defense mechanisms, we implemented them prototypically into JPlag. This allows us to evaluate our approaches' effectiveness and showcases how they can be integrated into existing token-based software plagiarism detectors. Token-based plagiarism detectors generally use a tokenization phase followed by a comparison algorithm [1, 21, 29]. Therefore, we expect our implementation approach to be transferable to other token-based detectors. We also created a framework for automatically running our implementations on different datasets and different versions and configurations of JPlag. In the following chapter, we explain how we implemented our prototypes and give insight into how our findings can be transferred.

5.1. Implementing the Approaches in JPlag

JPlags' two main phases, tokenization and comparison (see Section 2.4) allow us to implement our defense mechanisms as preprocessing steps for each phase. The tokenization step is done in the *language modules* and handled differently for each language. Consequently, we can limit our language-specific modifications to the C++ language module. Our preprocessing on the generic tokens can be done directly before each call to the core algorithm, encapsulated in a class.

5.1.1. C++ Scanner Extensions

The C++ `Scanner` class accepts scanned tokens and collects them into a `TokenList` object for further processing in the JPlag core. The tokens are passed to the scanner by the `CPPScanner` generated by *JavaCC* [13], which is a "parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar [13]." It is important to distinguish between the `Scanner` and `CPPScanner` classes, especially since both are contained in the `de.jplag.cpp` package. The `CPPScanner` performs the language parsing, but because it is generated by JavaCC, we cannot modify it directly. We implement our language-specific defense mechanisms by intercepting tokens when they are passed to the `Scanner` object and by filtering the list before it is built into the `TokenList`.

5.1.1.1. Deleting unused variables

In order to delete unused variables, we could implement our own source analysis tool to scan the code directly and remove the declarations from the source code. In practice, this is not feasible due to time constraints and the general complexity of the C++ programming

language. Instead, we use pre-existing tools to analyze the code before parsing and find unused variables this way. We configure the tools to output warnings about unused variables and parse their output to receive the lines that contain them. The `Scanner` can then discard the tokens belonging to the declaration before they are added to the token list.
We decided to use compilers and their warnings for our implementations. Specifically, GCC is a robust and proven compiler that also scans the code for possible user errors, including unused variables. Listing 5.1 shows an excerpt of an example output produced by GCC. The warning contains the file name, line and column, severity, and the type of the warning, in this case, "unused variable" followed by the variable's name. A colon separates this information, making it easy to extract from the output string.
We encapsulate our source analysis in a new `SourceAnalysis` class. Its public interface consists of only two methods. The first method starts the analysis on one submission, reads the compiler output, and caches the line numbers that contain unused variables. The second method checks if the passed token belongs to a line detected in the previous method and returns the result as a boolean. A new `SourceAnalysis` instance is created for every submission the scanner scans. The scanner then issues the analysis. Whenever the `CPPScanner` passes a token to the `Scanner` instance, the latter checks the `SourceAnalysis` instance if the token should be ignored. If it should, the token is discarded and not added to the token list.

```
plagiarism.cpp:122:21:  warning:  unused variable ’result’ [-Wunused-variable]
                int result = 4;
                    ^
plagiarism.cpp:123:21:  warning:  unused variable ’status’ [-Wunused-variable]
                int status;
                    ^
```

Listing 5.1: Example output warnings about unused variables. Generated by Clang [20] or GCC [12] using the -Wall or -Wunused-variables option [8, 9].

This prototypical implementation works well for basic, single-line code insertions. However, certain edge cases can prevent the detection of unused variables and even reduce the quality of plagiarism detection entirely. These issues are partly caused by our approach of only modifying the token level and not the source code directly. Our token-level approach causes issues when multiple variables are declared in the same line, only some of which are unused. JPlag produces one token for the type, regardless of the number of variables declared. This means that the token belongs to both used and unused variables simultaneously. Since we discard entire lines, the used variable is wrongly ignored. This is not an issue when we assume that only self-contained lines are inserted. However, a clever plagiarist knowing of this flaw, could introduce new variables into existing variable declarations and essentially delete tokens. Deletion of tokens is equivalent to the insertion of tokens, which means the core algorithm misses an otherwise detectable match.
Additionally, the JPlag tokenization step handles line numbers belonging to tokens differently than the GCC compiler warnings. The tokenizer assigns the line numbers to the location of the type identifier, while the GCC compiler warnings include the line number

of the variable name. For code that is formatted like expected, these line numbers are identical. The issue presents itself when the code is formatted as follows:

```
int // tokenizer saves this line number
    unused = 0; // compiler warnings give this line number
```

The lines of the compiler warning and token differ. Therefore our source analysis allows the token to be added to the token sequence - the code insertion attack is not prevented. Working on copies of the source code and preprocessing them in the future could solve both issues. The preprocessing step would include separating every declaration of multiple variables into multiple declarations of single variables. Additionally, we could integrate a code formatter so that the line numbers always match the compiler warnings.

5.1.1.2. Deleting unreachable code

In order to delete unreachable code, the scanner also utilizes an implementation of the state machine described in Section 4.2. The entire state machine is encapsulated in the new `BasicTokenFilter` class and is invisible to the caller. The class's public interface is a sole static `applyTo` function, which receives a generic list of `CPPToken`. The token filtering occurs in-place on the referenced list because the `Scanner` works on a `LinkedList`, rather than the usually recommended `ArrayList`. We justify this loss of cache efficiency because the list is only used for inserting and deleting. After the token sequence is formed, it is transferred to an `ArrayList` within the `TokenList`.
The state machine implementation uses an enum nested within the `BasicTokenFilter` class. Enums are usually used to define variables with a fixed amount of possible values. In Java, the enum values are class instances, which allows us to declare a method in the base class and override it in each possible enum value. Conceptually, each state of the enum corresponds to one state of the state machine. We define the state transitions by defining the `nextState(int tokenType)` in the base enum and overriding it in each state. The token type is encoded using the `CPPTokenConstants` interface, and the method returns the new state corresponding to the token input. To define the accepting state, we declare a method in the enum base class that returns `false` by default and overwrite it in the accepting state's subclass. This results in a very concise and readable implementation, as the entire implementation basically only consists of the states and a `if` chain deciding the new state. To further increase readability and maintainability in case of a change in the token set, we group some explicit token types. Instead of handling the explicit types, the `nextState` method can instead query the methods to determine if the current token belongs to the corresponding group. Figure 5.1 shows an extract of an UML class diagram representation of our state machine implementation. The additional states are implemented like defined in Figure 4.3.

5.1.2. Core Algorithm Extension

Our implementation of the generic token filtering algorithm described in Section 4.3 is also encapsulated in a class called `GenericTokenFilter`. The public interface only consists of the constructor, the method that performs the filtering, and two getters for retrieving

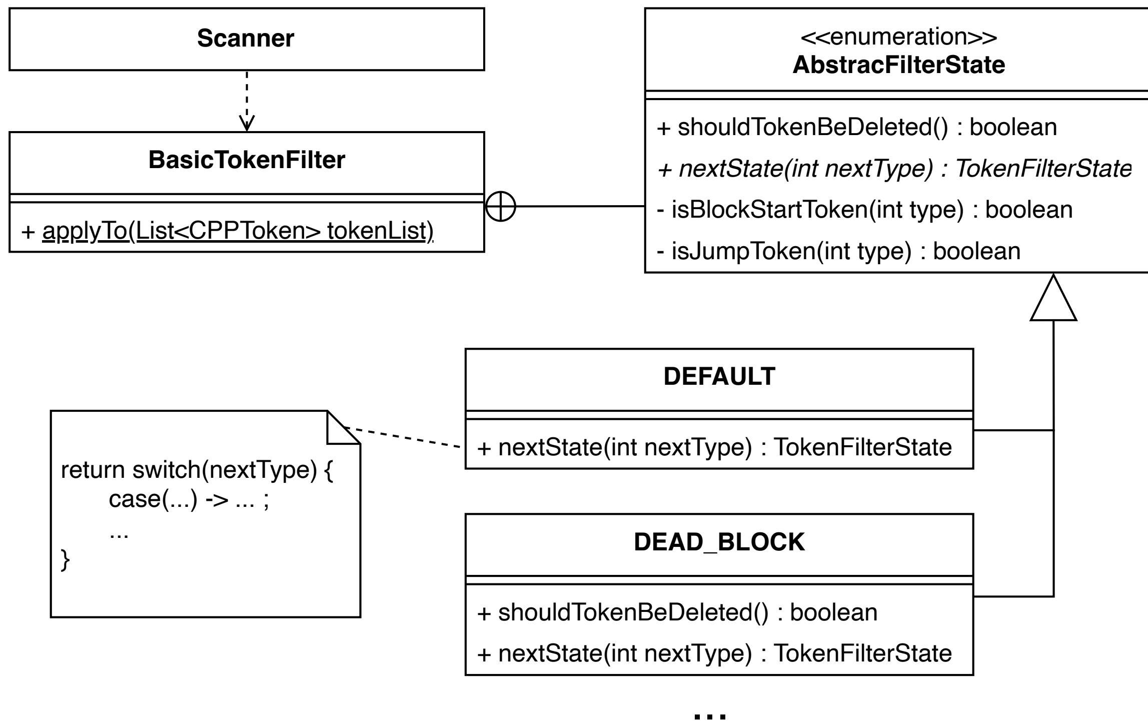

Figure 5.1.: Incomplete UML representation of the token filter and the state machine implementation. The subclasses are the enum values and correspond to one state each. Most states are missing but are implemented similarly.

the result. The constructor expects the two `TokenList` objects and an object containing options. The algorithm itself creates copies of the provided `TokenList` objects because the deletions done are exclusive to one specific pair.
The encapsulated algorithm is called directly in the `GreedyStringTiling`, which is the core algorithm of JPlag. There, it is checked if the additional filtering is enabled. If it is, the `GenericTokenFilter` is instantiated, the `TokenList` objects are passed, the algorithm is called, and the results are retrieved for further use in the core algorithm.

5.2. Framework for Automatically Evaluating JPlag

In addition to our prototypical defense mechanisms, we also implemented a program for automatically evaluating JPlag. We will henceforth refer to this program as the *Evaluation Framework*. Although it is not our main contribution, we believe it benefits future research and implementation work based on JPlag. Therefore, we will briefly introduce the evaluation framework's basic concepts.
The Evaluation Framework is a Java program that allows us to run different versions of JPlag with different configuration options on different labeled datasets. The results of the individual JPlag runs are then written to CSV files for further analysis. It simulates the manual use of the JPlag Java API, so the configuration possibilities are limited by the `JPlagOptions` Java object.

5.2.1. Configuration

To use the Evaluation Framework, the user has to provide JPlag JAR files and assignments containing student submissions. Then the run configurations have to be written into a configuration JSON file. A configuration is a set of JPlag options and is always connected to one JAR file. An arbitrary number of JPlag versions and configurations for them can be used to compare the impact of implementation changes or different JPlag settings.
The run configurations have to be set by providing the name of the JAR file and an array of *options overrides*. One option override represents a method call on the JPlagOptions object and therefore contains the name of an access method, the parameters data type, and the value. The data type and value must match the parameter of the access method. We currently support integers, floats, boolean, and enumerations. The enumeration data type corresponds to the full classpath of the enumeration, and the possible values are string representations of the enumeration values. For example, to enable parallel comparison we can set the access method to `"setComparisonMode"`, the data type to `"de.jplag.strategy.ComparisonMode"`, and the value to `"PARALLEL"`. Although this approach to configuration is unsafe, it gives us great flexibility and allows us to set newly added JPlag options without extending the Evaluation Framework implementation.
The different datasets containing the student submissions must be labeled using a truth file for each. In these, the user lists the pairs or tuples of submissions for each possible pair type.

5.2.2. Running the Framework

When running the Evaluation Framework, first, the configuration is loaded. This means parsing the configuration file and the dataset truth files. The truth files are cached in a map that provides a quick lookup of the type of a submission pair.
To run a JPlag jar using a configuration, we have to pass the JAR path and configuration to a `JPlagWrapper` class. This class encapsulates the entire API usage of JPlag so that no objects from JPlag leave the wrapper. It handles the loading of the JAR file and uses reflection to load all classes that would be required during the usage of JPlag via its Java API. This also includes possible enumeration types used for the option overrides. During initialization, the wrapper also sets the desired values in the `JPlagOptions` using the accessors provided in the option overrides. If the initialization is successful, JPlag can now be run by calling the wrappers `run()` method, which takes the dataset as a parameter.
The results are contained in a class mirroring JPlags `JPlagComparison` class. The Evaluation Framework handles the instantiation of the wrapper for each configuration and calls it for every dataset. The resulting data is then cleaned up and written into CSV files. An additional summary file provides a quick comparison between the different configurations.

6. Evaluation

In order to evaluate the effectiveness of our defense mechanisms and their prototypical implementation in JPlag, we created a dataset containing original plagiarized submissions. We ran our modified version of JPlag using different parameters. Section 6.1 describes how we conducted our validation. This includes creating the dataset and using our previously described evaluation framework to get our data. Section 6.2 then showcases our findings, using the defense mechanisms individually and in combination. In Section 6.3, we discuss how our implementation affects the generation of new plagiarism. Lastly in Section 6.4 we assess and discuss threats to the validity of our evaluation.

6.1. Methodology

We evaluate our implementation using a *Goal Question Metric* (GQM) approach [27], which we perform using the GQM plan depicted in Table 6.1. Mossad checks the similarity of its current variant by calling JPlag; therefore, we have two main goals: Our first goal is to improve the detection efficacy for already existing Mossad plagiarism. To obtain the metrics for this goal, we run our improved JPlag version on the dataset described in Section 6.1.1. This dataset contains original submissions and plagiarism Mossad created without access to our defense mechanisms.
For the second part of our evaluation, we give Mossad access to our defense mechanisms and try to create new plagiarism. Mossad only terminates when the similarity is below a threshold, so achieving a high similarity is impossible. Therefore, our second goal is to increase Mossad's runtime and line insertions required to create a low similarity variant when Mossad uses JPlag with all our mechanisms enabled.

6.1.1. Dataset

The creation of our dataset consists of two main parts: finding suitable collections of student submissions as a basis and then creating plagiarism out of them. The basis for our dataset must qualify some key requirements:

- Multiple different assignments with many submissions for each
- Each assignment should yield submissions of about 100 lines or longer in order to reduce the likeliness and effect of random code similarity.
- Every submission must be written in C or C++

We use a dataset created by Ljubovic [18] for their paper "Plagiarism Detection in Computer Programming Using Feature Extraction From Ultra-Fine-Grained Repositories [19]" as a

basis. This dataset contains assignments from multiple years, including many anonymized student submissions for each assignment. Additionally, the entire edit history of each file is included to enable experiments about plagiarism detection with this information available.
The three provided ground truth files list potential plagiarism due to failure of doing an oral defense by the student in or for submission pairs yielding high similarity scores when using plagiarism detection tools. Unfortunately, we found that the ground truth files are not reliable for our purpose: Some submission pairs contained identical sections without being listed in the truth file; some cases were even completely identical. Additionally, submissions of students who failed to do an oral defense are listed as plagiarism without mentioning the original. Therefore we cannot use this information to determine the success of plagiarism detection for pairs of submissions. In order to still label the dataset, we manually inspected large parts of the dataset and removed verbatim copies and pairs containing identical code sections. As a result, the similarity of non-plagiarized submissions seems to be lower than what can be usually expected from assignments this size. However, we are confident that no actual common plagiarism pair is wrongly labeled as non-plagiarism. The pairs we removed were similar enough that even if they are, in fact, not plagiarism, a manual inspection would still be advised. After inspecting the size and number of available submissions for each assignment, we chose four assignments.
To create the Mossad plagiarisms, we wrote a python script that can run multiple Mossad processes in parallel. To retain validity, we designed this script not to alter the functionality of Mossad itself. Each instance picks a random submission from one of the four assignments and creates a Mossad plagiarism using a current version of JPlag. After Mossad successfully creates a variant, our script saves it, cleans the instance, and randomly picks a new submission. The entropy file is preserved, meaning Mossad has access to successful mutations from multiple submissions for the same assignment. This follows the threat model we defined in Section 3.3, which states that students have arbitrary access to their peers' solutions. At this stage, Mossad has no access to our JPlag version and its defense mechanisms. We evaluate the effect of our prototype on new plagiarism generation in the second part of our evaluation.
When inspecting Mossad's plagiarisms, we found that many of them had long sections of mutated lines without any original code. This is highly suspicious, so we believe students would manually inspect their plagiarism and try to shorten it as much as possible without affecting the similarity. Therefore, we limited the maximum length of these mutated blocks to two lines for some of the Mossad plagiarisms, which we will call *minimized*. To ensure this restriction, we manually deleted the lines between the section's first and last line of the section. In some cases, this reduced the total lines of code by up to 50% while only affecting the similarity by a small amount. We believe students might be willing to take this tradeoff in suspiciousness between manual and automatic detection methods. Nevertheless, we will treat the original and minimized Mossad files separately in the following evaluation.

6.1.2. GQM Plan

We evaluate our first goal by running our JPlag version on our dataset with different sets of defense mechanisms enabled. In our first question, 1.1, we test the similarities achieved

	Goal		Question		Metric
1	Improve detection efficacy using new defense mechanism	1.1	Is the detection efficacy increased when using every mechanism simultaneously?	1.1.1	Similarity score of Mossad plagiarism pairs
		1.2	How much do the single mechanisms contribute?	1.2.1	Similarity score increase when enabling single mechanisms
		1.3	Are the mechanisms applicable to everyday use?	1.3.1	Runtime overhead of using every mechanism
				1.3.2	Share of non-plagiarism pairs that receive a high similarity
2	Impede the creation of new plagiarism using plagiarism obfuscators	2.1	Is plagiarism creation harder for plagiarism obfuscators?	2.1.1	Increase in iterations required for finding a low similarity variant
		2.2	Is Mossad still applicable?	2.2.1	Time investment for creating a low-similarity variant that retains readability

Table 6.1.: GQM-Plan

by enabling all mechanisms simultaneously. This gives us a first idea of whether the defense mechanisms we implemented succeeded in assigning Mossad plagiarisms a high similarity score. In an ideal scenario, the similarity of the Mossad pairs should be 100%, as this would mean all inserted tokens were deleted. To answer question 1.2, we must determine the single mechanisms' impact. We do this by activating only one mechanism at a time and looking at the change in the similarities. This allows us to estimates the impact of each defense mechanism. Additionaly, it can uncover potential faults in our concepts or implementations if we find that single mechanisms have no effect at all or are hindering the detection more than they help. Our final question for goal 1 is how the mechanisms affect non-plagiarism pairs by measuring the share of non-plagiarisms that wrongly receive a similarity. This ensures that we do not just shift the entire similarity upward by a similar amount. Instead, only Mossad plagiarism should be significantly affected.

Because students trying to create plagiarism can gain access to all our implementations, creating new plagiarism must be significantly more difficult. Otherwise, Mossad can add mutations until the desired similarity score is reached. Since Mossad uses a non-deterministic approach, we have to measure the number of iterations and unsuccessful mutations it takes before a plagiarism is created successfully. The average number of iterations is our metric 2.1.1. To estimate the applicability, we also measure the time Mossad took to create plagiarism as our metric 2.1.2. Additionally, we assess the quality of the plagiarism by measuring the ratio of mutations to original lines.

6.2. Effect on Plagiarism Detection

Our most important goal, 1.1, is to increase the efficacy of plagiarism detection. For this reason, we have designed our three defense mechanisms to delete the largest possible amount of mutation types. In the following sections, we analyze how our prototypical implementation of these mechanisms in JPlag can improve plagiarism detection for Mossad plagiarism and what configurations work best. To increase readability, we will sometimes call the deletion of unused variables *A*, the deletion of tokens in unreachable sections *B*, and the generic token filtering *C*. The combination of two or more mechanisms is represented by the corresponding letters, separated by a plus, e.g., *A+C* means running JPlag with the detection of unused variables and generic token filtering enabled.
We used JPlag on each dataset separately, but we found that all assignmets behaved similarly. Therefore, we plot the pairs in the same diagram for each assignment to increase readability.

6.2.1. Effect of Using All Mechanisms

The simplest configuration is enabling all mechanisms at the same time. Figure 6.1 depicts the similarity scores of our three different pair types when running on the default version of JPlag and then again with all our mechanisms enabled. Most non-plagiarism pairs (66.4%) have a similarity of 0%, so we omit them from the plot. As stated above, only pairs where one is the original and the other is its variant are labeled as Mossad pairs. We, therefore, do not omit any 0% Mossad pairs because each pair labeled as Mossad is a case of plagiarism.

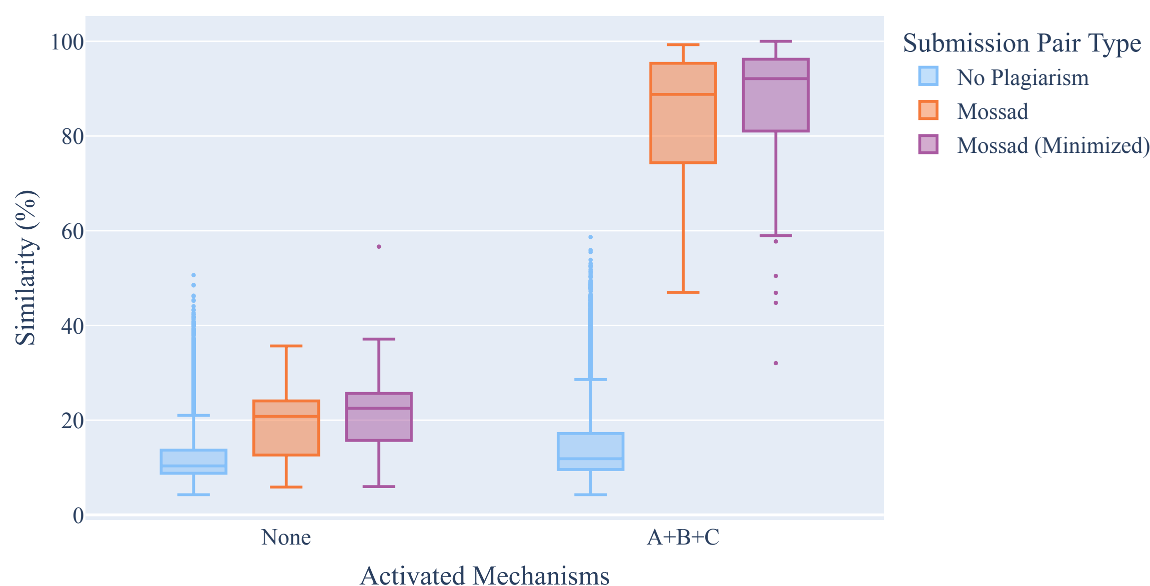

Figure 6.1.: Effect on similarity when using no mechanisms compared to using all mechanisms. Non-plagiarism pairs with 0% similarity are omitted.

It quickly becomes apparent that both Mossad plagiarism types' similarity scores have been greatly increased. The median similarity of default and minimized Mossad plagiarism pairs

has increased from 20.79% and 22.51% respectively to 88.68% and 92.12%, with individual pairs now reaching up to 100%. The similarities' deviation has increased overall, indicating that not all types of code insertion are detected. Nevertheless, the similarity of Mossad plagiarism was generally above 50%, except for a few outliers. We inspected the outliers that can be seen and found that the original source files taken from the dataset [18] described in Section 6.1.1 contained so many syntax errors that GCC was unable to detect any unused variables. We think it is reasonable to assume that this poses no issue for practical use, as plagiarising a broken submission is of no use to a plagiarist trying to pass an assignment.
Because students want to be as unsuspicious as possible, we assume that a minimized Mossad plagiarism is more likely to be submitted. The similarity increase when minimizing is relatively low without using any mechanisms. When all mechanisms are enabled, the minimum similarity for minimized Mossad plagiarism is about 10 percent of similarity higher than for unaltered Mossad plagiarism. These results show that programming instructors would likely find current Mossad plagiarism when the creation is performed using a JPlag version without our mechanisms.
Ideally, our mechanisms should not affect the similarity of non-plagiarism pairs. We can see that this is mostly the case, but there is a minor similarity increase if a pair already had a higher similarity score without our mechanisms enabled. As a result, we can see an overlap between non-plagiarism and unaltered Mossad plagiarism. Overall, we believe this effect is negligible compared to the drastic increase in similarity for the Mossad pairs.

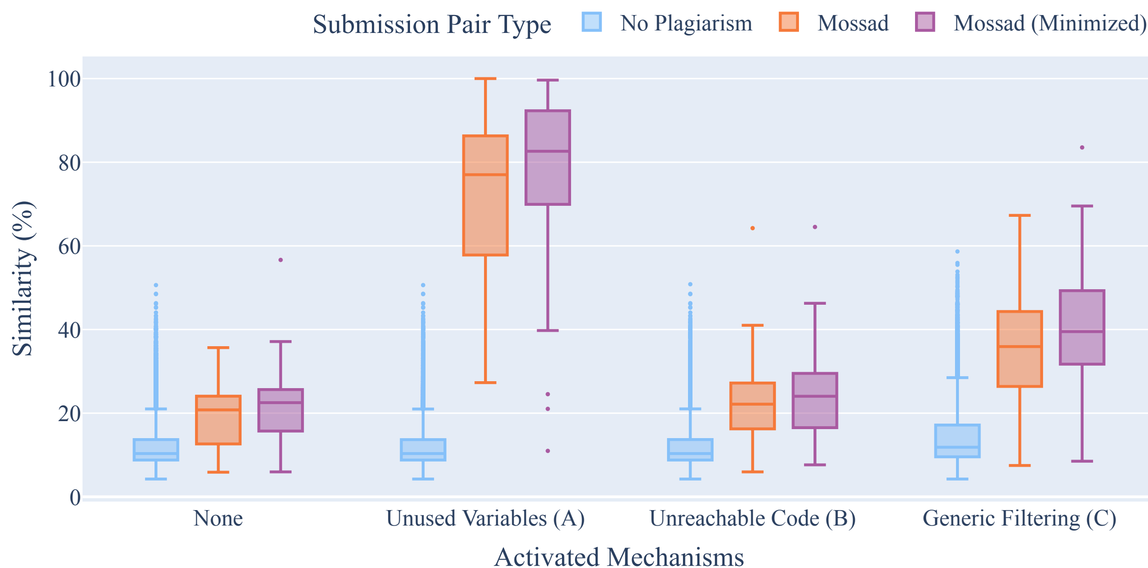

Figure 6.2.: Effect of the single defense mechanisms on the similarity score for the different types of pairs. Pairs with 0% similarity are omitted.

6.2.2. Impact of the Specific Mechanisms

As we have seen in the previous section, the combination of deleting unused variables, deleting unreachable code, and our custom generic token filtering algorithm drastically

increases the similarity of Mossad plagiarism pairs. This begs the question if all the mechanisms evenly contributed to this result, if some are more effective than others, or if they only work together. Looking at the dataset, we suspected that the deletion of unused variables would have the biggest impact, as a large portion of the mutations is variable declarations. This might be different for other plagiarism generators that choose their code insertions differently. We also know that the generic token filtering algorithm only works with a limited amount of inserted tokens and should therefore have little effect in isolation.
Figure 6.2 mostly confirms our assumptions by showing the similarity assigned to pairs of different plagiarism types when only one mechanism is enabled. Like above, non-plagiarism pairs with 0% similarity are omitted. By far the most effective mechanism is the deletion of unused variables. Compared to when all mechanisms are enabled, the median scores for Mossad and minimized Mossad plagiarism lies only 6.73 and 2.13 percent of similarity lower, respectively. Consequently, the deletion of unreachable code has almost no impact, only raising the median by 1.36 and 1.53 percent of similarity. We can see that the generic token filtering causes a moderate increase in plagiarism similarity, raising the median from 20.79% and 22.51% to 35.92% and 39.5%. However, the similarity of non-plagiarisms is more affected than by the other mechanisms, albeit by a small amount. The spread of plagiarism similarities is fairly wide for unused variable deletion and generic token filtering, with their minimum and maximum lying over 60% apart.

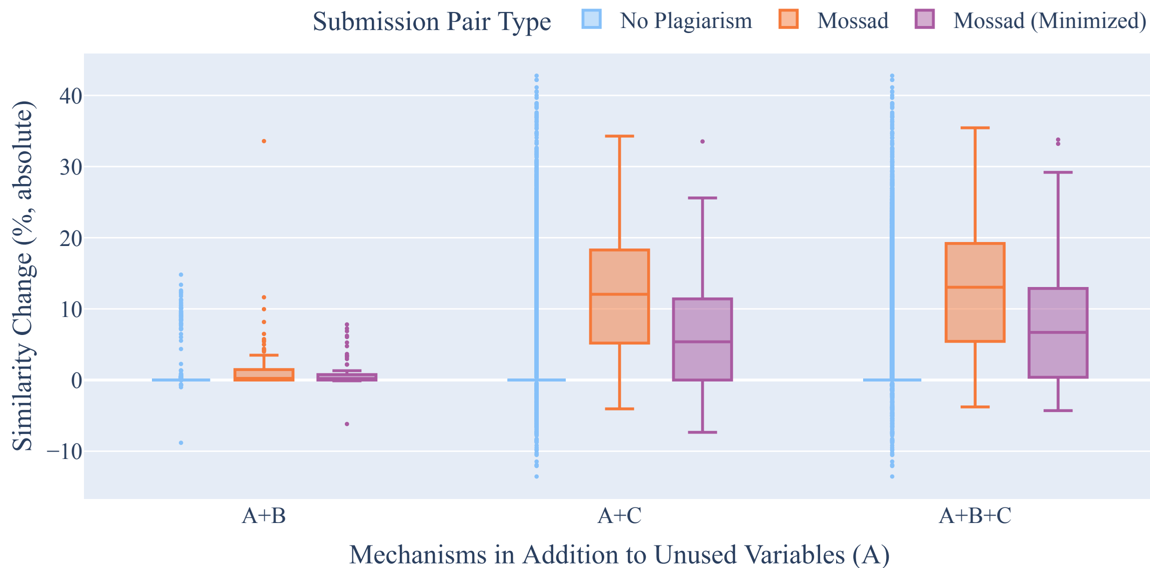

Figure 6.3.: Additional contribution of other mechanisms after deleting unused variables.

These results show that the deletion of unused variables is the major factor in detecting Mossad plagiarism when all mechanisms are enabled. We will now examine each mechanism's contributions when they are used together. Figure 6.3 shows the change in similarity when enabling unreachable code deletion and generic token filtering in addition to unused variable deletion.
Deleting tokens in unreachable sections has almost no effect on plagiarism detection when activated in addition to deleting unused variables. Because unused variables are likely

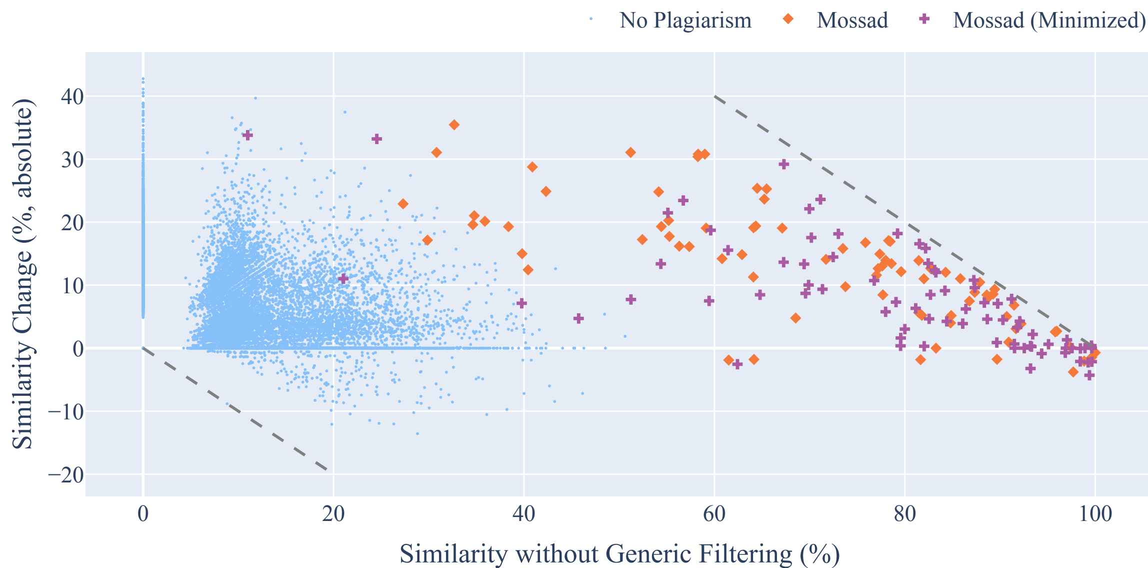

Figure 6.4.: The similarity changes generic token filtering causes when used in addition to deleting unused variables. Almost all similarity decreases for Mossad pairs occur when the pair is already above 80% similarity. The dashed lines mark 0% and 100% similarity after generic token filtering is used.

to be successful mutations, they are the most common type of code insertion Mossad adds to the entropy file. As a result, a large number of line insertions in unreachable sections are also unused variables and consequently covered by the deletion of unused variables. This reduces the positive impact of unreachable code detection. However, there is almost no negative effect either. In one case, this mechanism was crucial in detecting a Mossad plagiarism by raising its pair's similarity from 32.61% to 77.37%. The similarity stayed at 32.61% for the other mechanisms, which proves that there are cases where this mechanism is essential in detecting the plagiarism. During a manual inspection of the plagiarism in question, we found that most mutations were chains of multiple `break` or `return` statements, which only this mechanism can detect. We cannot answer whether students would actually submit plagiarism containing multiple `break` or `return` statements in a row because these can be instantly spotted during a manual inspection. Nevertheless, this mechanism has no noticeable runtime overhead and minimal negative worst-case effects.

Enabling generic token filtering in addition to deletion of unused variables shows much higher increases in similarity for Mossad plagiarism while not affecting the vast majority (77.83%) of non-plagiarism pairs. Figure 6.4 shows the change in similarity when enabling generic token filtering depending on the initial similarity. Before using generic token filtering, there is an overlap between plagiarism and non-plagiarism at 20% to 40%. We can see that in this section, the similarity of Mossad plagiarisms is increased by a higher amount than that of non-plagiarisms. As a result, the similarity overlap between non-plagiarism and plagiarism is reduced. We suspect the validity of this finding depends on the dataset and could be different for higher non-plagiarism baseline similarity. Therefore, the

positive impact of generic token filtering should be reevaluated on more diverse datasets in the future.

6.2.3. Generic Token Filtering Impact on Non-Plagiarisms

Our generic token filtering algorithm introduced in Section 4.3 searches for tokens that can be deleted to create a larger matching section. This relies on the assumption that accidental similarities are rare enough that actual code insertion removals outweigh the wrongly deleted tokens. To evaluate the validity of this assumption, we first examine the frequency of code deletions by our algorithm within pairs of non-plagiarized submissions. Then we set this in relation to the impact it has on the similarity of the pairs.

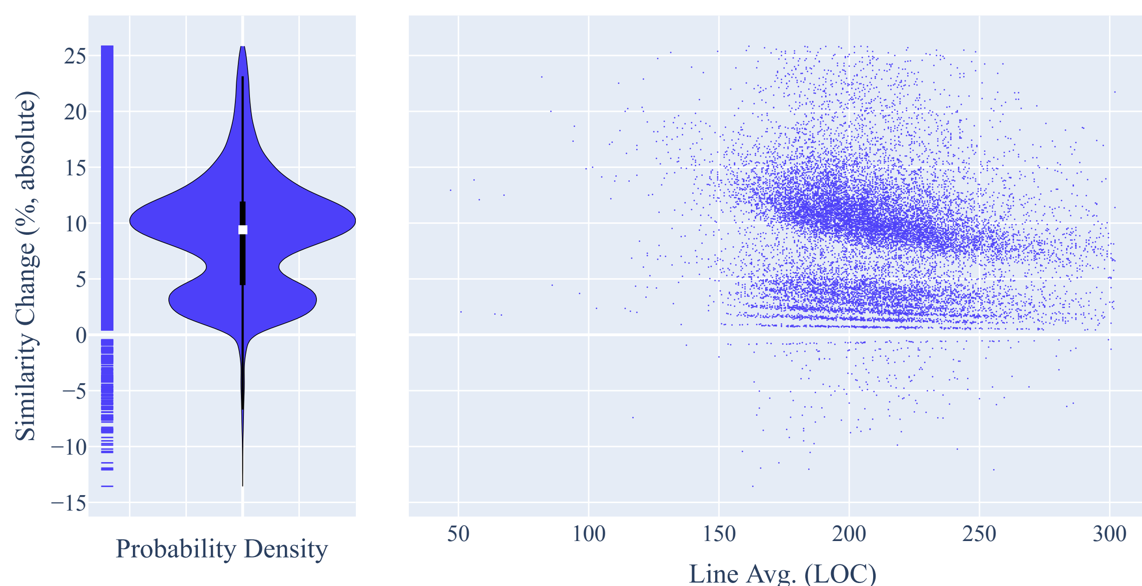

Figure 6.5.: Similarity change of non-plagiarisms depending on LOC. The points fall into groups depending on the number of new matches.

Out of all the non-plagiarism submission pairs, the vast majority (91.1%) are unaffected by the generic token filtering when any combination of mechanisms is enabled. Of the remaining 8.9%, the mean absolute change was 4.16% with a 2.0% median. Nevertheless, the maximum similarity changes were up to 37.36%, from a baseline value of 0%. A manual inspection shows no signs of plagiarism, which would have implicated a wrongly labeled pair. This shows that, although rare, some pairs can experience multiple token deletions by the generic token filtering algorithm purely by chance. However, there was not one case where JPlag assigned a non-plagiarism pair a similarity high enough that it could be considered a false positive.
Figure 6.5 shows a plot of the similarity change in non-plagiarism pairs when using generic token filtering depending on lines of code in the submission files. The x-axis represents a pair's average lines of code, while the y-axis shows the similarity increase. The two plots allow the following conclusions:
The frequency of similarity increases can be separated into two groups, as more easily visible in the violin plot. A submission pair falls within the group at around 2.5%, when no

new matches are found due to the token filtering. This can happen when the surrounding tokens are already part of two different matches. The change in total tokens then causes the similarity change.
The assumption that similarity changes in this group are not the result of new matches is further supported by Figure 6.4. Here we can see that pairs that had 0% similarity before have no similarity increases below 5% because a change in total token number has no effect when there are no matching tokens. Should the removal of the token create a new match that was not previously part of another match, the similarity increases more drastically by around 12% Notably, there is also a group forming, albeit much less dense on the negative side. This is because our generic token filtering is designed only to remove a token if its removal creates an identical sequence longer than the minimum token match. As a result, even if the token removal breaks an existing match, a new one is generally found in its stead. In some rare cases, however, both token sequences are already part of other matches. As a result, removing a token from the first sequence breaks up its match but does not form a new one because the second sequence is already matched to another subsequence. The total number of matched tokens decreases and, consequently, the similarity.
When no new matches are created, distinct parallel lines can be seen in the scatter plot around the x-axis. These lines mark the similarity change caused by removing one or two tokens from the token sequence. Although these deletions do not create new matches, they alter the total number of tokens and can increase or decrease the relative similarity by fixed increments. Because the impact of a single token removal becomes smaller with increasingly larger files, these lines converge towards the x-axis.

6.3. Effect on New Plagiarism Generation by Mossad

Mossad creates plagiarism by indeterministically inserting lines, checking for semantic equivalence, and then checking the similarity score by calling the plagiarism detector it tries to defeat. This means that unless every single type of code insertion can be detected, Mossad can keep adding new lines until the similarity is low enough. That is why examining how our mechanisms impede Mossad's plagiarism creation process is crucial. We measure the time it takes to create plagiarism and assess its quality.
To reduce external factors and increase comparability, we measure the number of iterations in Mossad's main loop in addition to the actual execution time. Mossad inserts one line during each iteration and assesses whether it is successful. In simpler terms, the iterations are the number of line insertions Mossad has to try before the plagiarism is finished. Evaluating iterations instead of execution time eliminates the impact of external factors like other processes and increases comparability between systems. Nevertheless, we will assess the applicability based on the time it took to create a Mossad variant on our system.
To establish a baseline, we measured the iterations Mossad needs to create plagiarism without calling JPlag with our defense mechanisms enabled. We then enabled our mechanisms and re-ran Mossad. Figure 6.6 shows the results we measured, differentiating between a successful run and terminations because of uncaught exceptions. With our defense mechanisms enabled, Mossad generally takes an order of magnitude more iterations to create plagiarism. The exact amount of additional iterations is mostly random because of

the indeterministic nature of Mossad. However, the worst case took more than 10 times more iterations than the maximum iterations in our baseline.

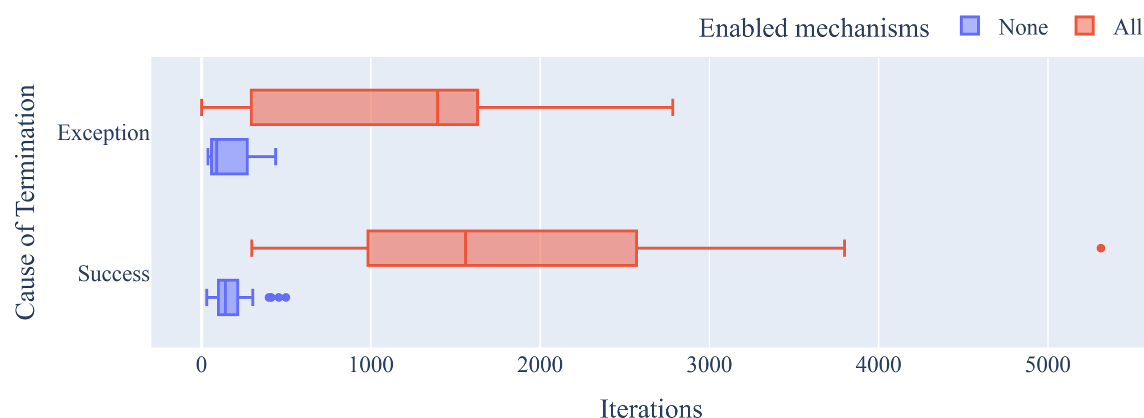

Figure 6.6.: Iterations needed for successful plagiarism or until an unhandled exception within Mossad occurs.

We examined the plagiarisms created and found that they are practically unusable without major additional effort. The reason for this is the drastic size increase, ranging from 40% up to 4000% in one case where Mossad kept adding lines into a block comment. Even without these extreme cases, the median size increase is 322.5%. Additionally, we found that Mossad frequently fails entirely to create a plagiarism and instead throws an unhandled exception. This termination generally takes a long time to occur, and the user has to restart Mossad. Figure 6.7 shows the runtimes we measured on an AMD Ryzen 7 3700X CPU. We generally had to run Mossad multiple times before a plagiarism was created. Even if Mossad terminates successfully, the plagiarizer must invest much time dissecting which code insertions are required and which they can delete. Overall we suspect the additional effort and time investment for plagiarism creation is higher than solving the assignment.

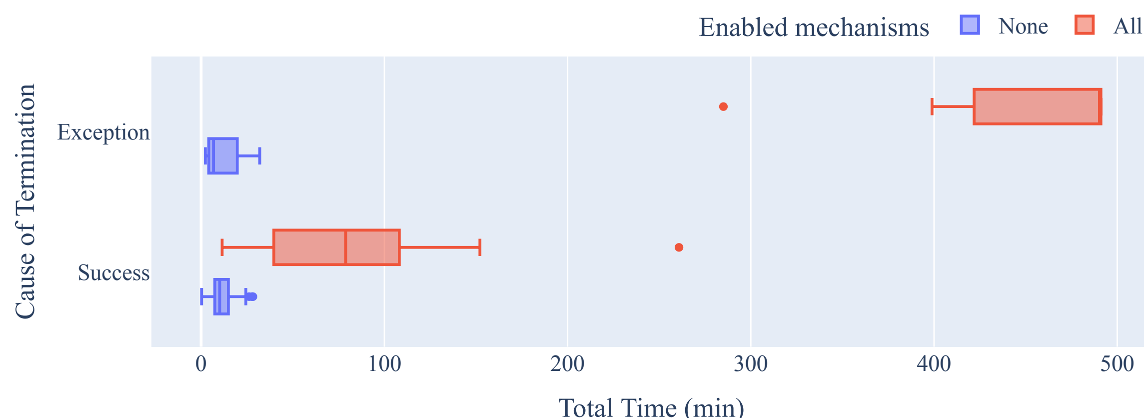

Figure 6.7.: Total execution time needed for a successful plagiarism or until an unhandled exception within Mossad occurs. Measured on a Ryzen 7 3700X Processor.

6.4. Threats to Validity

Our evaluation is subjected to limitations that might threaten its validity. Our dataset is fairly limited because, to our knowledge, there are no well labeled C++ datasets available in literature. As we mentioned in Section 6.1.1, the dataset by Ljubovic and Pajic [19] did not provide a reliable ground truth. As a result, we had to create our own ground truth. Therefore, we may have missed some cases of plagiarism, which we labeled as no plagiarism. Mislabeled pairs only have a small effect on our evaluation, only increasing the higher end of non-plagiarism similarities. The Mossad pairs, which are the most important part of our evaluation, are labeled correctly, so the central argument of our evaluation remains valid.

Another possible threat to validity concerning the dataset is the risk of possible overfitting. We only take our dataset from one source [18] due to a lack of suitable datasets. This poses the risk that our mechanisms perform worse on other datasets, decreasing the value of our evaluation. We reduced this risk by designing our mechanisms mostly independently from our dataset. Additionally, we used four different assignments, containing a total of 709 original submissions and 148 Mossad plagiarisms, resulting in a total of 80.421 submission pairs. This high number of submissions taken from different courses and years should counteract possible bias within the dataset.

During the creation of the Mossad plagiarisms, we relied on the default entropy file and the additions added by Mossad from other files. This poses the threat that other entropy files could circumvent our defense mechanisms. In Section 3.3, we assume that students who lack the skill or time to solve the assignments are also unable to create a better entropy file. This leaves the option of using an entropy file created by a third person, which could be published on the internet. Mossad is not publicly available, so to our knowledge, no file of this type exists at the moment. In its current form, Mossad has severe limitations for multiline mutations and does not support semantic context between mutations. This is why we assess the risk of powerful universal entropy files in the future as low.

Like all prototypical implementations, it is possible that our mechanisms contain unhandled edge cases or flaws that plagiarizers could exploit. Should plagiarizers find a type of code insertion our mechanisms cannot find, they can add them to the entropy file. Therefore it is important to recognize that our implementation is mainly for theoretical evaluation of the applicability of our mechanisms. In future implementations, new mechanisms can be included to find more types of code insertions and edge cases can be fixed.

7. Related Work

While our work aims to make existing software plagiarism detectors more robust to obfuscation attacks, others have done research on entirely new types of detectors. These are generally prototypes built from the ground up and impractical to transfer to existing plagiarism detectors.

GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis [17] introduces GPlag, a software plagiarism detector based on program dependence graphs (PDGs). A PDG is a graph representation of a program, depicting the control and data dependencies between basic statements [10]. The graphs are then checked for identical subgraphs, and a similarity score is calculated. The usage of a PDG can be seen as a generalization of the deletion of unreachable code and unused variables. Because these code insertions do not change the semantics of the program, they have no dependencies on the rest of the program. Therefore, GPlag is resilient to these code insertion attacks by default.

Dead and unreachable code detection is mainly relevant for code compaction in compilers. Therefore, most work in this regard is done in the context of compilers [6]. However, any code insertion that does not affect the program's execution is, by definition, dead. Wang et al. [30] provide a method for detecting dead code using program slicing. The dead code includes unused variables and the unreachable code we considered in this thesis, so their method is a valuable starting point for implementing a more sophisticated source analysis.

Mossad: Defeating Software Plagiarism Detection [7] is the work that challenges the working assumption that plagiarism creation can be automated and mainly motivated this thesis. Besides introducing Mossad as a plagiarism creation framework, the paper also analyzes the weakness of token-based plagiarism detectors, especially MOSS [1]. Furthermore, the authors suggest the comparison of assembly in addition to source files, which motivated our prototypical implementation. This is related to the work of Karnalim [15], who researched a bytecode-based approach to detecting plagiarism in Java files.

In addition to JPlag, many other similar token-based plagiarism detectors have been created using different tokenization and comparison approaches. We already mentioned MOSS [1] in Chapter 2. Nichols et al. [21] introduced a new token-based plagiarism detector utilizing parse trees for tokenization and the Smith-Waterman algorithm [26]. This algorithm specifically considers insertions and deletions during comparison and is, therefore, interesting for raising resilience against code insertions. Although Mossad inserts enough lines to defeat this comparison algorithm still, the Smith-Waterman algorithm could be an alternative to our generic token filtering.

8. Future Work

While our defense mechanisms can increase the difficulty of single-line code insertions, more sophisticated attempts at creating plagiarism using multiple semantically dependant lines will not be detected. Since a plagiarism must not alter the execution result of its original, any code inserted to create the plagiarism is by definition dead. This opens up opportunities for future improvements, especially in dead code detection. Although highly complex, implementing a sophisticated dead code detection and deletion should counteract most types of code insertions. Alternatively, concepts like GPlags [17] program dependence graph could be adapted for use with JPlag. As explained in Chapter 7, PDGs are invariant under the code insertions considered in this thesis.

Another major opportunity for future improvements is the token types. A prominent example is the `scope`, created whenever the namespace is explicitly stated in C++ code using the `::` keyword. We found that manually adding or removing explicit namespaces wherever possibly causes a significant amount of tokens inserted or deleted. The goal of tokenization is to be as invariant as possible under plagiarism creation. This is sufficient for comment modifications, whitespace insertions or removal, or modifier renaming. The previous example illustrates the need for further research on the best token choice.

Our evaluation was performed on plagiarism containing only plagiarism created using Mossad. However, this leaves the possibility that simple manual modifications to the plagiarism defeat our defense mechanisms. Ideally, there should be a comprehensive labeled dataset containing a variety of assignments containing original submissions and plagiarism created using various methods. To our knowledge, there currently is a lack of such a suitable dataset in literature. Creating it as a benchmark would be a helpful contribution to future research and enable better comparability between different research papers.

During our experiments, we found that Mossad utilizes a reasonably simple approach to plagiarism creation. For example, our observations of continuous mutations within block comments could be easily mitigated. Additionally, logically interconnected mutations could defeat a whole class of defense mechanisms by increasing the difficulty of dead code detection. Further improving plagiarism obfuscators, although potentially dangerous, would also help raise the resilience of plagiarism detectors.

9. Conclusion

To detect plagiarism among students, many programming instructors use token-based plagiarism detectors. To improve their resilience against code-insertion obfuscation attacks, we introduced three mechanisms: The first mechanism uses the source code to find unused variables and removes them from the token sequence. Our second mechanism looks for tokens in the language-specific token sequence that belong to unreachable code. The detection of unreachable code is limited to code after guaranteed jumps, like `return` or `continue`. Both of these mechanisms extend the tokenization phase. Our third mechanism acts as a pre-filtering step for the comparison algorithm. The filtering compares two submission's token sequences and tries to find subsequences that can be made equal by deleting a small number of sequential tokens. We call this pre-filtering algorithm *generic token filtering*. All these mechanisms could be added to token-based plagiarism detectors with only minor adjustments.
To evaluate the validity of our mechanisms, we first implemented them into JPlag, a state-of-the-art token-based plagiarism detector. We then extended a dataset from literature by adding automatically generated plagiarism using Mossad, a software plagiarism obfuscator. Mossad creates plagiarism by iteratively inserting single lines of code into an original submission of another student. To obtain the data for our evaluation, we implemented a framework for automatically running different versions of JPlag using different user settings on different datasets. Utilizing this framework, we ran JPlag on our dataset with different combinations of mechanisms activated.
Our results show a significant increase in detected code similarity when all mechanisms are activated. As a result, Mossad takes significantly more time to create plagiarism, and the readability of the resulting plagiarism decreases. Deleting unused variables has the most significant impact on plagiarism similarity increase in our dataset. The unreachable code detection is negligible for most plagiarism pairs but can be crucial in detecting plagiarism in certain cases. Our generic token filtering raises the similarity of non-plagiarism pairs more than we expected, but our results indicate that the increase is even higher for plagiarism pairs. As a result, the generic token filtering was able to separate plagiarism from non-plagiarism when there previously was an overlap. To further confirm our findings, we should evaluate this with other datasets.

Bibliography

[1] Alex Aiken. *MOSS: A System for Detecting Software Similarity*. http://theory.stanford.edu/~aiken/moss/. Accessed: 2022-05-02.

[2] The Editors of Encyclopaedia Britannica. *"plagiarism". Encyclopedia Britannica, 7 Nov. 2017*. https://www.britannica.com/topic/plagiarism. Accessed: 2022-02-05.

[3] Dong-Kyu Chae et al. "Software Plagiarism Detection: A Graph-Based Approach". In: CIKM '13. San Francisco, California, USA: Association for Computing Machinery, 2013, pp. 1577–1580. ISBN: 9781450322638. DOI: 10.1145/2505515.2507848. URL: https://doi.org/10.1145/2505515.2507848.

[4] Yih-Farn Chen, Emden Gansner, and Eleftherios Koutsofios. "A C++ Data Model Supporting Reachability Analysis and Dead Code Detection". In: vol. 1301. Apr. 2006, pp. 414–431. ISBN: 978-3-540-63531-4. DOI: 10.1007/3-540-63531-9_28.

[5] Debora Weber-Wulff, Katrin Köhler, Christopher Möller. *Collusion Detection System Test Report 2012*. https://plagiat.htw-berlin.de/collusion-test-2012/. Accessed: 2022-09-06.

[6] Saumya K. Debray et al. "Compiler Techniques for Code Compaction". In: *ACM Trans. Program. Lang. Syst.* 22.2 (Mar. 2000), pp. 378–415. ISSN: 0164-0925. DOI: 10.1145/349214.349233. URL: https://doi.org/10.1145/349214.349233.

[7] Breanna Devore-McDonald and Emery D. Berger. "Mossad: Defeating Software Plagiarism Detection". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428206. URL: https://doi.org/10.1145/3428206.

[8] Clang 16.0.0 git documentation. *Diagnostic flags in Clang*. https://clang.llvm.org/docs/DiagnosticsReference.html. Accessed: 2022-09-24.

[9] GCC Documentation. *GCC Warning Options*. https://gcc.gnu.org/onlinedocs/gcc-3.3/gcc/Warning-Options.html. Accessed: 2022-09-08.

[10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925. DOI: 10.1145/24039.24041. URL: https://doi.org/10.1145/24039.24041.

[11] Tomáš Foltýnek, Norman Meuschke, and Bela Gipp. "Academic Plagiarism Detection: A Systematic Literature Review". In: *ACM Comput. Surv.* 52.6 (Oct. 2019). ISSN: 0360-0300. DOI: 10.1145/3345317. URL: https://doi.org/10.1145/3345317.

[12] GCC. *GCC, the GNU Compiler Collection*. https://gcc.gnu.org. Accessed: 2022-09-24.

[13] JavaCC. *JavaCC Homepage*. https://javacc.github.io/javacc/. Accessed: 2022-09-07.

[14] JPlag. *JPlag GitHub Repository*. https://github.com/jplag/JPlag. Accessed: 2022-04-25.

[15] Oscar Karnalim. "Detecting source code plagiarism on introductory programming course assignments using a bytecode approach". In: 2016, pp. 63–68. DOI: 10.1109/ICTS.2016.7910274.

[16] Oscar Karnalim et al. "Source Code Plagiarism Detection in Academia with Information Retrieval: Dataset and the Observation". In: *Informatics in Education* 18 (Oct. 2019), pp. 321–344. DOI: 10.15388/infedu.2019.15.

[17] Chao Liu et al. "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis". In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. Philadelphia, PA, USA: Association for Computing Machinery, 2006, pp. 872–881. ISBN: 1595933395. DOI: 10.1145/1150402.1150522. URL: https://doi.org/10.1145/1150402.1150522.

[18] Vedran Ljubovic. *Programming Homework Dataset for Plagiarism Detection*. 2020. DOI: 10.21227/71fw-ss32. URL: https://dx.doi.org/10.21227/71fw-ss32.

[19] Vedran Ljubovic and Enil Pajic. "Plagiarism Detection in Computer Programming Using Feature Extraction From Ultra-Fine-Grained Repositories". In: *IEEE Access* 8 (2020), pp. 96505–96514. DOI: 10.1109/ACCESS.2020.2996146.

[20] LLVM. *Clang: a C language family frontend for LLVM*. https://clang.llvm.org. Accessed: 2022-09-24.

[21] Lawton Nichols et al. "Syntax-Based Improvements to Plagiarism Detectors and Their Evaluations". In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '19. Aberdeen, Scotland Uk: Association for Computing Machinery, 2019, pp. 555–561. ISBN: 9781450368957. DOI: 10.1145/3304221.3319789. URL: https://doi.org/10.1145/3304221.3319789.

[22] Lutz Prechelt and Guido Malpohl. "Finding Plagiarisms among a Set of Programs with JPlag". In: *Journal of Universal Computer Science* 8 (Mar. 2003).

[23] Timur Sağlam et al. "Token-Based Plagiarism Detection for Metamodels". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 138–141. ISBN: 9781450394673. DOI: 10.1145/3550356.3556508. URL: https://doi.org/10.1145/3550356.3556508.

[24] Saul Schleimer, Daniel Wilkerson, and Alex Aiken. "Winnowing: Local Algorithms for Document Fingerprinting". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* 10 (Apr. 2003). DOI: 10.1145/872757.872770.

[25] *Sherlock - Plagiarism Detection Software*. https://warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock/. Accessed: 2022-05-02.

[26] T.F. Smith and M.S. Waterman. “Identification of common molecular subsequences”. In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(81)90087-5. URL: https://www.sciencedirect.com/science/article/pii/0022283681900875.

[27] Rini van Solingen (Revision) et al. “Goal Question Metric (GQM) Approach”. In: *Encyclopedia of Software Engineering*. John Wiley & Sons, Ltd, 2002. ISBN: 9780471028956. DOI: https://doi.org/10.1002/0471028959.sof142. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471028959.sof142. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/0471028959.sof142.

[28] B. Stroustrup. *The C++ Programming Language: 4th Edition*. Always learning. Addison-Wesley, 2013. ISBN: 9780321563842. URL: https://books.google.de/books?id=L6MPBQAAQBAJ.

[29] Han Wan, Kangxu Liu, and Xiaopeng Gao. “Token-based Approach for Real-time Plagiarism Detection in Digital Designs”. In: *2018 IEEE Frontiers in Education Conference (FIE)*. 2018, pp. 1–5. DOI: 10.1109/FIE.2018.8658531.

[30] Xing Wang et al. “Dead Code Detection Method Based on Program Slicing”. In: *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. 2017, pp. 155–158. DOI: 10.1109/CyberC.2017.69.

[31] Michael J. Wise. “YAP3: Improved Detection of Similarities in Computer Program and Other Texts”. In: *SIGCSE Bull.* 28.1 (Mar. 1996), pp. 130–134. ISSN: 0097-8418. DOI: 10.1145/236462.236525. URL: https://doi.org/10.1145/236462.236525.

[32] Fangfang Zhang et al. “Program Logic Based Software Plagiarism Detection”. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014, pp. 66–77. DOI: 10.1109/ISSRE.2014.18.

A. Appendix

Project	Type	Count	LOC Avg.	LOC Median	Similarity Avg.
A	Original	64	100.45	85	0.29%
	MOSSAD (automatic)	5	259.6	184	23.27%
	MOSSAD (minimized)	4	145.75	123	36.38%
B	Original	189	140.35	135	2.79%
	MOSSAD (automatic)	19	196.16	212	19.56%
	MOSSAD (minimized)	18	195.89	173.5	21.45%
C	Original	162	140.22	136	2.59%
	MOSSAD (automatic)	18	209.06	194	16.85%
	MOSSAD (minimized)	15	195.33	187	18.16%
D	Original	294	137.02	134.5	2.44%
	MOSSAD (automatic)	35	177.46	173	17.61%
	MOSSAD (minimized)	34	190.44	183	20.78%

Table A.1.: Dataset