Intelligent Match Merging to Prevent Obfuscation Attacks on Software Plagiarism Detectors

Master's Thesis of

Nils Niehues

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

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

29. May 2023 – 29. Nov. 2023

I declare that I have developed and written the enclosed thesis completely by myself. I have submitted neither parts of nor the complete thesis as an examination elsewhere. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. This also applies to figures, sketches, images and similar depictions, as well as sources from the internet.

Karlsruhe, 29.11.2023

. . . . . .

(Nils Niehues)

Abstract

The increasing number of computer science students has prompted educators to rely on state-of-the-art source code plagiarism detection tools to deter the submission of plagiarized coding assignments. While these token-based plagiarism detectors are inherently resilient against simple obfuscation attempts, recent research has shown that obfuscation tools empower students to easily modify their submissions, thus evading detection. These tools automatically use dead code insertion and statement reordering to avoid discovery. The emergence of ChatGPT has further raised concerns about its obfuscation capabilities and the need for effective mitigation strategies.

Existing defence mechanisms against obfuscation attempts are often limited by their specificity to certain attacks or dependence on programming languages, requiring tedious and error-prone reimplementation. In response to this challenge, this thesis introduces a novel defence mechanism against automatic obfuscation attacks called match merging. It leverages the fact that obfuscation attacks change the token sequence to split up matches between two submissions so that the plagiarism detector discards the broken matches. Match merging reverts the effects of these attacks by intelligently merging neighboring matches based on a heuristic designed to minimize false positives.

Our method's resilience against classic obfuscation attacks is demonstrated through evaluations on diverse real-world datasets, including undergrad assignments and competitive coding challenges, across six different attack scenarios. Moreover, it significantly improves detection performance against AI-based obfuscation. What sets our method apart is its language- and attack-independence while its minimal runtime overhead makes it seamlessly compatible with other defence mechanisms.

Zusammenfassung

Aufgrund der steigenden Anzahl der Informatikstudierenden verlassen sich Dozenten auf aktuelle Werkzeuge zur Erkennung von Quelltextplagiaten, um zu verhindern, dass Studierende plagiierte Programmieraufgaben einreichen. Während diese auf Token basierenden Plagiatsdetektoren inhärent resilient gegen einfache Verschleierungen sind, ermöglichen kürzlich veröffentlichte Verschleierungswerkzeuge den Studierenden, ihre Abgaben mühelos zu ändern, um die Erkennung zu umgehen. Der Vormarsch von ChatGPT hat zusätzliche Bedenken hinsichtlich seiner Verschleierungsfähigkeiten und der Notwendigkeit wirksamer Gegenstrategien aufgeworfen.

Bestehende Verteidigungsmechanismen gegen Verschleierung sind oft durch ihre Spezifität für bestimmte Angriffe oder ihre Abhängigkeit von Programmiersprachen begrenzt, was eine mühsame und fehleranfällige Neuimplementierung erfordert. Als Antwort auf diese Herausforderung führt diese Arbeit einen neuartigen Verteidigungsmechanismus gegen automatische Verschleierungsangriffe namens Match-Zusammenführung ein. Er macht sich die Tatsache zunutze, dass Verschleierungsangriffe die Token-Sequenz ändern, um Übereinstimmungen zwischen zwei Abgaben aufzuspalten, sodass die gebrochenen Übereinstimmungen vom Plagiatsdetektor verworfen werden. Match-Zusammenführung macht die Auswirkungen dieser Angriffe rückgängig, indem benachbarte Übereinstimmungen auf der Grundlage einer Heuristik intelligent zusammengeführt werden, um falsch positive Ergebnisse zu minimieren.

Die Widerstandsfähigkeit unserer Methode gegen klassische Verschleierungsangriffe wird durch Evaluationen anhand verschiedener realer Datensätze, einschließlich Studienarbeiten und Programmierwettbewerbe, in sechs verschiedenen Angriffsszenarien demonstriert. Darüber hinaus verbessert sie die Erkennungsleistung gegen KI-basierte Verschleierung signifikant. Was diesen Mechanismus auszeichnet, ist seine Unabhängigkeit von Sprache und Angriff, während sein minimaler Laufzeit-Aufwand ihn nahtlos mit anderen Verteidigungsmechanismen kompatibel macht.

Contents

Abstract i

Zusammenfassung ii

1 Introduction 1
- 1.1 Contributions . . . 2
- 1.2 Outline . . . 2

2 Foundations 3
- 2.1 Token-based plagiarism detectors . . . 3
  - 2.1.1 Tokenization . . . 3
  - 2.1.2 Running example . . . 5
  - 2.1.3 Comparison . . . 5
- 2.2 Obfuscation attacks . . . 7
  - 2.2.1 Mossad . . . 7
  - 2.2.2 JPlag-GEN . . . 9

3 Types of obfuscation attacks 12
- 3.1 Semantic-preserving obfuscation . . . 12
- 3.2 Semantic-agnostic obfuscation . . . 14
  - 3.2.1 Alteration . . . 16
  - 3.2.2 Swapping . . . 18
- 3.3 Large language model obfuscation . . . 20
  - 3.3.1 Obfuscation . . . 20
  - 3.3.2 Fully generated . . . 22

4 Match merging 24
- 4.1 Idea . . . 24
  - 4.1.1 Neighboring matches . . . 24
  - 4.1.2 Merging . . . 25
  - 4.1.3 Position in pipeline . . . 26
- 4.2 Heuristic . . . 28
  - 4.2.1 Available decision basis . . . 28
  - 4.2.2 Information relevance . . . 28
  - 4.2.3 Definition of heuristic approach . . . 30
- 4.3 Algorithm . . . 31

5 Evaluation 36
- 5.1 Goal-Question-Metric plan . . . 36

5.2 Design . . . 38
5.2.1 Datasets . . . 38
5.2.2 Plagiarism generation . . . 38
5.2.3 Methodology . . . 41
5.3 Results . . . 42
5.3.1 Resilience against semantic preserving obfuscation attacks . . . 42
5.3.2 Resilience against semantic agnostic obfuscation attacks . . . 52
5.3.3 Resilience against AI-based obfuscation attacks . . . 57
5.3.4 Maintain performance of JPlag . . . 60
5.3.5 Ablation study . . . 62
5.3.6 Default hyperparameters . . . 65
5.4 Threats to validity . . . 66
5.4.1 Internal validity . . . 66
5.4.2 External validity . . . 67
5.4.3 Construct validity . . . 67
5.4.4 Reliability . . . 68
5.5 Limitations . . . 68

6 Related works **69**
6.1 Plagiarism detection . . . 69
6.1.1 Brute force token deletion . . . 69
6.1.2 Token sequence normalization . . . 70
6.1.3 Compiler optimization . . . 71
6.1.4 Semantics-based similarity comparison . . . 71
6.1.5 Common code deletion . . . 72
6.1.6 Various preprocessing techniques . . . 72
6.2 Bioinformatics . . . 73
6.2.1 Indel detection . . . 73
6.2.2 Needleman–Wunsch algorithm . . . 74

7 Future work **75**

8 Conclusion **77**

Bibliography **79**

List of Figures

2.1 Pipeline of plagiarism detector JPlag . . . 3
2.2 Example of assigning token types to lines of code . . . 4
2.3 Abstract syntax tree of an example program . . . 4
2.4 Running example with source code and tokens . . . 5
2.5 Demonstration of minimum token match . . . 6
2.6 Insertion into running example . . . 8
2.7 Effects of insertions on continuous matches . . . 8
2.8 Reordering of running example . . . 10
2.9 Effects of reordering on matches . . . 11

3.1 Detector pipeline with simulated attacks . . . 15
3.2 Applying alteration on running example . . . 17
3.3 Performing swapping on running example . . . 19

4.1 Neighboring matches example . . . 25
4.2 Demonstration of merging two neighboring matches . . . 25
4.3 Match merging in plagiarism detector pipeline . . . 27
4.4 Demonstration of heuristic on four examples . . . 30
4.5 Demonstration of match merging against obfuscation . . . 33

5.1 Summarized base version of Tic-Tac-Toe . . . 40
5.2 Similarities on the Soco-Dataset with Mossad plagiarisms . . . 43
5.3 Observing changes in Mossad's usability . . . 45
5.4 Similarities on Progpedia with insertion mode . . . 47
5.5 Similarities on Progpedia with reorder mode . . . 48
5.6 Similarities on Progpedia with combined mode . . . 49
5.7 Similarities on TicTacToe with insertion mode . . . 51
5.8 Similarities on Soco-Dataset with 10 percent changes . . . 53
5.9 Similarities on Soco-Dataset with 20 percent changes . . . 55
5.10 Similarities on Soco-Dataset with 30 percent changes . . . 56
5.11 Plotted base and merging similarities per prompt . . . 58
5.12 Similarities between fully-generated submissions . . . 59
5.13 Change in JPlag's runtime for different parameters . . . 61
5.14 Effects of using maximum gap size on Progpedia . . . 63
5.15 Effects of file boundaries merge on TicTacToe . . . 64

6.1 Example program dependency graph . . . 70
6.2 Overview of TransIndel . . . 73

6.3 Needleman-Wunsch algorithm on two genome sequences 74

List of Tables

5.1 Numerical base and merging similarities per prompt 58
5.2 Best parameter combination per test set 65

1 Introduction

Undergrad computer science students face a lot of coding assignments ranging from simple recreation of mathematical functions to feature-rich applications. Because time is short and students often have access to either online solutions or solutions from their peers many are tempted to plagiarize another student's solution [11, 24]. As the amount of computer science students steadily rises [25], educational personnel cannot feasibly check every submission by hand and must instead rely on plagiarism detection tools, which automatically compare submissions and assign each pair a similarity score. Most popular among them are Moss [23] and JPlag [13], with the former having over 300.000 registered users [9, 31]. State-of-the-art plagiarism detectors compare submissions based on their structure rather than lines of code [31, 28]. To this end, submissions are parsed and specific nodes are collected from the parse tree. The nodes, such as a variable definition or an assignment, serve as tokens representing the submissions and an algorithm marks matches between both token sequences. This abstraction makes them inherently resilient against low-effort obfuscation, like renaming or changes to whitespace [12].

Nevertheless, automatic obfuscation tools empower students to generate customized copies of original submissions, effortlessly evading plagiarism detectors. Notable among these tools are *Mossad*, developed by Devore-McDonald and Berger [9], and *JPlag-GEN*, created by Brödel [4]. Token-based plagiarism detectors such as *Moss* and *JPlag* incorporate a minimum token match threshold, designed to filter out common one-liners or short snippets, preventing an excessive number of false positives. Mossad exploits this vulnerability by strategically inserting lines of dead code into submissions, which breaks up matching sequences into smaller subsequences. If the remaining matches fall below the threshold, they are ignored by the detector. JPlag-GEN builds upon Mossad's approach by incorporating the reordering of independent statements, further disrupting matches. Both attacks significantly compromise the detection capabilities of Moss and JPlag [9, 4, 17].

Another prevalent thread is the increasing popularity of large-language models like ChatGPT which are often misused by students for academic purposes [8, 18]. In the context of source code plagiarism detection, they can be used to obfuscate existing submissions or generate new ones from a task description [22, 2].

Defending against automatic obfuscations is a challenging task as it requires increasing the similarity of plagiarized submission pairs while leaving genuine pairs (mostly) unaffected. Most existing defences suffer from one of two setbacks, being attack-specific or language-specific. Attack-specific defences perform very well against the attack they were made for but are ineffective against unanticipated attacks. A notable example is *token sequence normalization* [4] a defence mechanism against JPlag-GEN. Language-specific defences are only applicable for the respective language and re-implementation for different languages is tedious and error-prone [3, 14].

This thesis investigates the possibility of developing a defence mechanism against obfuscation that is language- and attack-independent. To this end, we utilize the fact that all obfuscation attacks break up continuous matches between two submissions until the remaining matches fall below the minimum token match threshold and are discarded. Our approach is to merge the broken matches back together to revert changes from the obfuscation tools and thereby raise the similarity. To decide which matches should be merged we developed a heuristic which among other things relies on the lengths of remaining matches and gaps between them. The heuristic aims to keep false positives as low as possible. The idea of aligning matches between two sequences where one was subject to many small changes was conceptualized by Krieg [16] and also finds use in bioinformatics, where it is used for *InDel* detection [41].

To evaluate the performance of our approach we analyze the similarities of plagiarism and unrelated solutions on four distinct datasets. The datasets vary in languages, containing programs written in *Java* and *C++*, ranging in lengths from 80 to 400 lines of code and differ in origin, from competitive coding to undergrad assignments. From these datasets, we generate plagiarism using Mossad and JPlag-GEN and additionally four novel attacks, two of which utilize large-language models for obfuscation. The results of our evaluation demonstrate the effectiveness of our approach. The diversity in datasets and obfuscation attacks further illustrates that our approach is language- and attack-independent. Finally, an exhaustive search in the parameter space of the heuristic yields optimal default values.

1.1 Contributions

There are two main contributions of this thesis:

1. We designed match merging, an approach for defending against automatic obfuscation attacks.

2. We extensively evaluated our approach to demonstrate its effectiveness, attack-independence and language-independence.

1.2 Outline

The thesis is structured as follows. In chapter 2 we cover the necessary foundations, explaining how the components of a plagiarism detector function and how Mossad and JPlag-GEN circumvent their defences. In chapter 3 we propose four novel attacks against which our defence mechanism will be evaluated. We additionally provide categorization and threat models for each. chapter 4 deals with the conception of match merging, going over the core idea, decision basis and algorithms. We extensively evaluate our approach in chapter 5. Finally, we discuss related works in chapter 6, future work in chapter 7 and conclude this thesis in chapter 8.

2 Foundations

In this chapter, we establish the necessary foundations for the thesis, split into two parts. The first discusses the functionality of token-based plagiarism detectors with special attention to the tokenization step and similarity scoring. The second part delves into two state-of-the-art obfuscation attacks on token-based plagiarism detectors explaining their effectiveness and limitations.

2.1 Token-based plagiarism detectors

State-of-the-art plagiarism detectors like JPlag [13], Moss [23] or Dolos [20] generally consist of two steps for measuring similarity between submissions. The first step is preprocessing the submissions, which is called tokenization. How it works and why it is helpful is presented in the next section. Afterwards follows the actual comparison of submissions, with regard to how they are performed and which metric is used to describe similarities.

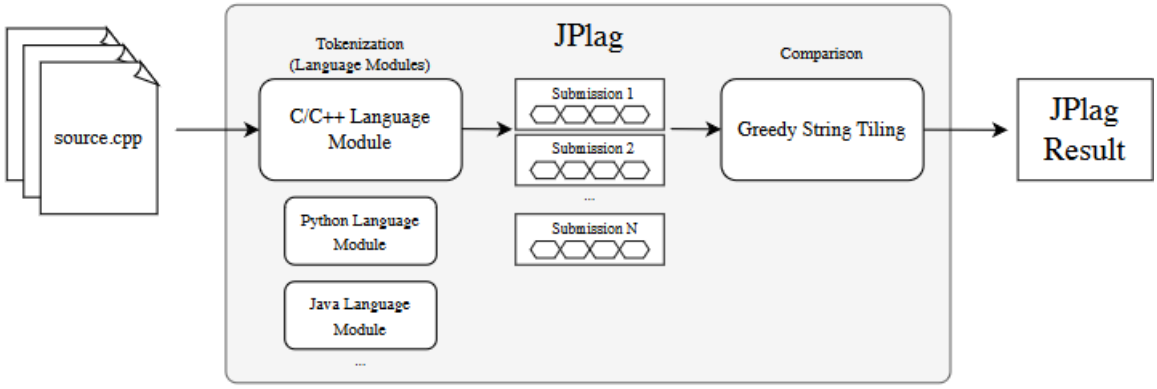


Figure 2.1: Pipeline of plagiarism detector JPlag [16]

Figure 2.1 depicts how these two components build a plagiarism detector pipeline, here JPlag.

2.1.1 Tokenization

Tokens are a layer of abstraction for submissions. Instead of comparing actual lines of code to each other, submissions are compared based on token types representing their structure [33, 42]. Each line of code maps to one or more token types. Figure 2.2 shows 3 example lines with their corresponding token:

```
int x = 1;                    VAR_DEF
x += 5                        ASSIGN
System.out.println(x)         APPLY
```

Figure 2.2: Example of assigning token types to lines of code

Comparing submissions based on sequences of (abstract) tokens has multiple advantages, as they are invariant to lexical changes [12] and retyping:

- renaming: $int\ x \cong int\ y$
- retyping: $int\ x \cong double\ x$
- switching of arguments: $func(x, y) \cong func(y, x)$
- bracketing
- comments
- whitespace

All these obfuscation methods could deceive a human but have no effect on the plagiarism detector. Submissions are compared on structure alone, ignoring nuances in implementation and stylistic choices [33]. The downside of tokenization is that short submissions or submissions with very little degree of freedom in structure, tend to be more similar to each other than they would be otherwise. This could seldom lead to false positives but this is an accepted risk for many plagiarism detectors as checks by humans are also susceptible to short submissions [35].

Let us take a look at how source code gets processed into token sequences.
For each supported language the plagiarism detector needs a specific language module. Its task is to parse the submissions and build an abstract syntax tree for them [39]. Figure 2.3 shows an example AST which contains a short method.

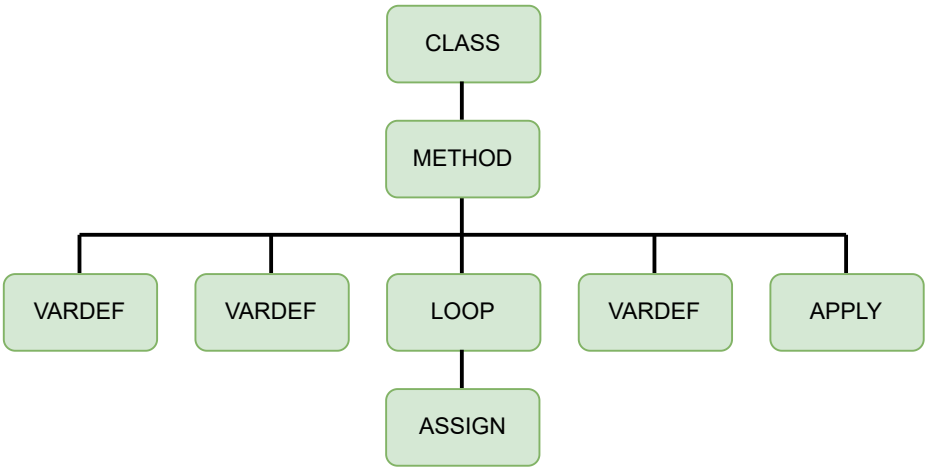


Figure 2.3: Abstract syntax tree of an example program

From this, a tree-walker would gather the selected nodes and their types are directly mapped to tokens. As syntax trees are highly language-specific, the token types for all languages are different too. This tokenization process adds two constraints for submissions:

1. Only parsable submissions can be compared
2. Only submissions of the same language can be compared

2.1.2 Running example

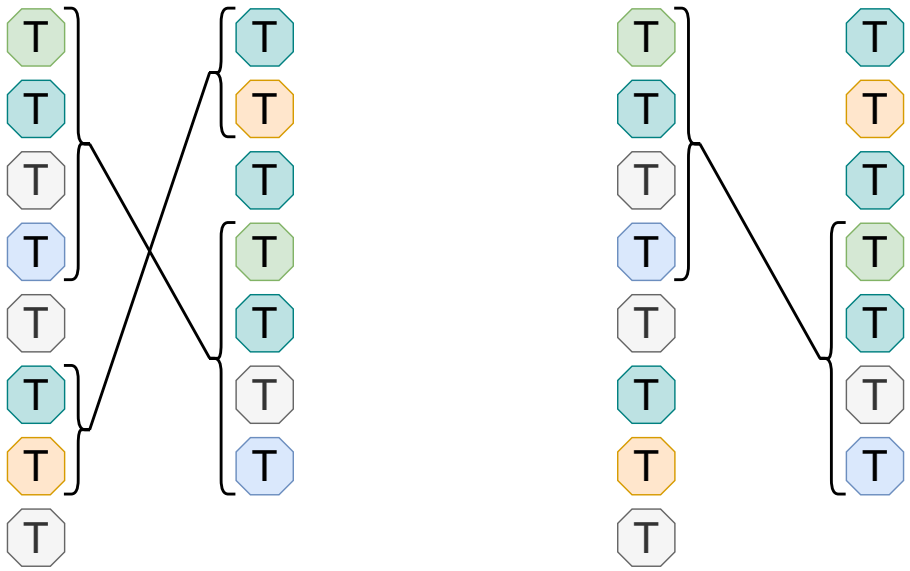
With tokenization discussed, we can define the running example for this thesis. The left side of Figure 2.4 shows a short Java method which is based on a typical first-semester code comprehension task. It increments one number by 3 until it is larger than 10, subtracts another number from it and prints the result. The right side shows the corresponding token sequence which was gathered from Figure 2.3 and is later used for similarity comparisons.

```
void calc() {                          BEGIN_METHOD
  int a = 1;                           VAR_DEF
  int b = 2;                           VAR_DEF
  while (a < 10) {                     BEGIN_LOOP
    a += 3;                            ASSIGN
  }                                    END_LOOP
  int result = a - b;                  VAR_DEF
  System.out.println(result);          APPLY
}                                      END_METHOD
```

Figure 2.4: Running example with source code and tokens

2.1.3 Comparison

With the preprocessing of submissions done, we can turn to the similarity comparisons. Each submission is compared with every other resulting in a total of $\frac{N\times(N-1)}{2}$ pairwise similarity scores. One commonly used comparison algorithm is *Greedy String Tiling* [40] created by Wise in 1993. The algorithm receives two (token)sequences a and b and returns all matching subsequences between the two. As each token can only be part of one match, Greedy String Tiling computes from the longest match to the shortest. This maximizes the amount of matching tokens. Another relevant comparison algorithm is *Winnowing* [37] which uses a similar approach to assess the similarity between two token sequences and is also based on subsequence matching. Most state-of-the-art plagiarism detectors rely on either Greedy String Tiling or Winnowing [31, 20].



(a) Unrestricted matches (b) Filtered matches

Figure 2.5: Demonstration of minimum token match

Figure 2.5a shows the matches produced by Greedy String Tiling on two example submissions. For simplicity, the token types are represented by a color, meaning that tokens of the same color share the same type. Each match is denoted by brackets which means that the subsequences are identical for both. One can see that Greedy String Tiling can mark matches regardless of the respective position in the submission, which in itself makes it resilient against reordering.

There are many different metrics to define the similarity score of a submission pair and they vary from plagiarism detector to plagiarism detector [6]. Following JPlag and Dolos we define the similarity score of a comparison pair as the harmonic mean of the ratio of matching tokens in the left- and in the right-submission. Applying this method to Figure 2.5a results in a similarity score of approximately 70.5%, calculated as follows:

$$\frac{2 * (6/9) * (6/8)}{(6/9) + (6/8)} = \frac{12}{17} \approx 70.5\%$$

Another important part of match-making is the minimum token match (MTM). It describes how long a match must be, in order to be considered by Greedy String Tiling. Without it, even single tokens of the same type would match. As almost all submissions contain some variable definitions or control structures this would drastically increase similarity. Figure 2.5b shows the same two submissions but now with the minimum token match set to 3. Because of this the matching block of length 2 is ignored and the similarity now is:

$$\frac{2 * (4/9) * (4/8)}{(4/9) + (4/8)} = \frac{8}{17} \approx 47.0\%$$

The minimum token match is a hyperparameter and must be chosen for each language. It controls the sensitivity of the comparison algorithm, if set too low, it increases the likelihood of false positives, while setting it too high may result in lower scores for instances of plagiarism [36]. Commonly used values are 9 for *Java* and 12 for *C++* [13].

Plagiarism detectors based on Winnowing also depend on a configurable parameter to set a minimum threshold to avoid excessive false positives [20].

2.2 Obfuscation attacks

2.2.1 Mossad

In their paper, Devore-McDonald and Berger [9] proposed Mossad, an automatic plagiarism attack for *C++* submissions. Their goal was to show flaws in popular plagiarism detectors like *Moss*, *JPlag* and *Sherlock* by creating a tool that makes plagiarism effortless and low-risk. Mossad aims to reduce the similarity between original and plagiarism by inserting lines to the latter. Lines are either taken from an entropy file or already inserted lines. After every potential insertion, it checks whether the program still compiles and if yes the results are the same. If a candidate line passes, it gets inserted and the new similarity gets computed. For this Mossad needs access to the plagiarism detector it wants to fool, for example, JPlag. After each insertion, the detector is run on the altered submission and returns the new similarity score. This loop is performed until a threshold is reached typically 25%, as scores this low are likely to be ignored. Formally Mossad works as follows:

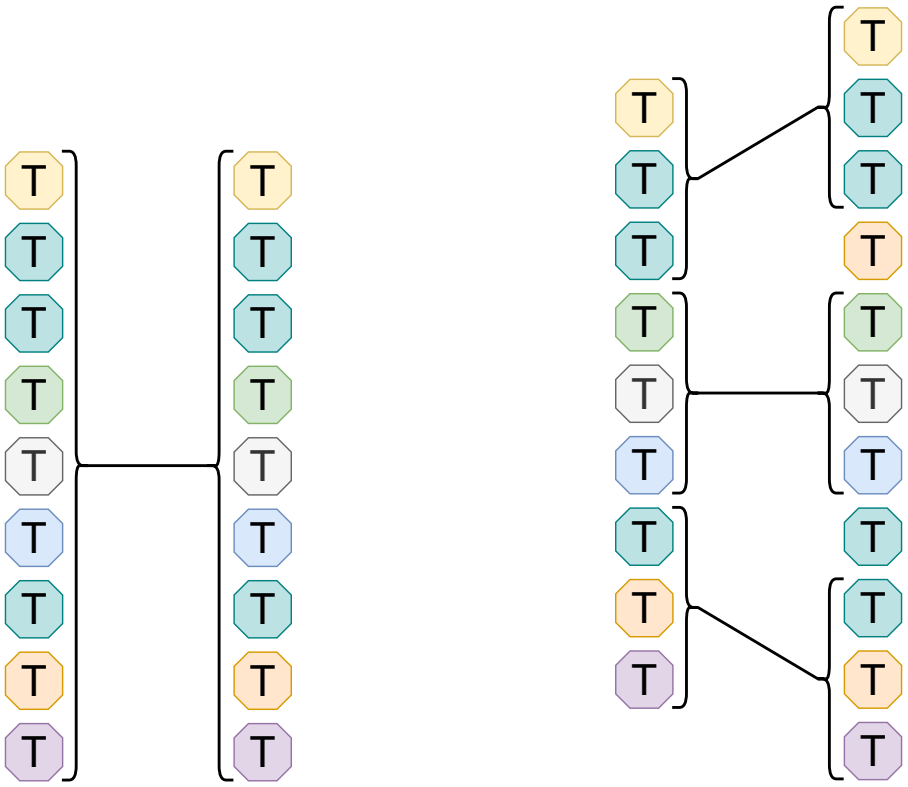
Algorithm 1 Mossad

$plagiarism \leftarrow original$
while $similarity(plagiarism, original) > threshold$ **do**
  do
    $candidate \leftarrow random(dictionary \cup original)$
    $spot \leftarrow random(0, |plagiarism|)$
    $plagiarism' \leftarrow insert(plagiarism, candidate, spot)$
  while not $compiles(plagiarism')$ **or** $output(plagiarism') \neq output(plagiarism)$
  $plagiarism \leftarrow plagiarism'$
  $dictionary \leftarrow dictionary \cup candidate$
end while

As the inserted lines must not change the output, they are often definitions of dead variables or print statements, which map to VAR_DEF and APPLY tokens. By storing successful insertions in the entropy file, their probability of being chosen increases over time, making Mossad faster with each submission. Applying Mossad on the running example could produce the snippet shown in Figure 2.6 where the inserted lines and corresponding tokens are marked:

```
void calc() {                          BEGIN_METHOD
  int a = 1;                           VAR_DEF
  int b = 2;                           VAR_DEF
⊕ System.out.println("test");        ⊕ APPLY
  while (a < 10) {                     BEGIN_LOOP
    a += 3;                            ASSIGN
  }                                    END_LOOP
⊕ bool done = false;                 ⊕ VAR_DEF
  int result = a - b;                  VAR_DEF
  System.out.println(result);          APPLY
}                                      END_METHOD
```

Figure 2.6: Insertion into running example



(a) Carbon Copy (b) Insertion Obfuscation

Figure 2.7: Effects of insertions on continuous matches

How does inserting semantic preserving lines into submissions deceive plagiarism detectors? Figure 2.7a shows the matches produced by the comparison algorithm from the running example and an identical copy. As one can see 9/9 tokens from the left submission match with 9/9 tokens from the right yielding 100% similarity.

Figure 2.7b shows the comparison where the right submission had two lines(token) inserted. On its own this doesn't do much as still 9/9 tokens match from the left with 9/11 from the right submission, resulting in the harmonic mean of 90%.

Now recall the minimum token match (MTM) from subsection 2.1.3, which describes how long a match must be, to be considered for similarity. If MTM is set to 4, the 3 matches from Figure 2.7b are too short and are ignored by the Greedy String Tiling algorithm resulting in a 0% similarity between both submissions.

Therefore all Mossad has to do, is insert just enough lines to get enough matches below MTM, so that the target similarity score is reached. Their evaluation in [9] shows that the inserted lines were few and authentic enough, as no participant in their survey could differentiate plagiarism from the original. Fooling plagiarism detectors and being inconspicuous to spot-checks, Mossad is very effective and requires little to no effort from a student. Its non-deterministic nature of putting random lines into random spots also makes mass plagiarism a viable option, as all generated plagiarisms from one original submission are also different from each other. As Devore-McDonald et al. only provide very limited countermeasures, Mossad was never released to the public and is only available for research purposes.

2.2.2 JPlag-GEN

JPlag-GEN is a plagiarism generation tool for *Java* programs built by Brödel for his thesis [4]. It reads original submissions passes them through the obfuscation modules and returns obfuscated plagiarisms. The user can choose between Insert-Obfuscation, Reorder-Obfuscation or a combination of both.

Insertion The idea of JPlag-GEN's insertion module is based on Mossad but deviates in some points from it. Like Mossad, it aims to insert statements into submissions that keep the output intact to lower the length of matching blocks, which inherently lowers the similarity of original and plagiarism. These insertions are also sourced from an entropy file and already processed submissions. But different to Mossad, JPlag-GENs insertion works in a deterministic way. Recall that Mossad inserts lines into random spots until a similarity threshold is reached. This randomness can lead to bloating of submissions as sometimes Mossad inserts multiple lines into the same spot, which has negligible impact on the similarity but noticeably increases the number of lines over time. JPlag-GEN on the other hand, tries to insert just one line into each spot. These one-line insertions suffice in breaking up matching blocks and avoiding bloating. Additionally, JPlag-GEN does not rely on a fixed similarity threshold but instead tries every possible insertion spot, which makes its plagiarism have much lower similarities than ones from Mossad and makes it harder to defend against. Formally it works as follows:

Algorithm 2 JPlag-GEN insertion

```
plagiarism ← original
while not empty(spots) do
    limit ← 0
    spot ← random(spots)
    do
        candidate ← random(dictionary ∪ original)
        plagiarism′ ← insert(plagiarism, candidate, spot)
        if compiles(plagiarism′) and output(plagiarism′) = output(plagiarism) then
            plagiarism ← plagiarism′
            dictionary ← dictionary ∪ candidate
            break
        else
            limit ← limit + 1
        end if
    while limit < 20
    spots ← spots \ spot
end while
```

Reordering JPlag-GEN's other obfuscation method is the reordering of independent statements. Independent here means that if you change the order of statements it does not affect the output of the program. For example, the blocks marked by ↑ and ↓ in Figure 2.8 are independent of each other and were switched.

```
void calc() {                          BEGIN_METHOD
  int a = 1;                           VAR_DEF
↑ while (a < 10) {                   ↑ BEGIN_LOOP
↑   a += 3;                          ↑ ASSIGN
↑ }                                  ↑ END_LOOP
↓ int b = 2;                         ↓ VAR_DEF
  int result = a - b;                  VAR_DEF
  System.out.println(result);          APPLY
}                                      END_METHOD
```

Figure 2.8: Reordering of running example

JPlag-GEN achieves reordering by testing for all pairs of consecutive lines if they can be swapped without changing semantics. After each swap is checked for legality, all allowed swaps are concatenated. Formally it works like this:

Algorithm 3 JPlag-GEN reordering

```
plagiarism ← original
legal ← ∅
while not empty(spots) do
    spot ← random(spots)
    plagiarism′ ← swapLines(plagiarism, spot, spot + 1)
    if compiles(plagiarism′) and output(plagiarism′) = output(plagiarism) then
        legal ← legal ∪ spot
    end if
    spots ← spots \ spot
end while
for all spot ∈ legal do
    plagiarism ← swapLines(plagiarism, spot, spot + 1)
end for
```

Reordering in itself is not a strong obfuscation attack because:

1. There are typically only a few independent lines in a program that can be swapped.
2. Subsequence matching still finds reordered matches as shown in Figure 2.9

Still, it can break up some matching blocks and should be used in conjunction with Insert-Obfuscation for best performance.

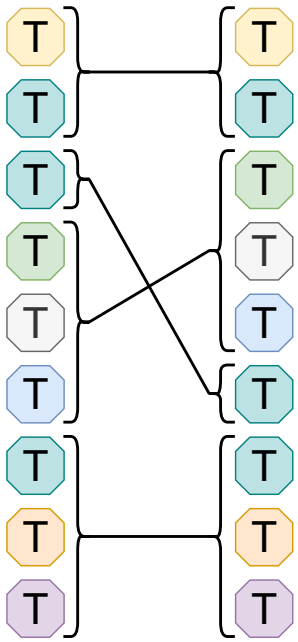


Figure 2.9: Effects of reordering on matches

As with Mossad, JPlag-GEN is not available to the public. The countermeasures Brödel provided in his thesis will be discussed in chapter 6.

3 Types of obfuscation attacks

The term "arms race" refers to a situation in which two parties engage in competitive efforts to outdo each other and establish superiority. When one party develops new and advanced weaponry, the other is driven to create even more sophisticated weapons, leading to an endless cycle of escalation. This phenomenon also applies to computer science and software engineering in the form of attack and defence papers. Author *A* develops a novel attack, and Author *B* creates a defence that protects against *A's* attack. Afterwards, Author *C* publishes an even better attack which fools *B's* defences.

We intend to break out of this circle by providing a defence mechanism against obfuscation attacks that doesn't target a particular attack but is as generally applicable as possible. To this end, we define the following three categories of obfuscation attacks based on which our defence is evaluated.

1. Semantic-preserving obfuscation
2. Semantic-agnostic obfuscation
3. Large language model obfuscation

In this chapter, we will define and illustrate attacks for each of these categories and provide threat models.

3.1 Semantic-preserving obfuscation

Semantic-preserving obfuscation attacks aim to hide plagiarism while leaving the semantics intact. Leaving the semantics intact here means that obfuscated plagiarism and plagiarism produce the same output when provided the same input. This is crucial for submissions in computer science courses as plagiarism must pass the automated grading tests in order to get good marks while being not too similar to the original to avoid plagiarism detection.

In chapter 2 we presented two attacks, Mossad by Devore-McDonald and Berger [9] and JPlag-GEN by Brödel [4] which fall under the semantic-preserving category. These only use dead-code insertion and/or reordering of independent statements to ensure semantics remain identically which is also confirmed by a comparison of the compiled objects file performed after each change to the source code.

Threat Model

We define the following threat model for semantic preserving attacks:

1. The attacker has access to the original submission and wants to hand in an obfuscated copy that deceives the plagiarism detector and still passes grading tests.
2. The attacker has access to the plagiarism detector, knows how it works and must break up matches between original and plagiarism while leaving the output identically.
3. To achieve this the attacker can use dead code insertion or reordering of independent statements
4. For dead code insertion the attacker uses a dictionary of *harmless* statements like variable definitions and print statements from which insertions are sourced.
5. For reordering of statements the attacker uses concatenation of pairwise line swaps.
6. For every obfuscation the attacker must confirm that changes leave the output unchanged, for example by comparing compilations of plagiarism and original.
7. The number of insertions must be kept as low as possible so as not to raise suspicion in spot-checks of the personnel.
8. To assure this the attacker can make insertions into random spots, calculate similarity of plagiarism and original after every change and stop when a similarity threshold is reached.
9. Alternatively, the attacker inserts one line into every possible spot.

3.2 Semantic-agnostic obfuscation

This section presents two new obfuscation attacks on which our defence will be evaluated. They differ quite much from attacks presented in the previous section as ours are:

- semantic-agnostic
- simulated attacks

First, we explain what both characteristics mean in this context and then define our attacks.

Semantic-agnostic Semantic-agnostic obfuscation stands as a broader framework than semantic-preserving obfuscation, designed specifically to transcend the inherent limitations that confine them. Semantic-preserving obfuscation has the constraint that the output of the submissions must remain the same after obfuscation. This limits the possible obfuscation to dead-code insertion or reordering of statements. Dead-code insertions are typically variable definitions and print statements which map to *VAR-DEF* tokens and *APPLY* tokens. Furthermore, the number of independent lines is quite small, as computations usually depend on previous computations. This directly limits the amount of possible swaps.

We decided to lift the output constraint for the semantic-agnostic attacks, which means they can change the submissions without caring if the semantics remain the same. This resulted in two powerful obfuscation techniques, *Alteration* and *Swapping*

Simulated attacks Obfuscation tools like Mossad or JPlag-GEN alter submissions on the source-code level, which are then fed into a plagiarism detector. Then they are tokenized and the comparison is run on the token sequences. For *Alteration* and *Swapping* we have decided to only simulate the attack. This means we directly alter the token sequence in the plagiarism detector, instead of altering the source code which would yield the same altered sequences. With this method, we can simulate these two attacks and evaluate our defenses against them, without providing the actual attacks.

Figure 3.1 shows plagiarism detector pipelines with real and simulated obfuscation attacks. The real one is red and produces an altered submission before they are fed into the detector. The simulated one is orange and changes the token sequences in the detector. For the comparison algorithm, it makes no difference when changes take place, making simulated attacks viable for evaluation.

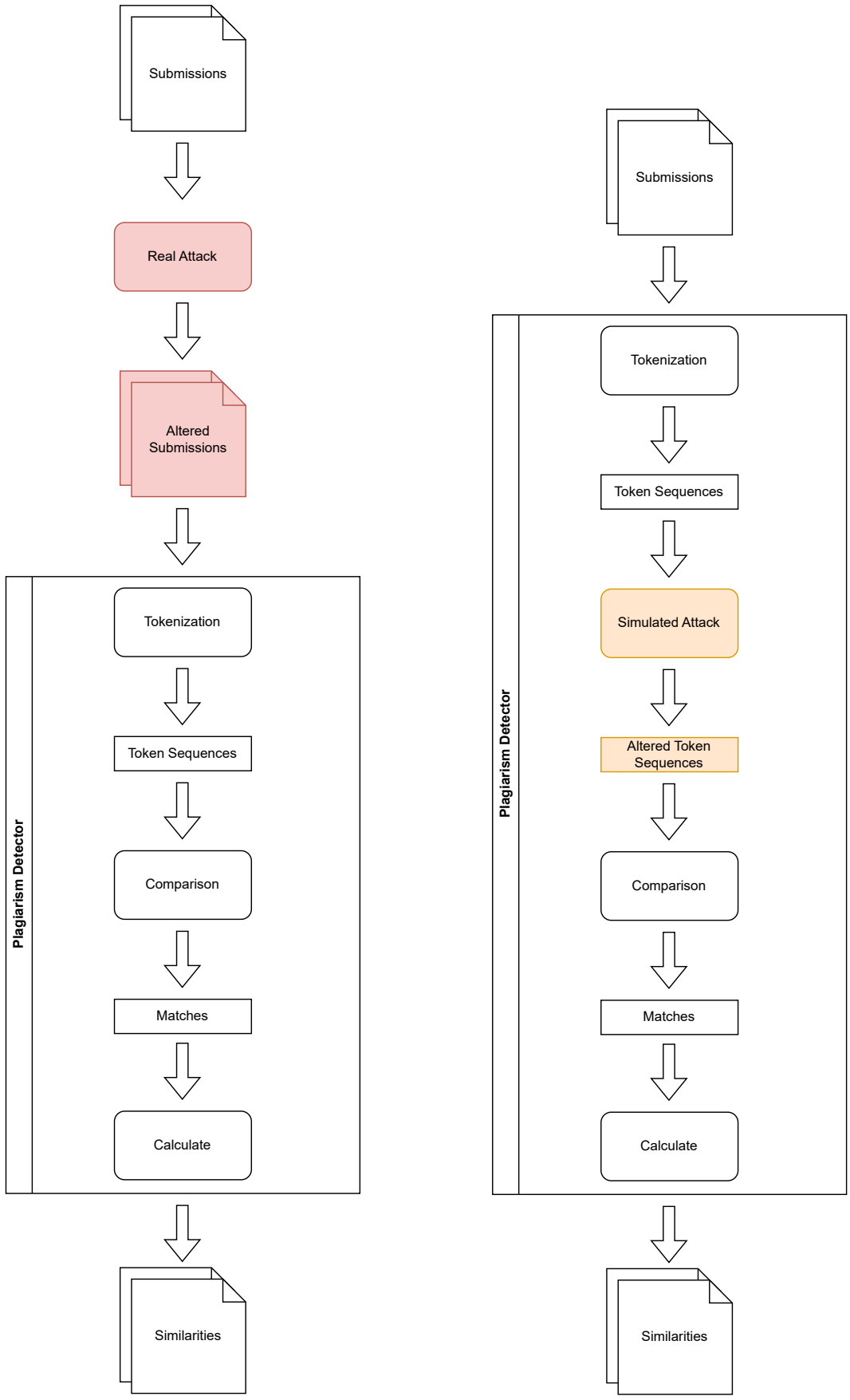


Figure 3.1: Plagiarism detector pipelines containing real and simulated obfuscation attacks

3.2.1 Alteration

Alteration is a simulated obfuscation attack where some tokens in the token sequence are replaced by tokens of another type. For example, it changes a *VAR_DEF* token to an *ASSIGN* token. Changes like this decrease the size of matching token blocks by breaking up larger match blocks into smaller ones. The ultimate goal is to undercut the minimum token length threshold, thereby drastically lowering the similarity. *Alteration* can be broken into two steps, building a token dictionary and performing the actual changes on the token sequence.

Token directory First, we build the token dictionary from which all alterations are sourced. This is necessary as token types are language-dependent and tokens of one language should be replaced with tokens of the same language. By building the dictionary from present submissions, we can ensure that alteration will not introduce new token types that no submissions previously used. This makes the simulated obfuscation more realistic since the resulting alterations are task-specific, reducing the risk of, e.g., encountering multithreading logic when the assignment was to create a simple calculator program.

To build the dictionary we iterate through all submissions and collect each unique type. Building the token dictionary from only the running example would yield these token types:

- BEGIN_METHOD
- END_METHOD
- BEGIN_LOOP
- END_LOOP
- ASSIGN
- VAR_DEF
- APPLY

Formally building the dictionary works as follows:

Algorithm 4 Build dictionary

```
dictionary ← ∅
for submission ∈ submissions do
    for token ∈ submission do
        if token.type ∉ dictionary then
            dictionary ← dictionary ∪ token.type
        end if
    end for
end for
```

Alteration of token types With the type dictionary, we can perform the actual alterations. The algorithm iterates through a submission and replaces the types of random tokens with random types from the dictionary. With a previously defined percentage, we can control how high the chance of each token is to be changed. As *Alteration* is semantic-agnostic and the comparison algorithm only produces matches, we do not need to limit the capabilities of *Alteration* by defining which token can be replaced by which.

The *Alteration* algorithm can be formally defined as follows:

Algorithm 5 Alteration

```
for token ∈ submission do
    if chance > random(0, 100) then
        replacement ← random(dictionary)
        token.type ← replacement
    end if
end for
```

Executing *Alteration* on the running example with the prior dictionary and a replacement chance of 30 percent could result in an altered token sequence like shown in Figure 3.2a, where the marked tokens were altered and therefore don't match with the original tokens:

(a) Tokens of altered running example

(b) Comparison with altered running example

Figure 3.2: Applying alteration on running example

Figure 3.2b shows the matches between the original running example with the altered one. The comparison algorithm is not aware of syntax and can compare the two even though the altered token sequence contains a *END_LOOP* token without a corresponding *BEGIN_LOOP*. The similarity between the two is drastically lowered and if the minimum token match is set to 4 the similarity would be 0. Finally, using *Alteration* for obfuscation does not increase the length of the token sequence.

3.2.2 Swapping

Swapping as a simulated obfuscation attack shares similarities with the *Reordering* performed by the attack of Brödel [4]. Both attacks use a sequence of swaps of adjacent lines to break up matching blocks which inherently lowers similarity. JPlag-GENs approach is to execute all possible swaps of independent lines, in that way, the output of the submission remains the same. This limits the amount of swaps and the effectiveness of the attack. As *Swapping* is a semantic-agnostic attack it is not bound to this constraint and has the ability to swap every pair of adjacent lines. Swapping all pairs would just lead to the first token moved to the last position. As this is ineffective we opted for randomness in the form of a percentage akin to *Alteration*. With it we can influence how many tokens are switched and as a byproduct get different obfuscated submissions with every run.

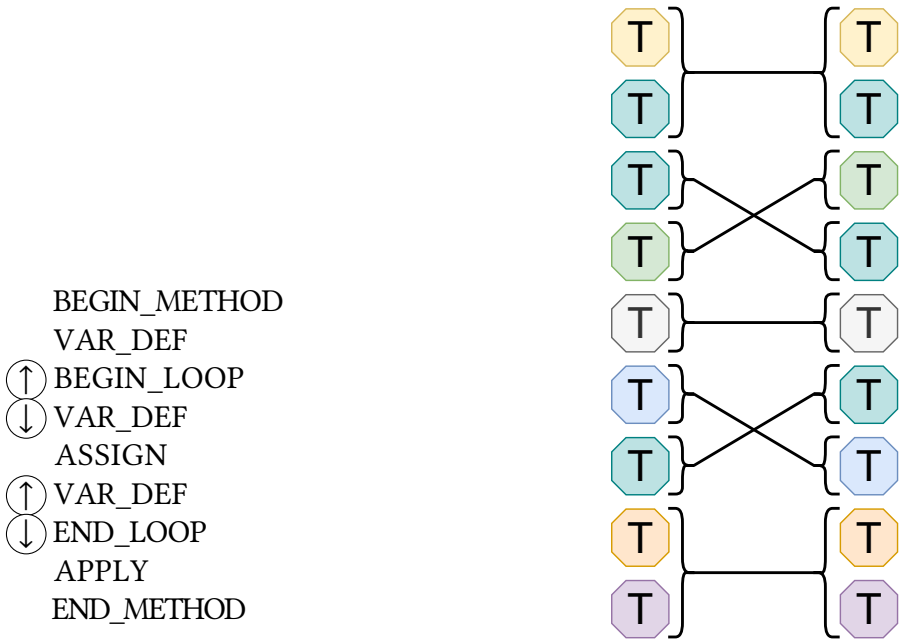
Algorithm To execute swapping we first set the chance of swaps to occur, i.e. 30 percent. Then we iterate through the submission and take the chance for each pair of tokens. If the chance occurs we swap them. Formally it can be defined like this:

Algorithm 6 Swapping

for $index \in (0, length(submission) - 1)$ **do**
  if $chance > random(0, 100)$ **then**
    $stored \leftarrow submission(index).type$
    $submission(index).type \leftarrow submission(index + 1).type$
    $submission(index + 1).type \leftarrow stored$
  end if
end for

Example To demonstrate the effects of *Swapping* we execute it on the running example with a swapping chance of 30 percent for each pair of tokens. As *Swapping* is random, the changed token sequences could look like Figure 3.3a, where pairs of tokens denoted with arrows indicate that they were swapped. *VAR_DEF*(3) was swapped with *BEGIN_LOOP*(4) and *END_LOOP*(6) with *VAR_DEF*(7).

The matches between the original and the swapped running example can be seen in Figure 3.3b. We can notice two characteristics of *Swapping*. First, the overlap between the submissions remains the same. This means that 9 tokens from the left- still match 9 tokens from the right-submission. This occurs due to the comparison algorithm being position invariant. Secondly, *Swapping* is very effective in breaking up large matching blocks into smaller ones. Here just two swaps break a matching block with the length 9 into 7 small blocks where none is larger than 2. Setting the minimum token match threshold to 3 for this example would result in a similarity of 0.



(a) Swapped tokens in running example

(b) Comparison with swapped running example

Figure 3.3: Performing swapping on running example

Threat Model

With *Alteration* and *Swapping* defined and explained we can define the threat model for semantic-agnostic obfuscation attacks.

1. The attacker has access to an original solution and wants to plagiarize it without getting detected.
2. The attacker knows how the comparison of submissions works and wants to change the internal token sequences to break up matching blocks, so the remaining parts are below the minimum token match.
3. The attacker can change the types of random tokens to other types present in the submission batch.
4. The attacker can randomly swap pairs of adjacent tokens in the sequence.
5. All changes are done in a random manner and the amount of changes can be controlled via a prior set percentage, specifying how much of the submission remains the same.
6. As the comparison algorithm produces matches and is semantic-agnostic, the changes in the token sequence may change the output of the obfuscated plagiarism.
7. As the comparison algorithm is also syntax-agnostic, changes can introduce syntax errors without hindering its matching.

3.3 Large language model obfuscation

As large language models (LLM) rise in popularity, they are increasingly likely to be misused for academic fraud. We want to evaluate how current state-of-the-art plagiarism detectors fair against LLM-supported plagiarism with and without our novel defence mechanism. To this end, we conduct experiments with Openai's GPT-3.5. With its 175 billion parameters, this particular model offers state-of-the-art performance and easy accessibility via the web-based chat application ChatGPT [5].

Use Case LLMs might be used for plagiarism in one of two ways:

1. Obfuscation: The LLM is prompted to obfuscate an existing solution to evade plagiarism detection.
2. Full solution generation: The LLM is prompted to solve the task and generate the submission on its own.

Even though only obfuscation constitutes plagiarism in a strict sense, we will investigate both strategies in this section. Afterwards, we present the threat model for large language model obfuscation.

3.3.1 Obfuscation

The quality of obfuscation via ChatGPT can be assessed by the similarity between the original and plagiarized submissions and the immutability of the submission's output. Both directly depend on the prompt used to instruct the LLM. As we cannot predict which prompts a student actually uses, we developed 14 different prompts, which utilise a broad range of obfuscation strategies. This aims to provide a diverse estimate of ChatGPT's obfuscation capabilities. The presented prompts can be split into generic- and engineered prompts. The effectiveness of these prompts is evaluated in detail in chapter 5.

Generic prompts Prompts in the generic category share two characteristics. They are low-effort queries, meaning a student could recreate one of those without investing much time or consideration. Secondly, they do not require knowledge of the functionality of plagiarism detectors, which makes them suitable for new undergrad students taking their first courses.

For prompts 2 to 8, the student just passes the original source code after the respective prompt and receives the plagiarized version. Prompt 1 is special, as it is a combined prompt meaning that the result of 1.a is fed into 1.b which then returns the final variant.

1.a Can you rewrite this code, so that the output is still the same, but it looks/is different:

1.b Can you refactor the code, not just change names and whitespace?

2. Can you refactor this code for me, so the output is still the same:
3. Can you change the syntax of this code without changing semantics:
4. Can you change the code so it looks different, but behaves the same:

5. Can you change the code so it looks like it was written by another person:
6. Can you take a look at this code, understand what it does and write a version of your own (same language) that achieves the same as the original, without relying on it:
7. Can you obfuscate this code by inserting dead code into it:
8. Can you obfuscate this code by inserting dead code into random spots:

Engineered prompts Prompts in this category are engineered to be more effective against plagiarism detectors and are less likely to be used by students as they require knowledge about how plagiarism detectors function. Prompt 9 tries to trick the LLM into performing obfuscation against detection to evaluate a new defence mechanism. Prompts 10 and 11 define what the LLM can and cannot do with the source code and employ obfuscation techniques akin to Mossad and JPlag-GEN. For prompts 12, 13 and 14 we provided the LLM a list of changes we'd like to do and asked it to engineer a prompt that would yield the best results.

9. I am developing a new defence against automated obfuscation attacks for plagiarism detectors. I would like to test the abilities of my approach. Could you attempt to obfuscate plagiarism on the following snippet, so I can evaluate my defenses?
10. Change this code for me, please. You are allowed to reorder the code, insert lines which don't do anything and remove comments. You are not allowed to change naming, whitespace and write comments yourself.
11. Change this code for me, please. You are allowed to reorder the code, change the structure, insert new code (which shouldn't affect the output) and remove comments. You are not allowed to change naming and write comments yourself.
12. Refactor the provided code snippet by accomplishing the following tasks:

    1. **Statement Reordering:** Rearrange the independent statements within the given code while preserving the original functionality. Do not rename any variables.

    2. **Method Reordering:** Rearrange the methods inside the provided class while maintaining the existing method signatures and variable names.

    3. **Unused Code Insertion:** Introduce an unused variable declaration and assignment that doesn't impact the code's behavior. In addition, insert an unused method within the class that includes a print statement.

    Here's the initial code snippet:

13. Refactor the provided code snippet by accomplishing the following tasks:

    1. **Statement Reordering:** Rearrange the independent statements within the given code while preserving the original functionality. Do not rename any variables.

    2. **Method Reordering:** Rearrange the methods inside the provided class while maintaining the existing method signatures and variable names.

3. **Unused Code Insertion:** Introduce multiple unused variable declarations and assignments that doesn't impact the code's behavior. In addition, insert 3 unused methods within the class that include a print statement.

   Here's the initial code snippet:

14. Refactor the provided code snippet by accomplishing the following tasks:

    1. **Statement Reordering:** Rearrange the independent statements within the given code while preserving the original functionality. Do not rename any variables.

    2. **Method Reordering:** Rearrange the methods inside the provided class while maintaining the existing method signatures and variable names.

    3. **Unused Code Insertion:** Introduce unused variable declarations and assignments that don't impact the code's behavior. Do so approximately every 5 lines.

    Here's the initial code snippet:

3.3.2 Fully generated

The second use case of LLMs for plagiarism is fully generating submissions based on their task description. To make our process replicable we define the following methodology which presumes the task is present in the form of a *Latex* document:

1. Remove pictures and diagrams from the document and rephrase them as text.
2. Input the task into the LLM and ask it to summarize the task as detailed as possible.
3. Let the LLM build the source code based on its summary

We remove graphics in step 1 as this ensures the LLM understands the task completely as we can only input text. Additionally, we observed the LLM being overwhelmed when asked to generate the submissions directly from the task, so we opted to let it produce an intermediate condensed version of the task first, which will then be used to generate the source code.

Threat Model

We now provide the threat model for large-language-model-supported plagiarism. It contains three different cases based on the attacker's knowledge and if an original submission is present:

1. The attacker has access to a capable LLM and wants to use it to pass a coding assignment
2. If the attacker only has access to the task description they can use it to fully generate a submission and hand it in.
3. If the attacker has access to an original solution, they want to hand in a copy while avoiding getting caught by plagiarism detectors.
4. For this he decides to prompt the LLM to obfuscate the original solution.
5. Depending on his knowledge of plagiarism detection they can either use generic prompts that require little effort or engineer more sophisticated prompts to increase their effectiveness.
6. The attacker must ensure that the prompt they use doesn't change the output of the solution.

4 Match merging

This chapter contains the development of and experiments on our novel defence mechanism against automatic obfuscation attacks. It is called Match Merging and will be explained in four sections. First, we will go over the general idea while providing definitions and examples. Afterwards follows the available decision basis with a short ablation study. Followed by the formal definition of the algorithm and finally the actual implementation of the defence mechanism. Match merging is inspired by the work of Krieg [16] which is discussed in chapter 6.

4.1 Idea

Plagiarism obfuscation attacks like Mossad or JPlag-GEN fool detectors by splitting up matching blocks of tokens until enough are so small that they fall below the minimum token match threshold and are ignored by the comparison algorithm, here greedy string tiling. Splitting up the token sequences can be achieved by inserting tokens, deleting tokens, reordering tokens or a combination of all three. The general idea is now to revert the splitting up of blocks by merging neighboring blocks of matches. First, we need to define what *neighboring* and *merging* means in this context:

4.1.1 Neighboring matches

Let there be two submissions and a list of matches between them. From here onwards the submissions are referred to as left submission and right submission. We define neighboring matches are those pairs of matching blocks that occur directly after one another in both sequences. We refer to these as upper and lower neighbors.

Figure 4.1 displays an example of four matches between two submissions. For simplicity, the nodes of this figure are matches instead of tokens and are denoted by an *M*. The length of each match is not relevant here.

Each of the four matches on the left has a counterpart in the right submission. For example, the fourth block on the left matches the first one on the right, giving us match $M_{left,4,right,1}$. Neighboring matches are those pairs of matches, where the upper neighbor and lower neighbor occur in conjunction in both the left- and right submission. The only two that fit these requirements are marked green in the figure, with the upper neighbor being $M_{left,2,right,3}$ and the lower $M_{left,3,right,4}$. The third block on the left comes directly after the second and the fourth on the right comes directly after the third. All other matches in this figure don't fit the neighboring criteria.

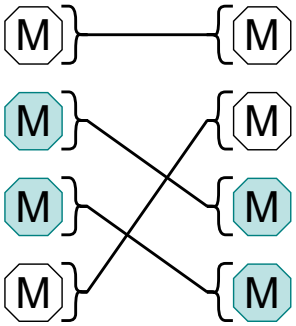
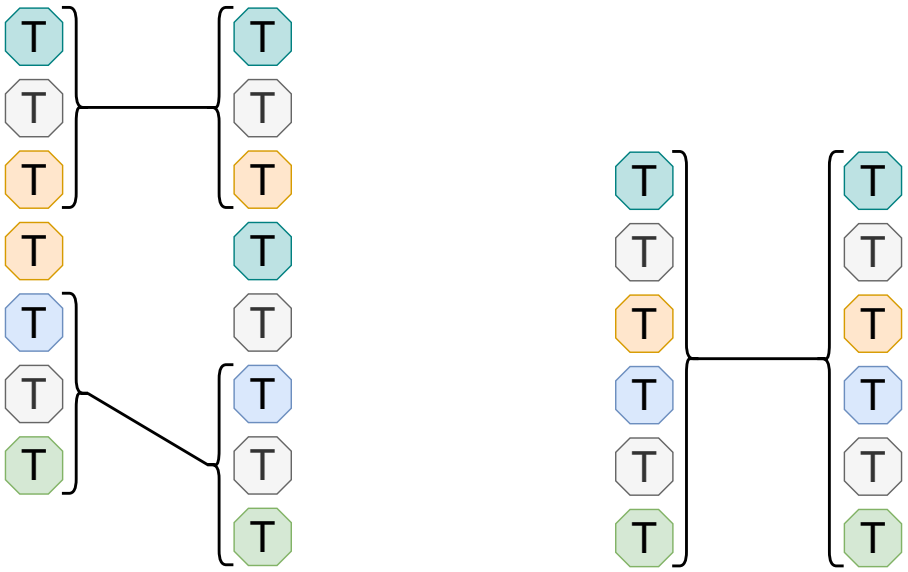


Figure 4.1: Neighboring matches example

4.1.2 Merging

We define merging of neighboring matches as removing the separating tokens from both token sequences and combining both matches into one large match which contains the tokens from both neighbors.

As seen in Figure 4.2a there were two neighbors which both have a length of three tokens. One token separates the upper match from the lower match in the left submission, while two separate these in the right. After merging there is only one matching block with a length of six tokens, as depicted in Figure 4.2b.



(a) Before Merging

(b) After Merging

Figure 4.2: Demonstration of merging two neighboring matches

Characteristics of match merging are:

1. Merging increases the size of matching blocks
2. Neighboring matches can have different length

3. The amount of separating tokens on each side can be different
4. Merging shortens submissions by removing separating tokens

4.1.3 Position in pipeline

Before we switch to deciding which matches we actually want to merge, we take a look at where Match Merging is placed in the pipeline of a plagiarism detector. Recalling chapter 2 detectors can be reduced to these steps:

1. Parse submissions through the language model to get token sequences
2. Compare token sequences pairwise using subsequence matching
3. Calculate similarity from the ratio between matched tokens and non-matched token

Match merging takes place between steps two and three and alters the matches produced by the comparison algorithm before they are used to calculate the similarities from them. Figure 4.3 displays the discussed pipeline. Round boxes represent components and square boxes data that is passed on. The component and data colored in green is our contribution.

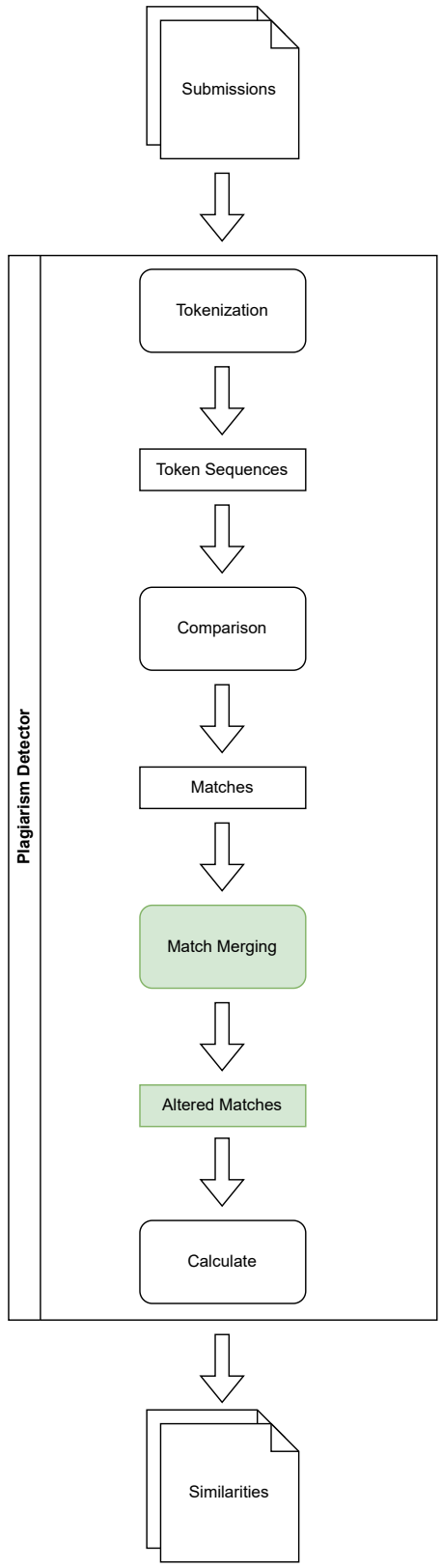


Figure 4.3: Match merging in plagiarism detector pipeline

4.2 Heuristic

Submissions fed into a plagiarism detector usually contain only a small amount of plagiarized submissions and a large amount of original submissions. We only want to increase the similarity of the plagiarized submissions to their respective originals while keeping similarities to unrelated originals as low as possible. Therefore we cannot just merge random or even all neighbors but instead need a heuristic and only merge those matches that fit it. This section goes over the available decision criteria, explains which are useful and which are not and finally describes the heuristic used for Match Merging.

4.2.1 Available decision basis

As Match Merging takes place after the comparison algorithm it has access to this information about compared pairs of submissions:

- List of matching blocks
  ⇒ Mapping of matching blocks to individual tokens in both sequences
  ⇒ Length of matching blocks
  ⇒ Amount of separating tokens between two neighboring matches.
- Token Sequences of both submissions
  ⇒ Length of both submissions
  ⇒ Explicit token types
  ⇒ Mapping of tokens to the corresponding file
- Minimum token match threshold

4.2.2 Information relevance

List of matches The list of matches is at the heart of the heuristic. Each match is defined as a triple of

- Starting position in the left submission
- Starting position in the right submission
- Length

From this list, we can extract two crucial information:

Neighbor length defines the length threshold of a match in order to be considered as a neighbor for merging. It is set lower than the minimum token match threshold, as matches previously ignored by the comparison algorithm should be considered for merging.

Gap size defines a size threshold for the gap between two neighboring matches. As gap sizes do not need to be identical for both submissions, recall Figure 4.2a, the left and right gap sizes have to be aggregated. I have considered taking the maximum of both values and taking the average. Experimenting with both showed that taking the average instead of the maximum provides better results because it is more stable. Refer to subsection 5.3.5 for a comparison between utilizing average and maximum. The averaged gap is rounded down to the next integer.
With neighbor length and gap size defined we can take a look back at Figure 4.2a. The depicted merge would only be allowed if

$$\text{neighbor-length threshold} \leq 3 \text{ and gap-size threshold} \geq 1$$

Both thresholds are treated as hyperparameters and different combinations yield different results. In chapter 5 their effects are shown and default values are presented.

Minimum token match threshold As discussed previously the minimum token match defines which matches are considered by the subsequence matching algorithm and passed on to Match Merging. It is set for each language by the developers of the plagiarism detector.

If we would only merge matches that are above this threshold, it would render this defence mechanism useless as obfuscation tools break up matches so they are below MTM. Resulting in them being ignored and never reaching the merging component.

To circumvent this we give the comparison algorithm a second threshold, the neighbor length discussed above, as we want to see all matches above this value to consider them for merging. We can put these ignored matches into a second list and pass them on.

Token Sequences The provided token sequences yield another two possible decision criteria:

Explicit token types With explicit token types, one could make rules for allowing merges based on tokens in the neighboring matches or the gaps. Such a rule could look like:

$$\forall token \in gap : isType(token, APPLY) \vee isType(token, VAR\_DEF)$$

The problem with token-type-based rules is that types are different for every language and there is currently no unified representation. This would make our approach highly language-dependent and would force rule-creating for each current and future supported language. Therefore we opt to ignore token types for our approach, to maintain its language-independence.

Tracing token to file Submissions can consist of multiple files containing source code. The way plagiarism detectors work with this is by using a special token named FILE_END. It is inserted into the token sequences between files and serves as a pivot element so the comparison algorithm does not mark matches beyond file boundaries. Match Merging could theoretically merge over file boundaries but testing showed that:

1. they yielded very little increased similarity between plagiarism and original (around 1% increase). subsection 5.3.5 shows a comparison of allowing or forbidding file-boundarie-merges.

2. sometimes results in weird phenomena such as similarities above 100%.

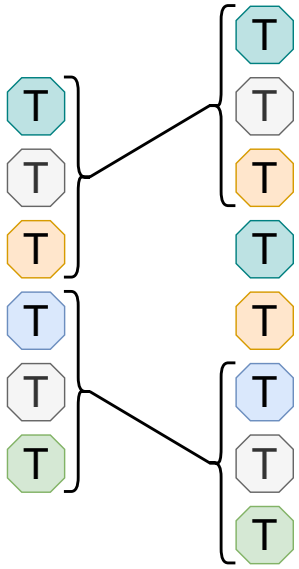
Therefore the heuristic will prohibit merges across files.

4.2.3 Definition of heuristic approach

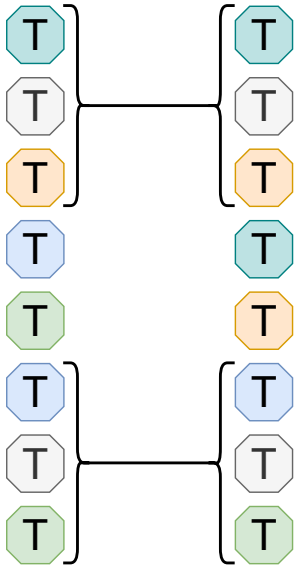
Formally, our heuristic is defined to merge every pair of neighboring blocks under the following conditions:

- The length of both neighbors is at least *neighbor-length*
- The average size of the gap between neighbors is at most *gap-size*
- There is no *FILE_END* token in the gap

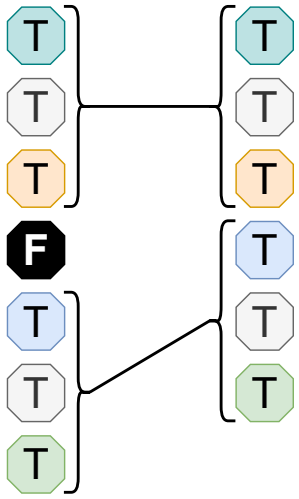
with *neighbor-length* and *gap-size* being hyperparameters.



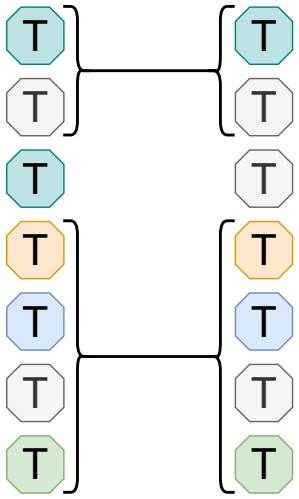
(a) **Valid**



(b) **Invalid**: Average gap-size > 1



(c) **Invalid**: Gap contains *FILE_END* token



(d) **Invalid**: Length of upper neighbor < 3

Figure 4.4: Demonstration of heuristic with neighbor-length = 3 and gap-size = 1

Figure 4.4 demonstrates the heuristic on four example situation with neighbor-length = 3 and gap-size = 1. The only valid merge is example (a). In example (b) the average gap is two which is greater than one, in example (c) the merge would contain a *FILE_END* token and in example (d) the length of the upper neighbor is two which is less than three.

4.3 Algorithm

In this section, we define the formal algorithms for Match Merging. Firstly we will cover a few subroutines that make the then following core logic more readable and understandable. *Compute Neighbors* receives the list of matches and returns all neighboring matches. *Gap Tokens* receives a pair of neighbors and returns all tokens in the gap for further processing. *Merge Neighbors* performs the merging of two neighbors as previously defined. *Prune Neighbors* is used at the end of our procedure and removes leftovers from the list of matches.

Compute Neighbors

This algorithm is responsible for computing neighboring matches. The input is an unsorted list of matches and the output is a list of pairs of upper and lower neighbors. Recall that each match is defined by a starting position on the left and a starting position on the right. We create two sorted lists, one based on the left-start and one on the right. Now we check if two matches appear directly after one another in both lists. Pairs that do so are saved to the neighbor list which gets returned in the end.

Algorithm 7 computeNeighbors

$neighbors \leftarrow \emptyset$
$sortedByLeftStart \leftarrow sort(matches, leftStart)$
$sortedByRightStart \leftarrow sort(matches, rightStart)$
for $index \in [0, length(matches) - 1]$ **do**
    $upperNeighbor \leftarrow sortedByLeft(index)$
    $lowerNeighbor \leftarrow sortedByLeft(index + 1)$
    if $indexOf(sortedByRight, upperNeighbor) = indexOf(sortedByRight, lowerNeighbor) - 1$ **then**
        $neighbors \leftarrow neighbors \cup (upperNeighbor, lowerNeighbor)$
    end if
end for

Gap Token

This subprocedure receives a token sequence and two neighbors. Its task is to return the tokens in the gap between both neighbors from the token sequence. For this, it computes a range from the matches and returns tokens in the range

Algorithm 8 gapTokens

$endOfUpper \leftarrow startOfUpper + lengthOfUpper$
$slice \leftarrow tokenSequence(endOfUpper, startOfLower)$

Merge Neighbors

At last, we need a procedure that takes two matches and returns the merged one. We achieve this by extracting the start on the left and the right from the upper neighbor and adding the length of both neighbors together.

Algorithm 9 gapTokens

$startLeft \leftarrow upperNeighbor.startLeft$
$startRigth \leftarrow upperNeighbor.startRigth$
$length \leftarrow upperNeighbor.length + lowerNeighbor.lenght$
$merged \leftarrow (startLeft, startRight, length)$

Prune Neighbors

Match Merging works on normal matches which are at least *minimum token match* long and ignored matches which are at least *neighbor-length* long. After merging all eligible matches the leftover ignored matches have to be removed. For this, we iterate through all matches after merging and remove those whose lengths are less than *minimum token match*.

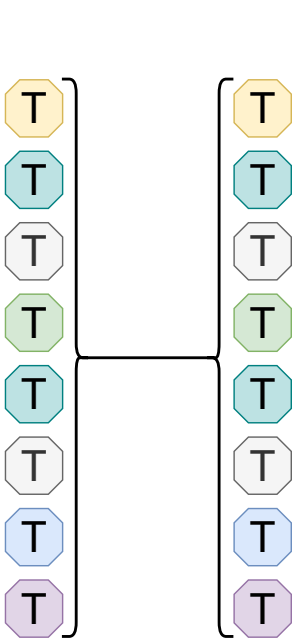
Algorithm 10 pruneMatches

$prunedMatches \leftarrow \emptyset$
for $match \in matches$ **do**
  if $match.length \geq MTM$ **then**
    $prunedMatches \leftarrow prunedMatches \cup match$
  end if
end for

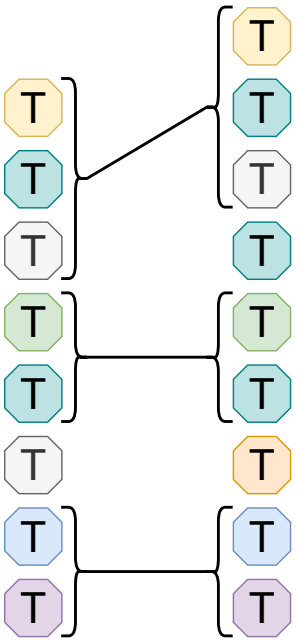
Match Merging

Obfuscation attacks use insertions, swaps and alterations to hide a plagiarized submission. These changes break up continuous matches between plagiarism and original into smaller ones and if the remaining matches fall below the minimum token match threshold they are discarded by the plagiarism detector. Our approach is to merge the broken matches back together to revert changes from the obfuscation tools and thereby raise the similarity. To decide which neighboring matches should be merged we developed a heuristic which relies on two parameters. Neighbor-length indicates the required length of a neighbor, while gap-size specifies the allowable average size of the gap between two neighboring matches.

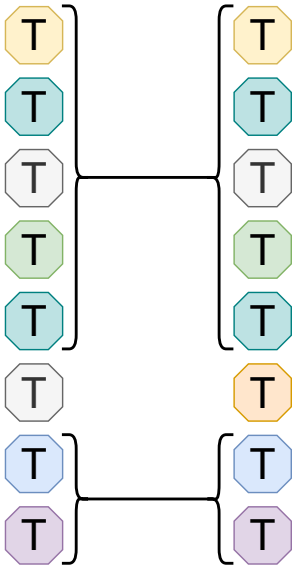
With the previously provided subroutines we can define the algorithm for match merging. As input, it receives both submissions in the form of token sequences alongside the normal and ignored matches. From here we compute the neighbor and iterate through them. If a neighbor suits the criteria discussed in the last section, we merge it and remove the gap from the sequences. After every merge, we compute the neighbors again and this loop runs until all neighbors have been checked. Finally, we remove the leftover ignored matches from the result to ensure that every match in there is above MTM.



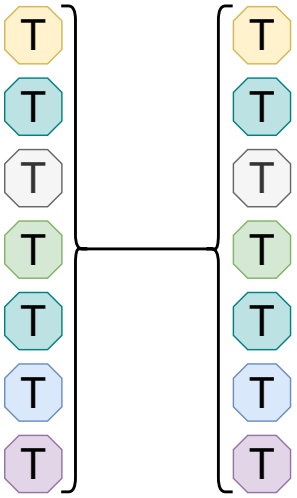
(a) Untouched copy - Similarity = 100%



(b) Obfuscated copy - Similarity = 0%



(c) After one merge - Similarity = 62.5%



(d) After two merges - Similarity = 100%

Figure 4.5: Demonstration of match merging with minimum token match = 4, neighbor-length = 2 and gap-size = 1

Before we provide the pseudocode for match merging we demonstrate the algorithm and its effects on the similarity score in an example displayed in Figure 4.5. For this example, we set the minimum token match = 4, the neighbor-length = 2 and the gap-size = 1.

In Figure 4.5a we see the comparison between two identical token sequences which are 8 tokens long. As all tokens from the left submissions match the ones from the right the similarity is 100%:

$$\frac{2 * (8/8) * (8/8)}{(8/8) + (8/8)} = 1$$

In Figure 4.5b the right submission has undergone obfuscation via Insertion and Alteration. One token was inserted between the third and fourth token and the type of the sixth token was changed. This results in the large match being split into three parts. The three remains are 2 or 3 tokens long and fall below the minimum token match of 4. Therefore the similarity between both submissions is 0:

$$\frac{2 * (0/8) * (0/9)}{(0/8) + (0/9)} = 0$$

We now utilize match merging to revert the obfuscations and raise the similarity. The first possible pair of neighboring blocks is match 1 and match 2. As both neighbors are at least 2 tokens long and the average gap between them is at most 1, we merge them. We remove the gap and unionize the neighbors to a match of length 5 which is displayed in Figure 4.5c. As the new block is above the minimum token match the similarity rises to 62.5%:

$$\frac{2 * (5/8) * (5/8)}{(5/8) + (5/8)} = \frac{5}{8} = 0.625$$

The list of matches has changed and we once more search for suitable neighbors. Match 1 and match 2 are next to each other. As their lengths 5 and 2 are longer than neighbor-length and their average gap of 1 is at most gap-size, we merge them together by removing the gap and unionizing the neighbors. The resulting match has length 7 and is displayed in Figure 4.5d and we raised the similarity back to 100%:

$$\frac{2 * (7/7) * (7/7)}{(7/7) + (7/7)} = 1$$

There are no more neighbors left and we check if we must prune matches. As all present matches are at least minimum token match long, we don't have to remove anything and the match merging procedure terminates.

The pseudocode for match merging is defined as follows:

Algorithm 11 Match Merging

```
matches ← matches ∪ ignoredMatches
neighbors ← computeNeighbors(matches)
index ← 0
do
    upperNeighbor ← neighbors(index).upper
    lowerNeighbor ← neighbors(index).lower
    gapLeft ← gapTokens(leftSequence, upperNeighbor, lowerNeighbor)
    gapRight ← gapTokens(rightSequence, upperNeighbor, lowerNeighbor)
    valid ← true
    if FILE_END ∈ gapLeft or FILE_END ∈ gapRight then
        valid ← false
    end if
    if (length(gapLeft) + length(gapRight))/2 > gapSize then
        valid ← false
    end if
    if valid then
        matches ← (matches \ upperNeighbor) \ lowerNeighbor
        merged ← merge(upperNeighbor, lowerNeighbor)
        matches ← matches ∪ merged
        leftSequence ← leftSequence \ gapLeft
        rightSequence ← rightSequence \ gapRight
        neighbors ← computeNeighbor(matches)
        index ← 0
    else
        index ← index + 1
    end if
while index < length(neighbors)
matches ← pruneMatches(matches)
```

One might notice that the neighbor-length check is not performed in the Match Merging procedure. The reason is that the matches provided from the comparison algorithm fulfill these requirements:

$$\forall match \in matches : |match| >= MTM,$$
$$\forall match \in ignoredMatches : |match| >= \textit{neighbor-length},$$
$$\text{with } \textit{neighbor-length} \leq MTM$$

Hence all matches that reach Match Merging are above the *neighbor-length* threshold and a separate check would be redundant.

5 Evaluation

This chapter contains the evaluation of our approach against automated obfuscation attacks. We first define the Goal-Question-Metric plan which serves as the framework of this evaluation. Afterwards follows *Design*, where used datasets, generated plagiarisms and underlying methodology will be discussed. The *Results* section is split into plots and discussion. Finally, we will go over threats to validity and limitations.

5.1 Goal-Question-Metric plan

The Goal-Question-Metric plan [38] is the foundation of our empirical evaluation. It defines the goals of our work, the questions necessary to show if we achieved our goals and the metrics to access the questions. Preceding the presentation of the plan, it is essential to establish precise definitions, ensuring unambiguous terminology within the context of our study.

Similarity

We previously defined the similarity of two submissions as the mean ratio between matched and non-matched tokens. The higher the similarity, the more likely one of the submissions is an instance of plagiarism. We will use three different categories of similarity in this evaluation:

Plagiarism × original The first category consists of similarities between plagiarized submissions and their corresponding original. These are the similarities we want to increase as much as possible, to raise suspicion and prevent academic fraud. In classical terms, they are true positives for high similarity scores and false negatives for low scores.

Original × original The similarities in this category stem from comparisons between different originals and comparisons between originals and non-corresponding instances of plagiarism. We want to keep these similarity scores as low as possible so as not to raise suspicion about genuine submissions. In classical terms, they are true negatives for low similarity scores and false positives for high scores.

Plagiarism × plagiarism The last category considers similarities between different instances of plagiarism and is only of interest for comparing fully generated submissions to each other where there is no original submission in the classical sense.

For evaluation, we integrated match merging into the widely-used open-source plagiarism detector JPlag [13]. This choice was driven by JPlag's popularity as the second most used plagiarism detector [31] and its open-source nature, distinguishing it from closed-source detectors like Moss [23].

With the completed implementation and the previously outlined similarity categories, we can now formulate the Goal-Question-Metric-Plan, outlining the evaluation objectives and metrics for assessing our novel defence mechanism against obfuscation attacks:

G.1 **Make JPlag more resilient against semantic preserving obfuscation attacks**

- Q.1.1 How does match merging defend against Mossad?
  - M.1.1.1 Similarities of plagiarism × original
  - M.1.1.2 Similarities of original × original
  - M.1.1.3 Difference in the number of inserted lines (absolute/relative)
  - M.1.1.4 Difference in required plagiarism generation time
- Q.1.2 How does match merging defend against different attacks from JPlag-GEN?
  - M.1.2.1 Similarities of plagiarism × original
  - M.1.2.2 Similarities of original × original

G.2 **Make JPlag more resilient against semantic agnostic obfuscation attacks**

- Q.2.1 How does match merging defend against alteration?
  - M.2.1.1 Similarities of plagiarism × original
  - M.2.1.2 Similarities of original × original
- Q.2.2 How does match merging defend against swapping?
  - M.2.2.1 Similarities of plagiarism × original
  - M.2.2.2 Similarities of original × original

G.3 **Make JPlag more resilient against AI-based obfuscation attacks**

- Q.3.1 How does match merging defend against obfuscation prompts?
  - M.3.1.1 Similarities of plagiarism × original
- Q.3.2 How does match merging defend against fully generated submissions?
  - M.3.2.1 Similarities of plagiarism × plagiarism

G.4 **Maintain performance of JPlag**

- Q.4.1 How does match merging affect the performance of JPlag?
  - M.4.1.1 Runtime of base JPlag and JPlag with match merging

Similarity is assessed considering the median, mean, and interquartile range (IQR)

5.2 Design

This section delves into the intricacies of the evaluation process, exploring the selection of datasets, the methodology behind generating plagiarism instances, and the seamless process of the evaluation pipeline.

5.2.1 Datasets

Our contributions to plagiarism detection are evaluated based on three different datasets:

AI-SOCO The first dataset is *AI-SOCO* [10] which contains 100.000 *C++* programs. The creators collected submissions from *Codeforces* [7] a competitive coding website, where users tackle programming problems of varying difficulty either against other users or the clock. These tasks range from computing graph distances to inverting binary trees and are single-file submissions. We took the first 100 submissions from the dataset to evaluate the defence against Mossad.

PROGpedia The second dataset is *PROGpedia* [32] which consists of collected submissions for 16 different programming assignments from 2003-2020. All submissions were from undergrad computer science students, which fits the threat models. Although *PROGpedia* contains solutions in multiple languages we will use Java submissions to evaluate the defence against JPlag-GEN. As Brödel [4] based his work on assignment 19 and assignment 56, we will do the same, giving us 66 and 85 submissions to work with. As with *AI-SOCO*, they are all single-file submissions.

TicTacToe The last dataset is *TicTacToe*, an assignment for first-semester undergrad students at Karlsruhe Institute of Technology. The task was to recreate the game tic-tac-toe in Java to support human-vs-human and human-vs-ai game modes. This dataset is of interest because its submissions are around 4 times larger than those in *AI-SOCO* and *PROGpedia* and additionally contain multiple files per submission. *TicTacToe* contains 738 submissions and we chose 63 at random for testing.

5.2.2 Plagiarism generation

We use the datasets from above with the obfuscation attacks from chapter 3 to generate plagiarism for the evaluation. In the following, we will go over each of them describing the generation process and noting intricacies.

Mossad We use Mossad on the 100 submissions from *AI-Soco* to generate plagiarism. Recall from chapter 2 that it inserts lines into the submissions until a similarity threshold is reached. As it was originally built for the plagiarism detector Moss we use a slightly tweaked version which allows the usage of JPlag for checking similarities. We used 25 percent as the target similarity threshold, which is the default value presented by Devore-McDonald and Berger [9].

From the 100 submissions, 9 had encoding errors and 1 did not compile therefore we use the remaining 90 originals along with 90 generated plagiarism which serves as the test set with 180 submissions.

JPlag-GEN JPlag-GEN supports three different modes for obfuscation:

1. insertion
2. reordering
3. insertion and reordering

We use all three modes on task 19 and task 56 from *Progpedia* which provides six test sets where sets based on task 19 contain 132 submissions and the ones based on task 56 contain 170.

Furthermore, we use JPlag-GEN on submissions from *TicTacToe*. As these are much longer than the previous submissions, computation time increases exponentially and therefore we only consider the insertion mode. From 63 random submissions, 12 had syntax errors or were empty resulting in the test set containing 102 submissions.

Alteration/Swapping To test against alteration and swapping we use the 91 *C++* submissions from *AI-Soco*. As both attacks are simulated we copy each submission and let the internal JPlag module perform the obfuscation. The amount of obfuscation can be tweaked via a percentage and we choose 10%, 20% and 30% to observe how the performance of match merging changes when more or less tokens in the submissions are changed. This offers 6 different test sets with 182 submissions each.

Obfuscation Prompts We applied the 15 obfuscation prompts from chapter 3 to one file each from *AI-Soco*, *PROGpedia* assignment 19 and assignment 56. Furthermore, we wanted to test the limits of prompt 6, where the LLM, here GPT-3.5, should understand the program and write a version of its own, instead of relying on the original. To this end, we applied prompt 6 on one submission from *TicTacToe* as submissions from this dataset are longer and more complex. In summary, the obfuscation prompt test set contains 4 originals and 46 plagiarisms.

Fully generated submissions To make fully generated submissions more comparable we decided to base them on the task description of the *TicTacToe* datasets. Students should implement the game Tic-Tac-Toe in Java. The task is split into the base game and an extension. For the base game, two humans sit at the same computer and can alternately pick squares, until either one of them wins or there are no more free squares. For the extension, students should create a simple AI which makes decisions based on the 5 rules provided in the task description. Following the methodology in chapter 3 we first clean the task from figures and pictures and prompt ChatGPT to summarize the complete task and generate submissions from this summary. We quickly noticed that GPT-3.5 is not capable of generating the complete task which was reflected in responses being just the structure of the program and containing no actual logic. Because of this, we decided only to consider the base game for the evaluation. From the summarized base game task shown in Figure 5.1 we generated 10 different solutions which have no syntax errors and provide the specified functionality. Although Tic-Tac-Toe can be played like the description demanded, the fully-generated submissions fail the automatic grading tests, because they don't adhere to the formatting rules for command line output.

```
The task you provided is to implement a program for playing the game
    of Tic-Tac-Toe. Here are the precise steps that need to be done:

1. **Game Description**:
   - Tic-Tac-Toe is a simple game for two players played on a \(3 \
     times 3\) grid.
   - One player uses the symbol "x," and the other uses the symbol "o
     ."
   - The goal is to be the first to place three of your symbols in a
     row, column, or diagonal.
2. **Game Board**:
   - The game board consists of a \(3 \times 3\) grid with cells
     numbered from 0 to 8 (left to right, top to bottom).
3. **Game Flow and Interaction**:
   - The game is started by running a `main` method.
   - The program takes alternating moves from two players, starting
     with "x" and followed by "o."
   - Before the first move, the empty game board is displayed.
   - After each move, the updated game board is displayed.
4. **Player Moves**:
   - Moves are numbered starting from 1.
   - A player's move consists of the following steps:
     1. Display the move number and the player's symbol (e.g., "1.
        Zug: x").
     2. Read the player's move as an integer between 0 and 8 (
        representing the cell to place their symbol).
     3. Display the updated game board with the player's symbol in
        the chosen cell.
5. **Game End**:
   - The game ends when a player has three symbols in a row, column,
     or diagonal, and the winner is declared (e.g., "Sieger: x" or "
     Sieger: o").
   - If the game board is full, and no player has won, the game ends
     with "Kein Sieger" (No Winner).
6. **Implementation**:
   - Create a class for the game board, including methods for
     checking if a cell is occupied, checking if a player has won,
     checking if the board is full, placing a player's symbol, and
     returning the textual representation of the game board.
   - Create a class for input and output, with the `main` method.
   - Ensure that all input and output are handled in the input/output
      class using `System.out.println` for output and `Scanner` for
     input.

These steps provide a clear and precise overview of what needs to be
    done to implement the Tic-Tac-Toe game as described in the task.
```

Figure 5.1: Summarized base version of Tic-Tac-Toe

5.2.3 Methodology

The evaluation pipeline was realized in *Python* and can be split into three parts:

Core The core applies to every evaluation in this chapter and works as follows: We run the plagiarism detector JPlag on a test set while setting a specific combination of the hyperparameters *neighborLength* and *gapSize* via the command line. After comparison, JPlag produces an archive with the results in it. We extract the archives, load the JSON files and extract the pairwise similarities from it. We previously set up the test sets in a way, that original submissions are named 1 to n and plagiarized submissions have the specific prefix *p_*, for example, plagiarism *p_7* belongs to original *7*. With the defined naming scheme we can easily assign similarities to the three categories discussed above:

- Plagiarism × original
- Original × original
- Plagiarism × plagiarism

We utilized the similarities in one of two ways, depending on the test set:

Full grid search We want to understand how different combinations of the hyperparameters affect the performance of match merging. To this end, we conduct an exhaustive grid search to test all reasonable combinations. *neighborLength* can be between 1 and the minimum token match threshold, which is 9 for *Java* and 12 for *C++*. For *gapSize* we decided on an upper limit of 20, meaning there can be 20 tokens in the gap separating two neighbours. Formally we applied the core for all hyperparameter combinations

$$(neighborLength, gapSize), \text{with } 1 \leq neighborLength \leq MTM \land 1 \leq gapSize \leq 20$$

We compare the performance based on aggregated similarities rather than similarity distributions for full grid search, as it makes changes easier to recognize. To this end, we utilize each similarity category's median, mean and interquartile range. By using aggregate functions with varying degrees of outlier resistance, we can assess how match merging affects the broad mass and singular outliers, giving us valuable insight. For the interquartile range, we take the 25 percentile of Plagiarism × original and the 75 percentile of Original × original. Full grid search is used for the assessment of Goal 1 and Goal 2.

Single combination For test sets in this category, we apply the core once on JPlag without match merging and once with match merging with a single combination of *neighborLength* and *gapSize*. The visual presentation differs from the full grid search as we don't aggregate the similarities but instead compare the two distributions. We use this for the assessment of Goal 3 as there are fewer submissions in the test sets.

5.3 Results

In this section, we will go over the results of our evaluation on match merging as a defence against automatic obfuscation attacks. We've structured our presentation based on the four goals outlined in the GQM-Plan. We'll showcase the results using plots and then discuss them. Following the goals, we'll also touch on a brief ablation study, explaining two design choices from chapter 4. Lastly, we examine the default hyperparameters utilized for the underlying heuristic approach.

5.3.1 Resilience against semantic preserving obfuscation attacks

Mossad The performance of match merging against Mossad is displayed in Figure 5.2. There are three plots based on the Mossad Soco test set. Each plot utilizes a different aggregation function for the similarities, the first median, the second mean and the last interquartile range. In each plot, the y-axis denotes similarities ranging from 0 to 1. Scatter points in blue refer to the similarity class plagiarism × original and red points to original × original. A higher distance between blue and red points indicates better performance, as plagiarism gets more distinguishable from original submissions. The x-axis represents neighbor length values and **O** stands for off and refers to the base version of JPlag without our defence mechanism. As the minimum token match for *C++* is set at 12, the other values range from 11 to 1. Because neighbor length is the dominant hyperparameter, these values display the performance of a combination from the given neighbor length with the best gap size per bin.

We can see, that the base similarity of plagiarism × original is slightly under 20 percent for median and mean, which is to be expected as Mossad is run with a target threshold of 25 percent. The base similarities of original × original are around 0 percent. As we start to use match merging we see that the similarities of plagiarism × original increase steadily with the increasing neighbor length until it reaches the peak of around 80 percent at neighbor length 2, after which it slightly lowers. Meanwhile, the similarities of original × original are mostly unaffected with the highest increase being 6 percent for the mean at neighbor length 3. We observe, that the more we use match merging, the larger the gap between plagiarism × original and original × original gets, which is the desired behavior. This even holds true for the interquartile range which is highly outlier affected.

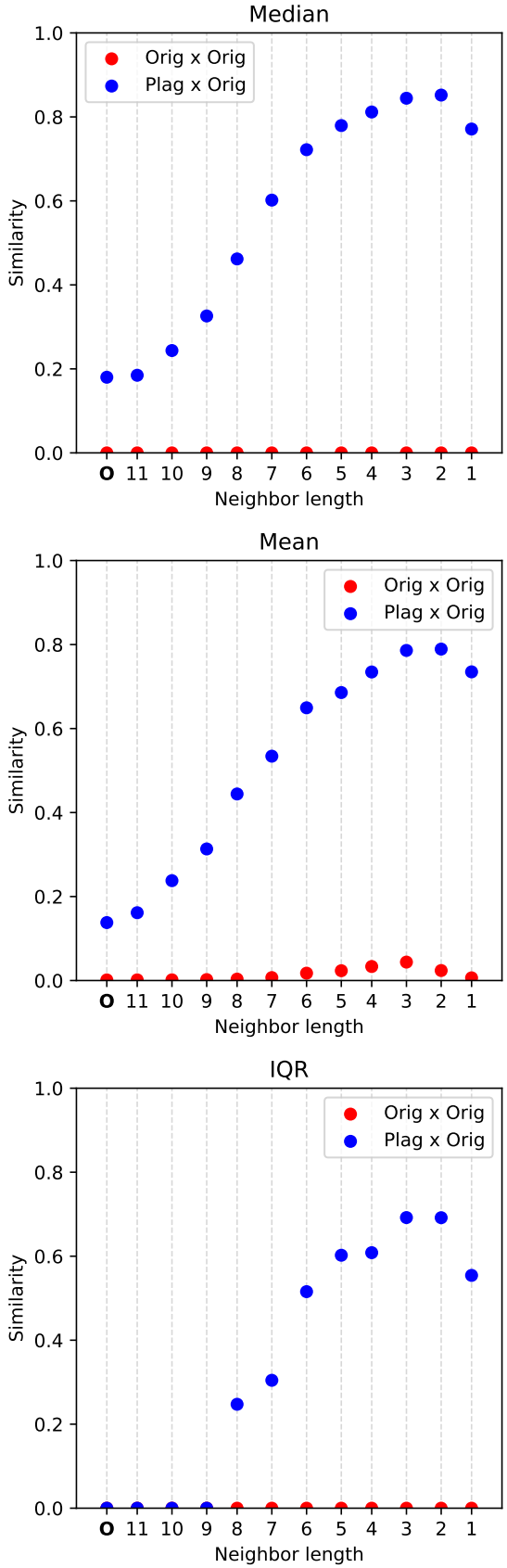


Figure 5.2: Similarities on the Soco-Dataset with Mossad plagiarisms for varying neighbor lengths

Mossad uses JPlag after every insertion to calculate the new similarity between original and plagiarism. This raises the question of how it would affect the plagiarism generation process of Mossad if match merging is already integrated into the plagiarism detector pipeline. It is interesting how the amount of inserted lines and generation time changes when our defence mechanism increases similarities in the generation process. To this end, we generated plagiarism from the *AI-Soco* for the targets 10, 25 and 40 percent and for each target used base JPlag and JPlag with match merging, resulting in 6 plagiarism generation runs.

Figure 5.3 shows the results of this comparison. Each column covers one percentage threshold, the first 10 percent, the second 25 percent and the last 40 percent. Each row covers a different attribute of interest. The first row displays the total amount of inserted lines, the second row shows the factor of increase and the last row covers the plagiarism generation time. Each of the 9 plots contains 2 boxplots, one for base JPlag and one for JPlag with match merging. Finally, it should be noted that each plot has its own y-axis scaling.

The following observations are based on the median of each boxplot for easier understanding. We can see that the amount of inserted lines is tripled for the 10 percent target (from 51 to 149), doubled for 25 percent (from 39 to 88) and tripled for 40 percent from (26 to 74). Factor of increase denotes how the relative size of a submissions changes after plagiarizing. A factor of 2 would denote that a file is twice as large after obfuscation. For the 10 percent target the factor of increase changes from 1.75 to 3.1, for the 25 percent target 1.55 to 2.3 and lastly for the 40 percent target from 1.4 to 2.0. Finally, the generation time is tripled, doubled and tripled akin to the amount of inserted lines.

While Mossad is still able to obfuscate submissions and reach the targeted similarity scores, running it with our defence mechanism integrated into the plagiarism detector affects the usability of Mossad. Due to match merging compensating changes on the token sequence up to a degree, Mossad needs significantly more insertions and generation time. The increase in generation time can be somewhat negligible as the student just needs to wait longer until the obfuscation process is finished. The factor of increase on the other hand greatly affects the usability. When a student hands in a submission that is twice or thrice as long as the others, it might raise suspicion even if the similarity with the original is sufficiently low.

How does match merging defend against Mossad? With the metrics shown and discussed, we can answer the first question of goal one: We showed that match merging is capable of increasing the similarities of plagiarized submissions (plagiarism × original) from 20 percent to around 80. With scores like this, these submissions clearly stand out and will be found by the education personnel. On the other hand, match merging leaves the similarities of innocent submissions (original × original) mostly unaffected, with the largest increase being 6 percent.

When Mossad is used for obfuscation with match merging already running in JPlag, the resulting length of plagiarized submissions increases in such a way that they raise suspicion even with low similarity scores.

Conclusively, we showed that match merging is capable of defending against Mossad.

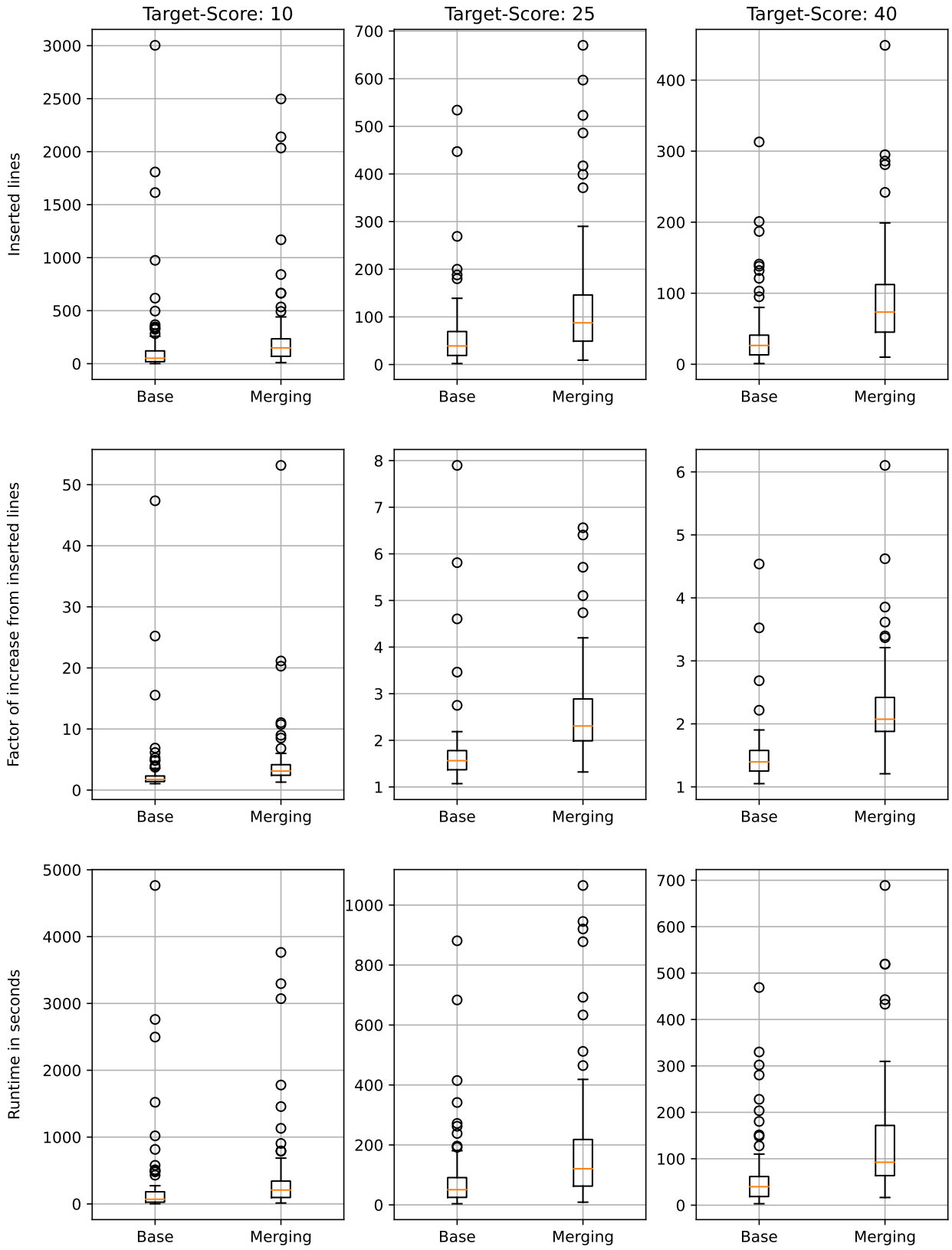


Figure 5.3: Observing the changes in Mossad's usability when executed with match merging. Changes to the amount of inserted lines, factor of increase from them and runtime are assessed for different target similarities.

JPlag-GEN This section presents the result of the experiments with JPlag-GEN generated plagiarism. First, we will cover the *Progpedia* test sets with all three obfuscation modes and then the *TicTacToe* test set with insertion mode. All plots in this section are constructed identically to the ones from the Mossad Soco testset. As *Java* has a minimum token match of 9, neighbor lengths range from 8 to 1 and **O** still refers to the base version of JPlag without match merging.

Figure 5.4 shows the performance on task 19 and task 56 from *Progpedia* with the insertion obfuscation mode. We can see that plagiarism × original have base similarities of around 10 percent for task 19 and around 18 percent for task 56. Submissions in the original × original start with a median of 0 and a mean of 10 percent. As we fade match merging in by decreasing the neighbor length, similarities in both categories rise. Similarities of plagiarism rise higher than similarities of originals, so the difference between both categories increases when neighbor length is lowered. The difference reaches its peak at neighbor length 1 where plagiarism from task 19 reaches 40 percent and the ones from task 56 reaches 70 percent. As both categories are relatively close to each other in the beginning the interquartile range is intertwined, but at neighbor length 4 the difference begins to increase for task 56.

From these observations, we can conclude three things about the insertion mode:

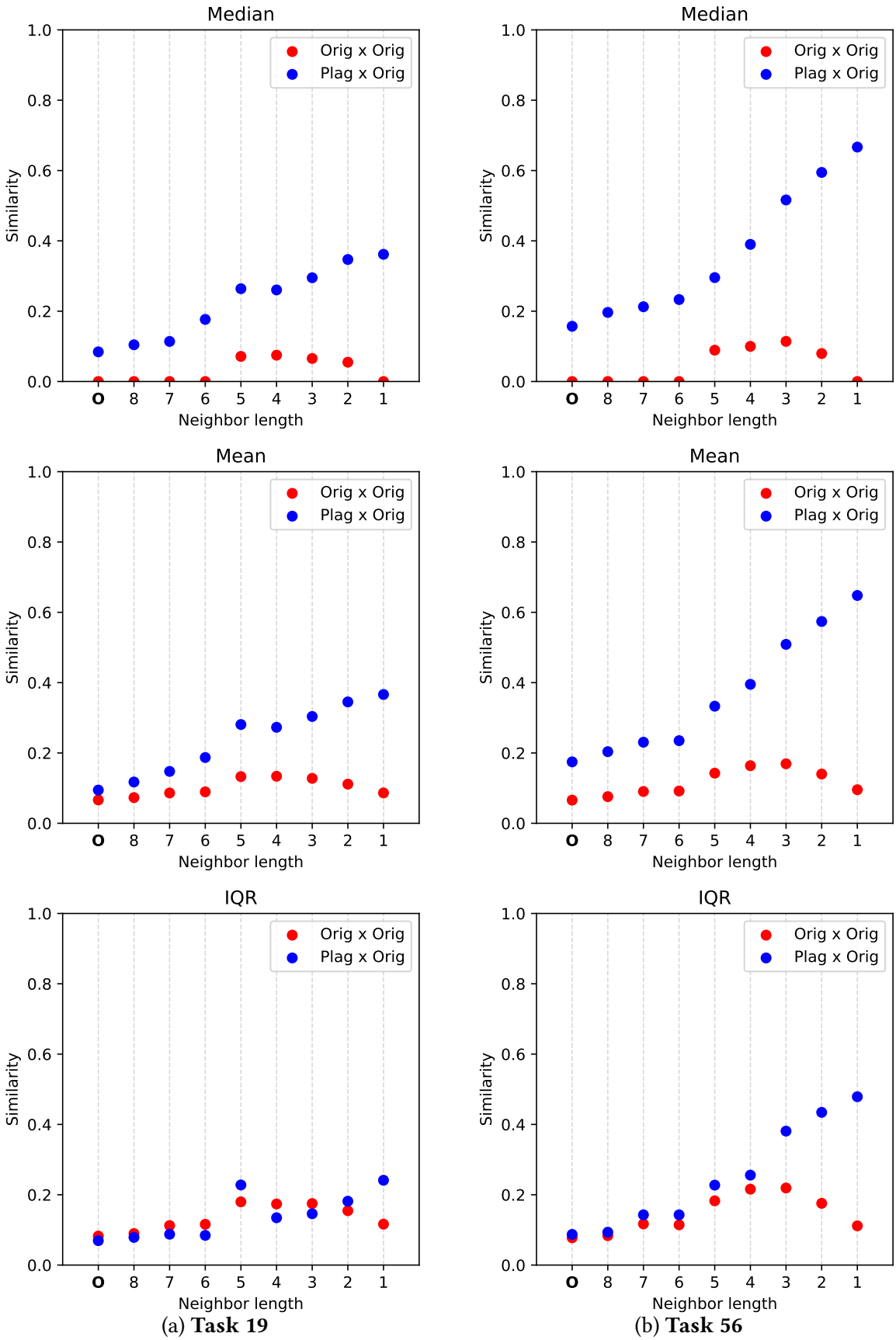
1. The more we use match merging, the more the difference between the two categories gets. This is due to true positive similarities rising faster than false ones, and false positives decreasing after a certain point.

2. Our performance is up to a degree dependent on the dataset as the performance of match merging is better on task 56 than on task 19.

3. We perform worse on JPlag-GEN with insertion mode, than on Mossad. This was expected as Mossad inserts lines into random spots until the similarity hits a threshold and JPlag-GEN insert one line into every possible spot, which lowers the score more drastically and requires more merges to revert.

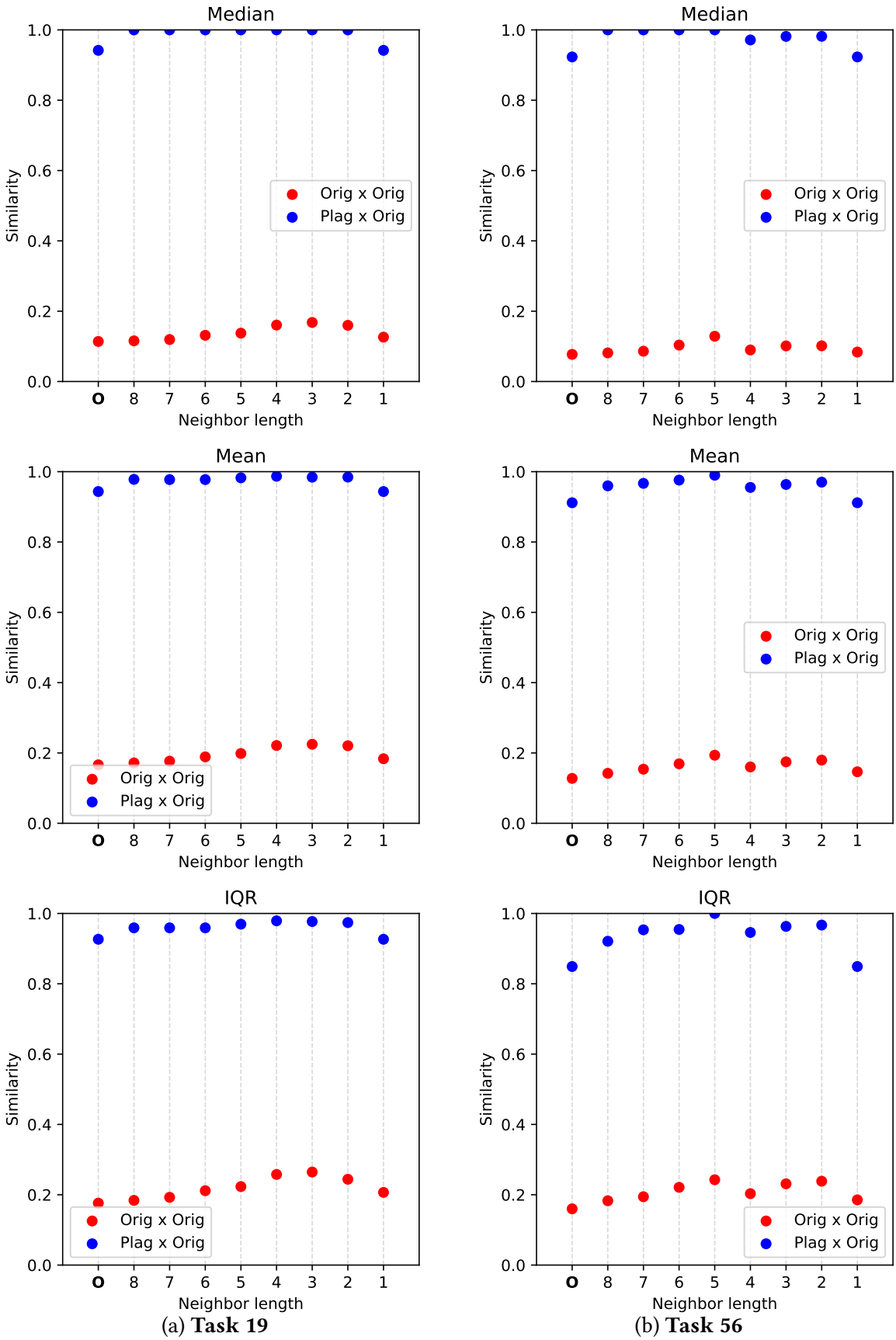
Figure 5.5 shows the performance on JPlag-GENs reordering mode. We can see that similarities of plagiarism × original start above 90 percent and reach 100 percent for the median, after just one neighbor length decrease. Valid submissions have a base similarity of about 18 percent and are mostly invariant to match merging with the peak at neighbor length 4 with 23 percent. The metric show that reordering on its own is a weak attack, that does not require many countermeasures. This was expected because the amount of possible reordering is limited by JPlag-GEN's main constraint, being a semantic-preserving obfuscation tool. Beyond that, the internal comparison algorithm, here greedy string tiling produces matches regardless of relative positions in the sequences. Both attributes of reordering result in match merging being capable of negating it with neighbor length $MTM - 1$.

Lastly, Figure 5.6 shows task 19 and task 56 with both obfuscation modes combined. First *reordering* was applied, then *insertion*. We can observe that the plots look almost identical, to the ones from Figure 5.4. The only difference is that the similarities of plagiarism × original are slightly lower with a maximum difference of under 10 percent at its peak. This supports our observation from Figure 5.5 where we noticed, that reordering itself is a weak obfuscation attack because we can defend against both attacks almost as well as against insertion only.



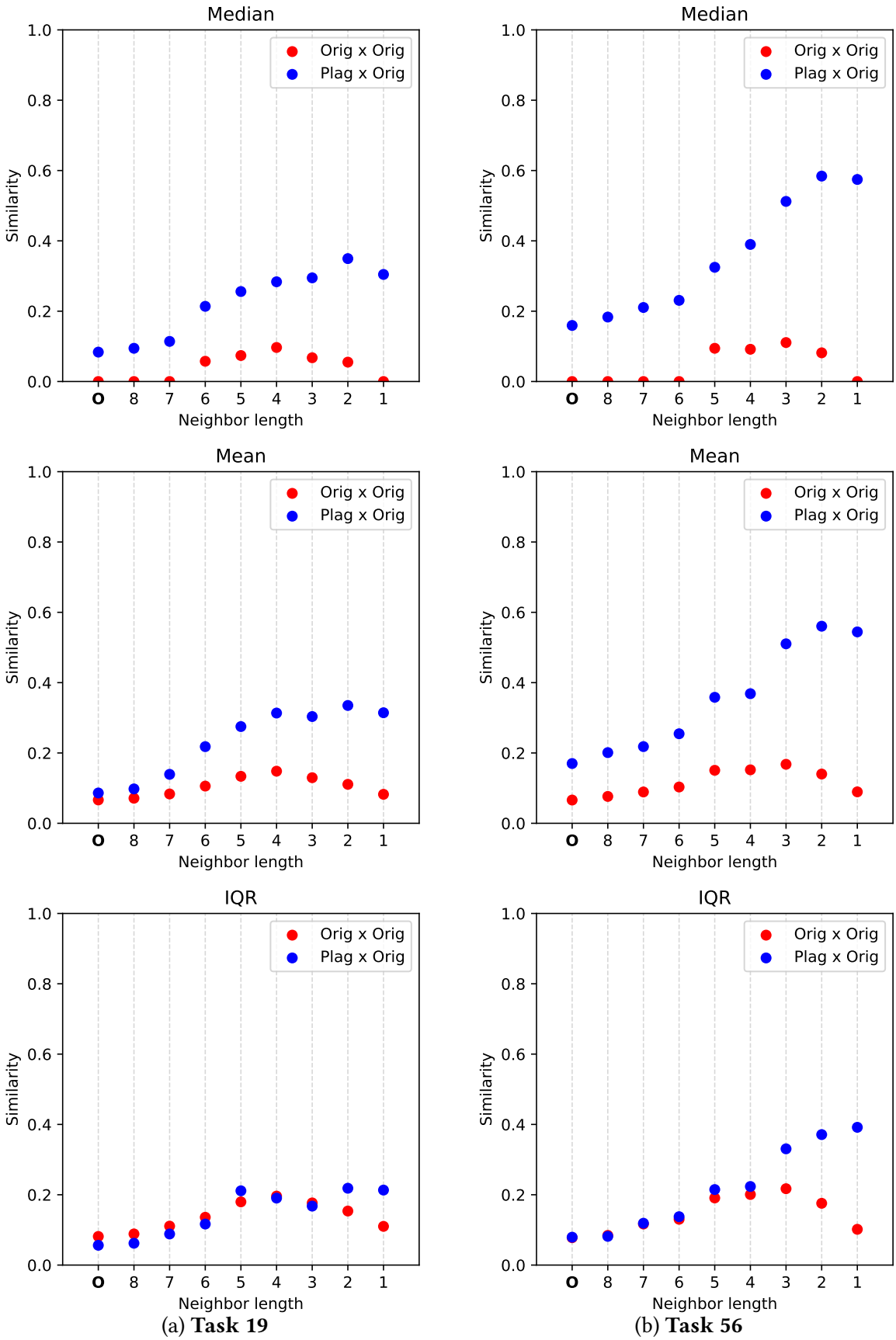
(a) **Task 19** (b) **Task 56**

Figure 5.4: Similarities on Progpedia with plagiarism from JPlag-GEN's insertion mode



(a) **Task 19** (b) **Task 56**

Figure 5.5: Similarities on Progpedia with plagiarism from JPlag-GEN's reorder mode



(a) **Task 19** (b) **Task 56**

Figure 5.6: Similarities on Progpedia with plagiarism from JPlag-GEN's combined mode

We now shift our focus to the plagiarism generated from the TicTacToe test set with insert obfuscation. Submissions in this set differ from the ones from Progpedia, as these are around 6 times as large and contain multiple files per submission.

We can see how the similarities scores behave in Figure 5.7. For all aggregate functions, true positives and false positives start out very close to each other. This remains true until we use match merging with neighbor length 5 for the median and mean, and neighbor length 3 for the interquartile range. At these points, the difference between the scores starts to increase until it reaches its peak at neighbor length 2 for the median and neighbor length 1 for the mean and interquartile range. At the peak submissions from the plagiarism × original category have a median similarity of 23 percent while the ones from original × original have 3 percent.

Although match merging increases the difference between true positives and false positives, the performance is worse than the results on task 19 and task 56. This is most likely due to the submissions being substantially larger, which gives JPlag-GEN more possible insertion spots. Because JPlag-GEN inserts one line into every possible spot, the amount of merges needed to increase similarity to a notable level rises. Nevertheless, with a peak difference of 20 percent, true positives will likely raise suspicion, while leaving false positives under 5 percent.

How does match merging defend against different attacks from JPlag-GEN? With the metrics shown and discussed, we can answer the second question of goal one: We saw that match merging can defend against JPlag-GEN's insertion mode, as the difference in similarity between true positives and false positives rises, the more we use match merging, denoted by a lower neighbor length. We observed that the reordering mode doesn't pose much of a threat as it is only capable of reducing true positive similarities to above 90 percent. As a direct consequence, match merging performs on combined attacks almost as well as on insertion mode. Additionally, we showed that it also works on large submissions spanning multiple source code files.

Conclusively, we showed that match merging is capable of defending against all obfuscation modes from JPlag-GEN.

Make JPlag more resilient against semantic preserving obfuscation attacks We answered both questions for goal one, by showing that match merging is capable of defending against Mossad as well as JPlag-GEN. We thus see the goal of making JPlag more resilient against semantic preserving obfuscation as fulfilled. Additionally, we reduced the usability of Mossad itself when it is operated with the enhanced JPlag.

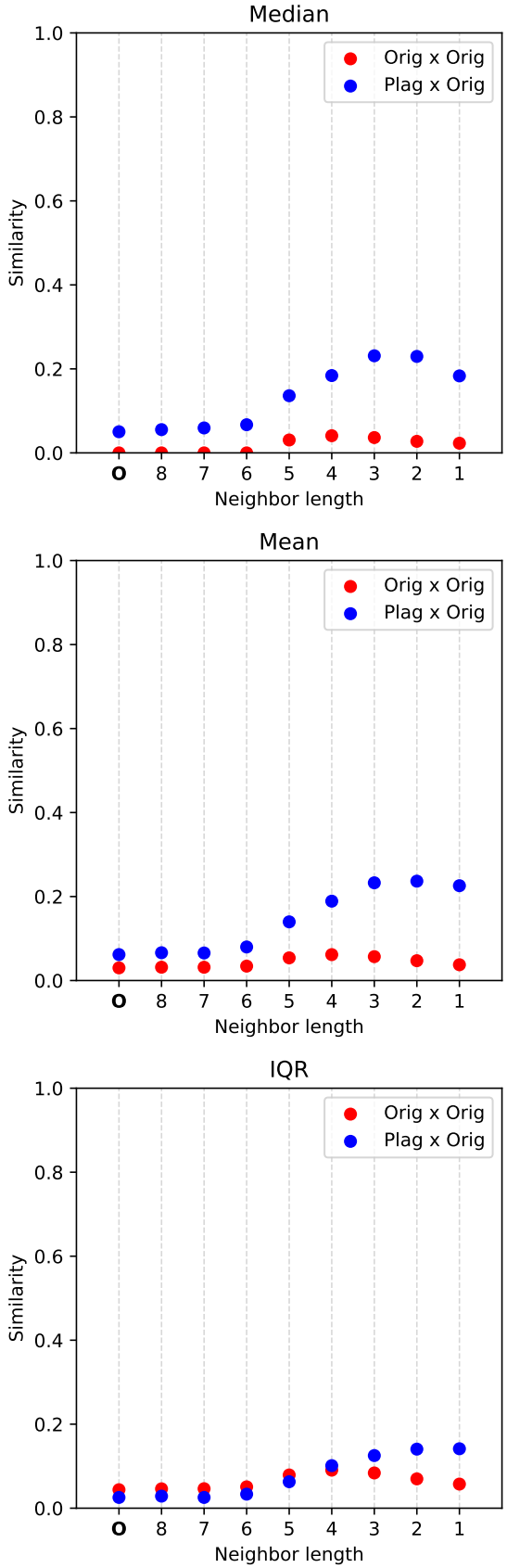


Figure 5.7: Similarities on TicTacToe with plagiarism from JPlag-GEN's insertion mode

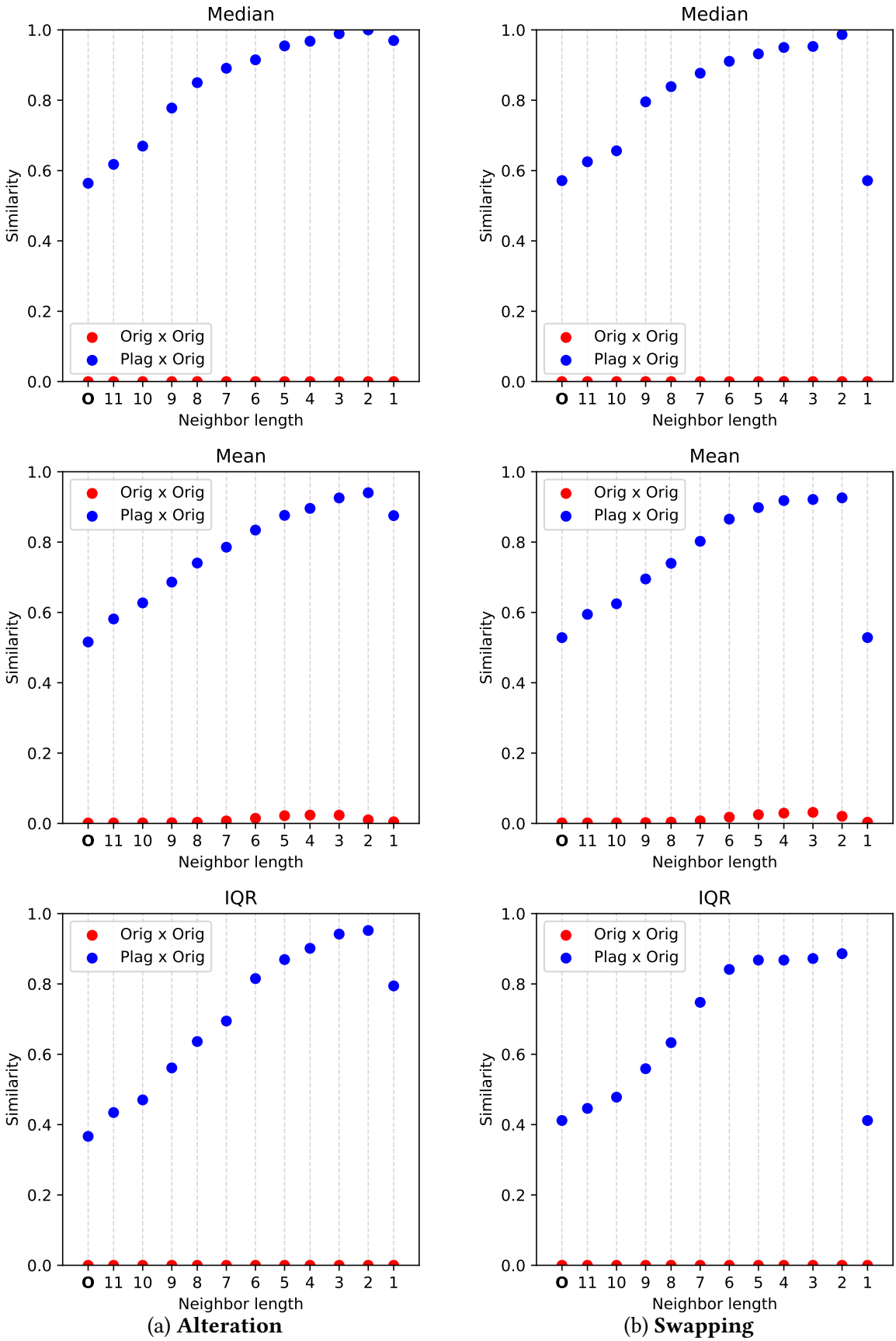
5.3.2 Resilience against semantic agnostic obfuscation attacks

In this section, we assess the capability of match merging to aid JPlag against semantic agnostic obfuscation attacks. To this end, we designed the simulated attacks Alteration and Swapping. Alteration can change the types of random tokens and Swapping can swap random pairs of tokens. Both attacks can be adjusted by setting how much of the submission will be changed. We experimented with 10, 20 and 30 percent for both based on the Soco dataset. The findings will be presented and discussed grouped by percentages instead of attacks.

10 percent change Figure 5.8 shows Alteration and Swapping adjusted to change 10 percent of the plagiarism. Both attacks exhibit similar behavior. Submissions from the plagiarism × original category start above 50 percent similarity. These steadily increase while reducing neighbor length, until they reach the peak above 90 percent for neighbor length 2. While with neighbor length 1 the similarity only slightly decreased for Alteration, for Swapping it falls back to the initial values at around 50 percent. Submissions from the original × original category start around 0 percent similarity and reach their peak at neighbor length 3 with 5 percent.

20 percent change Figure 5.9 displays the changes in similarities with 20 percent changes. For the median and mean true positives have base similarities at around 20 percent. For the interquartile range, true positives start out at 0 percent for both. As we fade match merging the score begins to rise until it peaks at neighbor length 2 for Alteration and neighbor length 3 for Swapping. At the peak, true positives have similarities around 80 percent. After the peak scores decrease drastically and reach 60 percent for Alteration and 20 percent for Swapping. Additionally, we notice, that for the interquartile range true positive scores begin to rise not until neighbor lengths 9 and 7 respectively, which indicates that there are outliers in the true positives that are strongly obfuscated. False positives are largely unaffected with the highest value of 5 percent at neighbor length 3.

30 percent change Figure 5.10 presents the final result for simulated attacks at 30 percent and differs quite from the predecessors. The score of plagiarism × original have base similarities at around 0 percent. They stay this low until neighbor length 6 for Alterion and neighbor length 7 for Alteration. From then on the scores follow an upside-down parabola with the peak at neighbor length 3 with a score of 70 percent. After the peak the score fall to 20 percent for Alteration and to 0 percent for Swapping. For the outlier-affected interquartile range results are worse, with only 3 neighbor length values with result in a true positive score above 0 percent. False positives behave almost identically as for 10 and 20 percent, being almost unaffected and having their peak at neighbor length 3 with under 5 percent similarities.



(a) **Alteration** (b) **Swapping**

Figure 5.8: Similarities on Soco-Dataset with 10 percent changes

How does match merging defend against alteration and swapping? Since both simulated attacks exhibit remarkably similar behaviors across the three change percentages, we will address their corresponding questions collectively.

While reviewing the result, we noticed two distinct intricacies when using match merging against our simulated obfuscation attacks:

1. While increasing the change percentage from 10 to 20 to 30 percent only lowers the peak true positive score from 90 to 80 to 70 percent, the area under the function represented by the true positives decreases drastically. This is somewhat expected as changing more and more of a submission renders weaker configurations of match merging incapable of significantly improving score. We need a neighbor length of 3 across all percentages to achieve the highest performance.

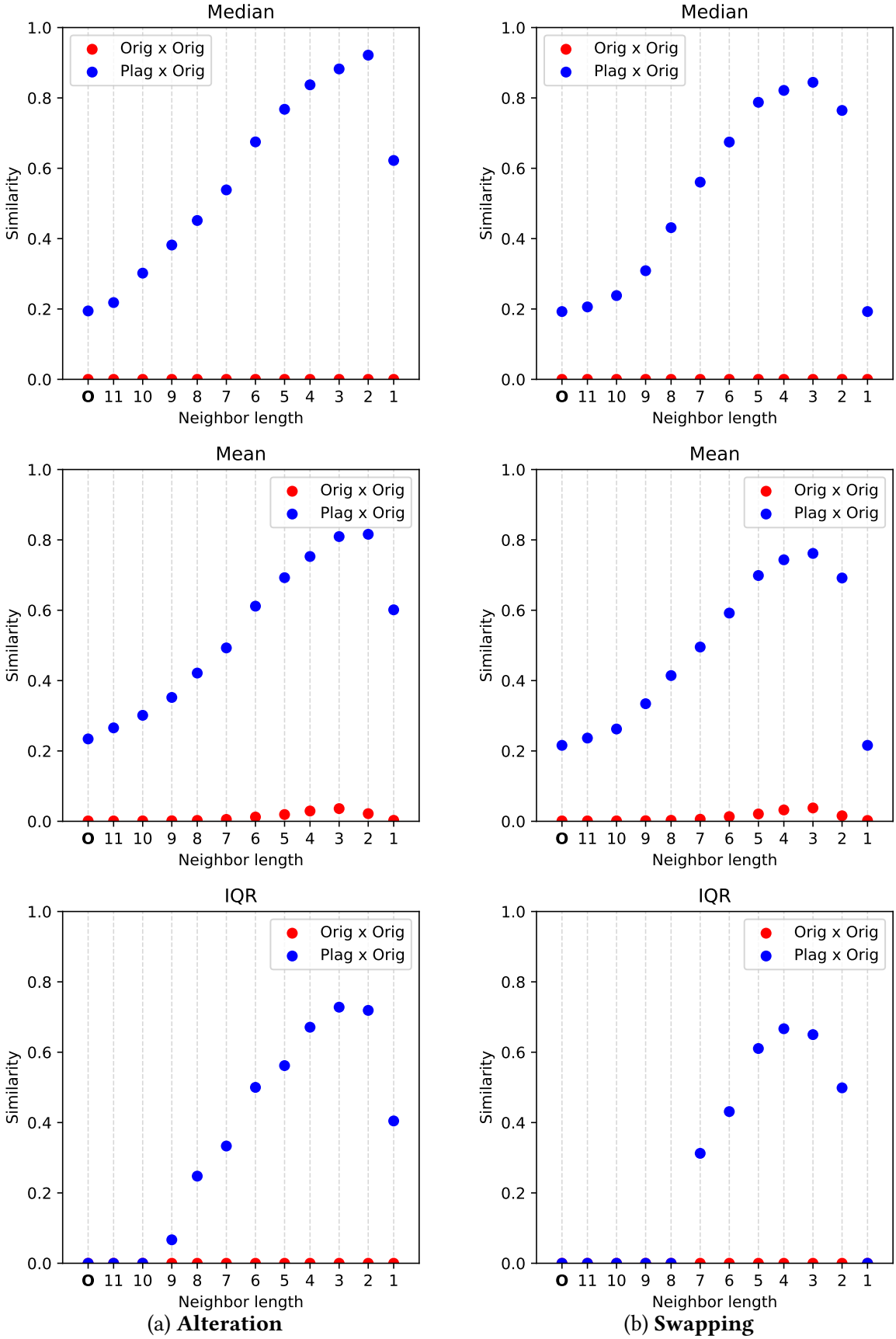
2. At lower neighbor lengths like 2 or 1 similarity scores for true positives collapse. This is due to submissions being changed so much, that merging removes most tokens in the sequences which results in there being not much left to link the original to obfuscated plagiarism.

Make JPlag more resilient against semantic agnostic obfuscation attacks What does this mean for match merging's defensive capabilities?

Even when we change 30 percent of our submissions without any restriction whatsoever, we can increase true positive similarities to over 70 percent. As we leave the false positive scores mostly unaffected, with a peak of 5 percent, this makes differentiating valid from malicious submissions very reliable.

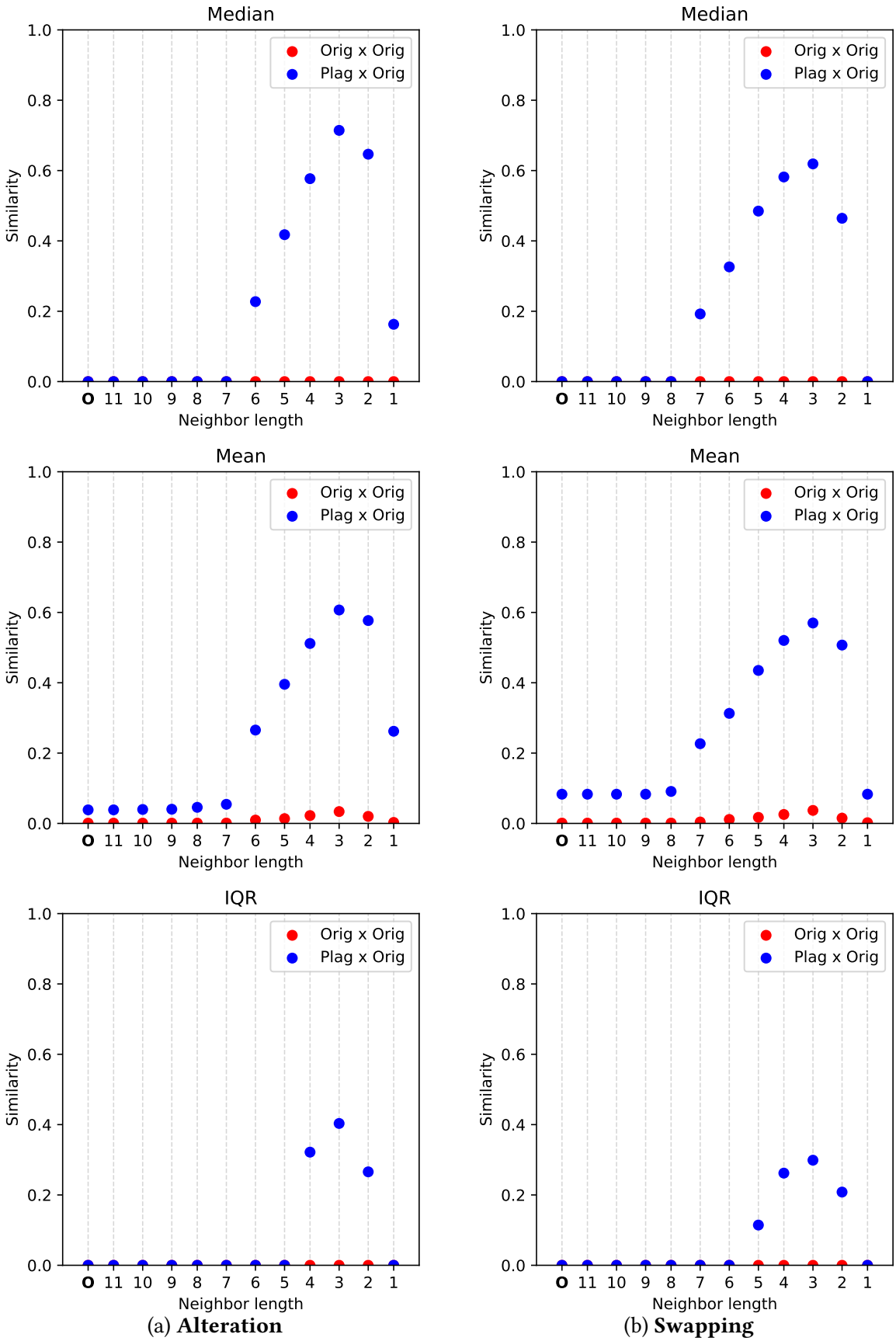
Unhinged changes in submissions cause collapsing of similarity scores at very low neighbor lengths. Hence, it is advisable to choose more conservative combinations with neighbor lengths of 3 or 4.

In the end, we successfully achieved Goal 2, enhancing JPlag's resilience against semantic-agnostic attacks.



(a) **Alteration** (b) **Swapping**

Figure 5.9: Similarities on Soco-Dataset with 20 percent changes



(a) **Alteration** (b) **Swapping**

Figure 5.10: Similarities on Soco-Dataset with 30 percent changes

5.3.3 Resilience against AI-based obfuscation attacks

The third goal is split into obfuscation prompts and fully generating submissions:

Obfuscation prompts We evaluated the power of obfuscation prompts by looking at the base similarities and similarities with match merging. Figure 5.11 displays all 15 prompts sorted by the base similarity to indicate how strong a prompt initially is. For each prompt, we display the mean of base and merging performance. The numerical values can be seen in Table 5.1.

First, we clustered the prompts via their initial impact using k-means and used the silhouette coefficient to find the optimal amount of clusters k. This yielded two distinct clusters which we categorized as low-impact and high-impact prompts:

- low-impact prompts = $[1.a, 2, 3, 4, 5, 7, 9, 10, 12, 13, 14]$
- high-impact prompts = $[1.b, 6, 8, 11]$

For the low-impact category, the mean base similarity is 0.9156 which is increased to 0.9526 when using match merging with the parameter combination ($neighborLength = 2 \times gapSize = 6$). Although match merging only makes a difference of 4 percent submissions obfuscated via these prompts will most likely be caught as they have similarities of over 90 percent.

The high-impact obfuscation prompts achieve a mean base similarity of 0.5580 which is substantially lower than before. Using match merging with the same parameter combination as above the similarities are increased to 0.6671 which is about 11 percent higher.

We must examine the reasons behind the remarkable potency of high-impact prompts:

1.b Prompt 1.b is the second prompt of a combination where the LLM should first change the could to look different and then refactor the interim result.

6 With prompt 6 the LLM is instructed to read the source code, understand its innards and write a version of its own without relying on the original. As this prompt recreates the submission it is naturally very strong. Furthermore, the LLM is capable of understanding submission from the TicTacToe set with over 400 lines.

8 Prompt 8 replicates the function of Mossad by prompting the LLM to insert dead code into random spots.

11 Prompt 11 is an engineered prompt where we meticulously specified what the LLM should do to break up blocks of matching tokens. Among the methods were reordering, structure changes and insertion which must not change the output.

How does match merging defend against obfuscation prompts? Most prompts have a very low impact and fail to reduce the similarity of plagiarism × original below 90 percent. With scores this high match merging is not needed to easily identify plagiarism and catch cheaters.

For prompts that successfully obfuscate, we can improve the scores by about 11 percent. Although this alone might not be enough to reliably spot plagiarism, it aids the process and should be combined with other defence methods.

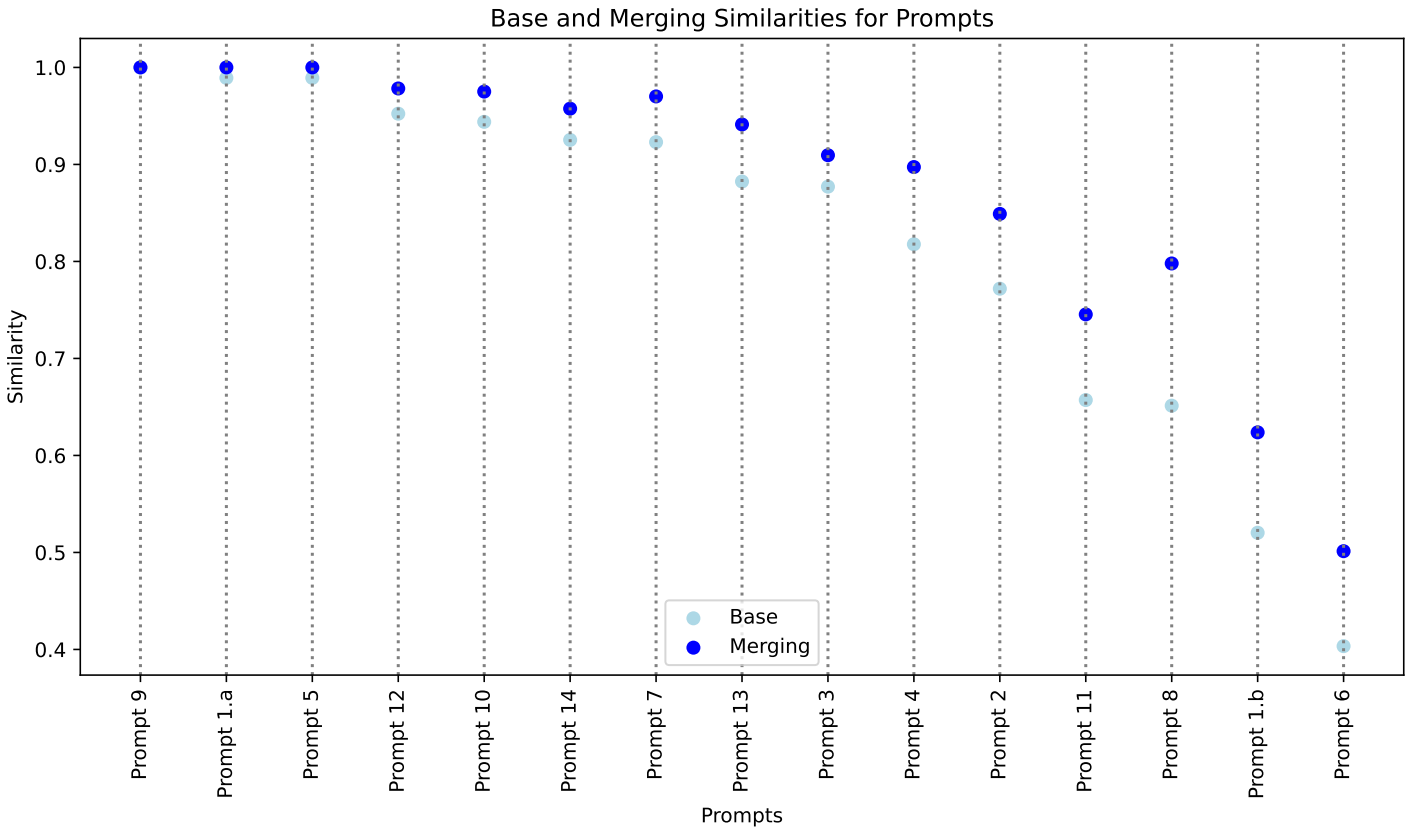


Figure 5.11: Plotted base and merging similarities per prompt, sorted by the base similarity to indicate their initial potency

	Base	Merging
Prompt 1.a	0.9892	1.0000
Prompt 1.b	0.5203	0.6238
Prompt 2	0.7719	0.8490
Prompt 3	0.8771	0.9095
Prompt 4	0.8177	0.8973
Prompt 5	0.9892	1.0000
Prompt 6	0.4034	0.5013
Prompt 7	0.9230	0.9701
Prompt 8	0.6513	0.7979
Prompt 9	1.0000	1.0000
Prompt 10	0.9439	0.9751
Prompt 11	0.6571	0.7454
Prompt 12	0.9523	0.9783
Prompt 13	0.8824	0.9412
Prompt 14	0.9253	0.9575

Table 5.1: Numerical base and merging similarities per prompt

Fully generated submissions We fully generated 10 submissions from the base game task description of the TicTacToe dataset. Even though they are generated through the same prompt, they all differ as ChatGPT has a degree of randomness in its answers. In the following, we evaluate the similarity scores between the fully generated submissions.

Figure 5.12 contains two boxplots of similarity distributions between fully generated submissions. The left displays base JPlag and the right JPlag with integrated match merging whose parameter combination was $neighborLength = 2 \times gapSize = 6$.

The base similarity distribution is centred around its median at 46.5 percent and the interquartile ranges from 37 to 55 percent. The spectrum of similarity scores within this distribution ranges from a minimum of 15 percent to a maximum of 78 percent.

In contrast, the distribution of similarity with match merging is notably higher, centred around its median at 68 percent. The interquartile ranges from 58.5 to 82 percent and the longer range indicates a more concentrated and denser distribution. The lowest score is now at 42 percent and the highest at 100 percent.

We gained two key insights from the boxplots:

1. With match merging we increased the median similarity score significantly by 22 percent.
2. With match merging there is no score left under 42 percent.

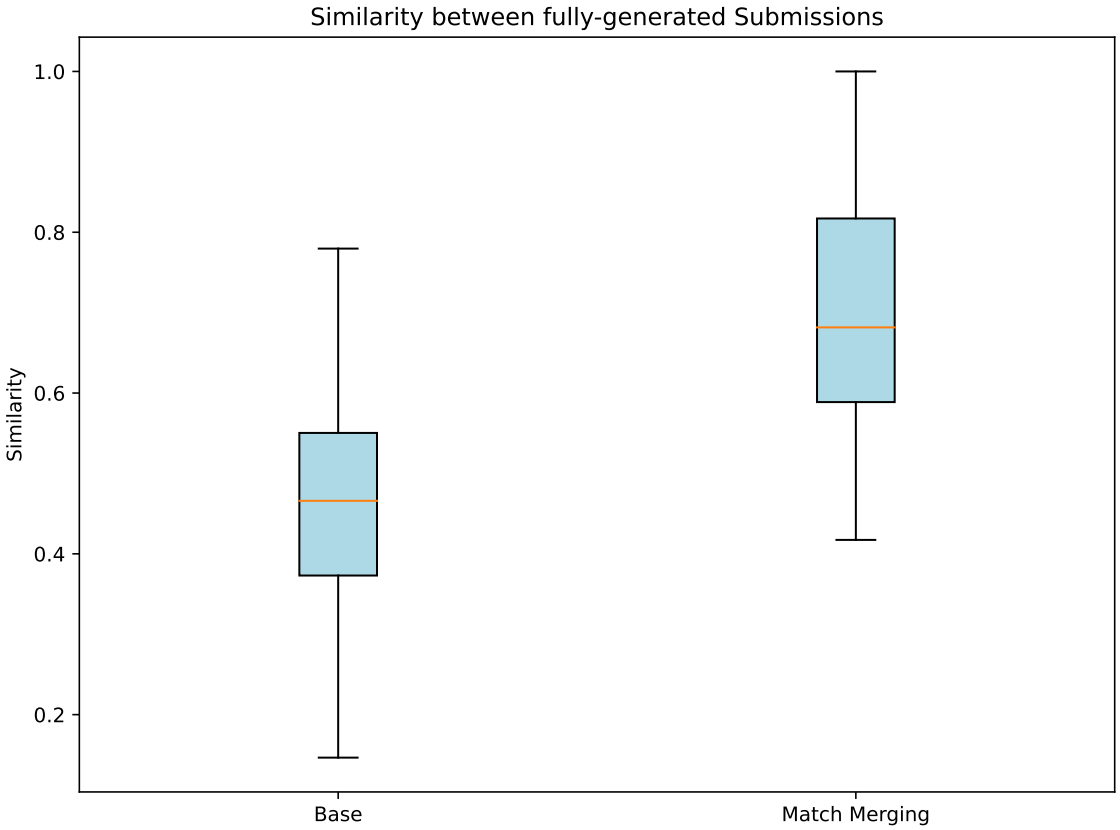


Figure 5.12: Base and merging similarities between fully-generated submissions

How does match merging defend against fully generated submissions? We can now assess the dangerousness of fully generating submissions to pass coding assignments. Our findings are based on the capabilities of GPT-3.5, which represents the current state-of-the-art in natural language processing. However, due to the significance of large language models in computer science, they undergo frequent improvements and advancements.

First of all, the LLM was not capable of generating the complete Tic Tac Toe task and we had to compromise for the base game version. Although the resulting source codes were syntax-error-free and provided the specified gameplay functionality, they did not pass the automated grading test, due to not adhering to the required formatting.

The similarities of fully generated plagiarism to each other ranged from 15 percent to 78 percent for the base version of JPlag, which would result in some of them remaining unnoticed. With match merging, we increased all similarity scores to above 42 percent and raised the median by 23 percent. This significantly increases their susceptibility to detection, particularly considering that legit submissions to one another exhibited a maximum similarity of less than 10 percent, as depicted in Figure 5.7.

In summary, the danger of fully generated submissions is manageable and utilizing match merging as a defence mechanism aids in reducing it further.

Make JPlag more resilient against AI-based obfuscation attacks We showed that match merging is capable of making AI-based plagiarism more prone to detection by intelligently raising similarities. Furthermore, we discussed the limitations of LLMs, resulting in most prompts having a negligible impact on the similarity scores and fully generated submissions not being able to pass grading tests. Thus we consider this goal fulfilled.

5.3.4 Maintain performance of JPlag

The last goal of match merging is to maintain the performance of JPlag. Usability is an important factor because even a perfect defence mechanism would not be much of help if it quadrupled the runtime of the plagiarism detector.

To assess the changes in runtime, we enhanced our evaluation script to measure the seconds between executing the plagiarism detector via a subprocess and it returning the calculated similarities. With this, we ran a full-grid search on four testsets while capturing the respective runtimes.

The results of this evaluation can be seen in Figure 5.13. Each subplot stands for a test set and shows neighbor length in relation to runtime in seconds. The leftmost neighbor length **O** stands for off and displays the performance of base JPlag without match merging and consists of a single point. The neighbor length values ranging from $MTM - 1$ to 1 resemble match merging and each has 20 different points, one for each gap size from 1 to 20. Each plot contains all parameter combinations from the full-grid search in its test set as well as a regression line. We can notice two distinct behaviours in the plots:

1. The runtimes for test sets created with JPlag-GEN slowly increase linearly when fading in match merging by lowering neighbor length. For *Progpedia* task 19 runtime increased from 9.5 seconds to 11 seconds, for task 56 from 14.5 seconds to 16 seconds and lastly for *TicTacToe* from 8 seconds to 10 seconds.

2. The runtime for MossadSoco does not follow this pattern and even gets lower up to neighbor length 6 and then increases up to neighbor length 1. This erratic behavior likely stems from Mossad's random nature. The base runtime is 15.75 seconds and goes up to 16.25 seconds when using match merging with neighbor length 1.

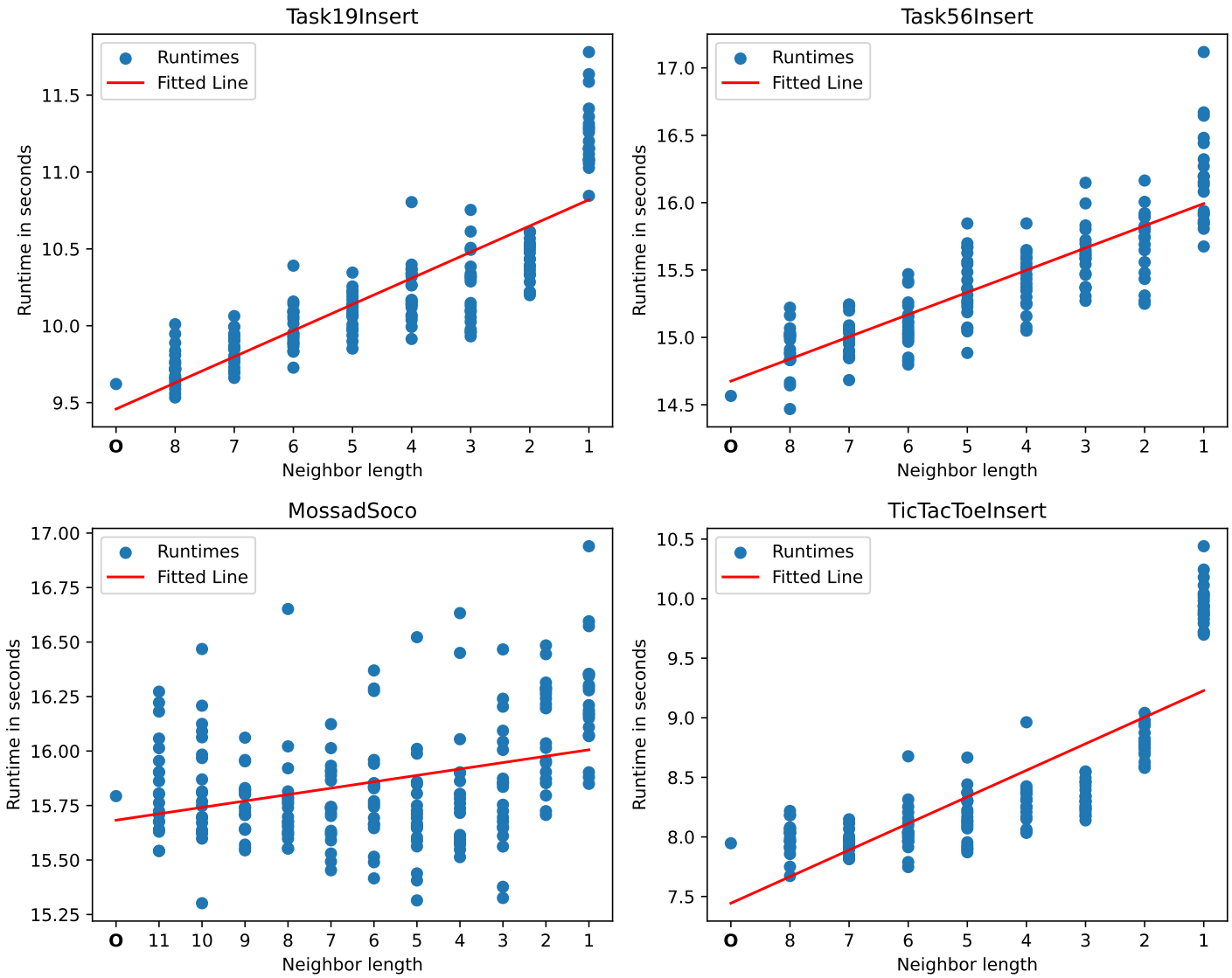


Figure 5.13: Change in JPlag's runtime for different parameter combinations

How does match merging affect the performance of JPlag? We can now answer the single question for this goal: Even when running match merging with the most powerful combination ($neighborLength = 1 \times gapSize = 20$) we notice a mean runtime increase of 13.5 percent which translates to execution taking about 1.375 seconds longer than before.

Based on our findings we consider the final goal fulfilled and move on from the Goal-Question-Metric plan to the ablation study and discussion on default hyperparameters.

5.3.5 Ablation study

In chapter 4, strategic decisions were made in designing the foundational heuristic for match merging. At two critical junctures, we deliberated and selected specific approaches over alternatives. In this concise ablation study, we meticulously rationalize each of these choices by evaluating their impact on the performance of our defense.

Average vs. maximum gap size Gap size is one of the two parameters for match merging and denotes the maximum number of tokens there can be between two neighboring matches to be candidates for merging. As the gap length can vary between left and right submissions, we must aggregate both values. We experimented with the average and the maximum:

For example, if there is 1 token separating the neighbors in the left and 3 tokens separating the neighbors in the right submission the average would be 2 while the maximum would be 3.

We have tried both variants on the test sets task19Insert and task56Insert and the performances are plotted in Figure 5.14. The performance of averaging is colored in deep blue and deep red, while the performance of taking the maximum is colored in light blue and light red. If only one color is visible for a category, both aggregates produce the same performance.

We can notice two phenomena from the plots: In 70 percent of all cases, both aggregates exhibit identical performance levels across various datasets and metrics. For the remaining 30 percent the average reliably outperforms the maximum.

Considering these findings, the average emerges as the preferred aggregate function for differing gap sizes.

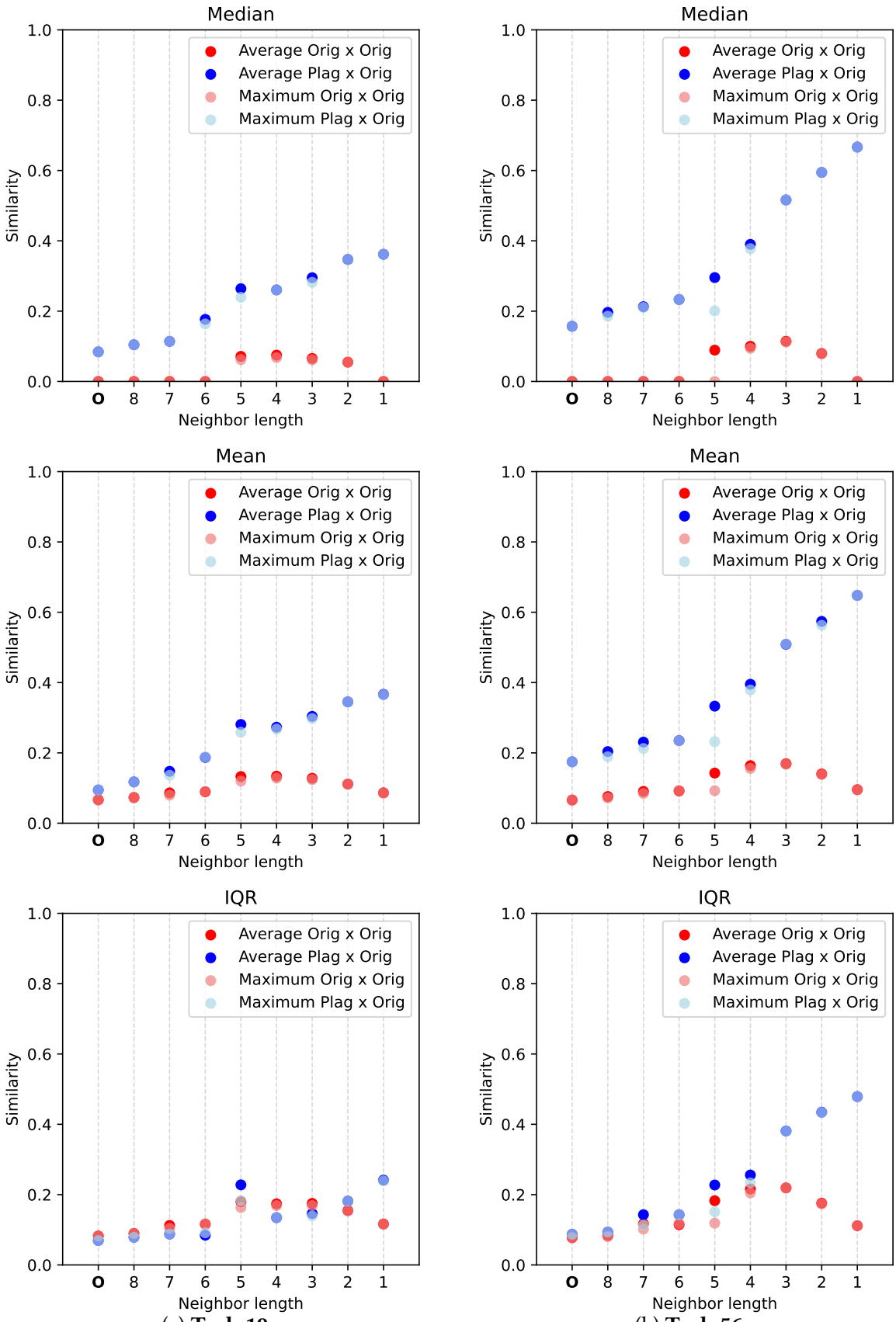
Merge across file boundaries The second design choice was if we should allow merges across file boundaries. If a submission consists of multiple files, the tokenized sequences are put into one large list and the plagiarism detector inserts a pivot token *FILE_END* to signalize that the file changes. The comparison algorithm does not produce matches with tokens from two files and achieves this by checking for the pivot element. Match merging would be capable of merging across file boundaries but we must weigh up the costs and benefits.

Figure 5.15 shows the performance of match merging based on the TicTacToe test set where we once forbid and once allowed file boundaries merges. The similarity scores with forbidden merges are colored in dark blue and dark red, while the ones with allowed merges are colored in light blue and light red.

We can see that most of the time both variants achieve identical performances. For the few occurrences where allowed merges outperform forbidden merges, the difference is very slim with a maximum of 2 percent. Hence the benefits of allowing merges across file boundaries are rather slim.

The downside of allowing these merges is that we remove pivot tokens from the token sequences which effectively lowers the amount of files the plagiarism detector thinks the submission consists of. This results in unintended behavior like similarity score going beyond 100 percent and the report viewer, which shows the results, being unable to correctly show matching blocks.

Due to negligible benefits and indisputable downsides, we opted to forbid merges across file boundaries.



(a) **Task 19** (b) **Task 56**

Figure 5.14: Progpedia: Average vs. maximum gap size

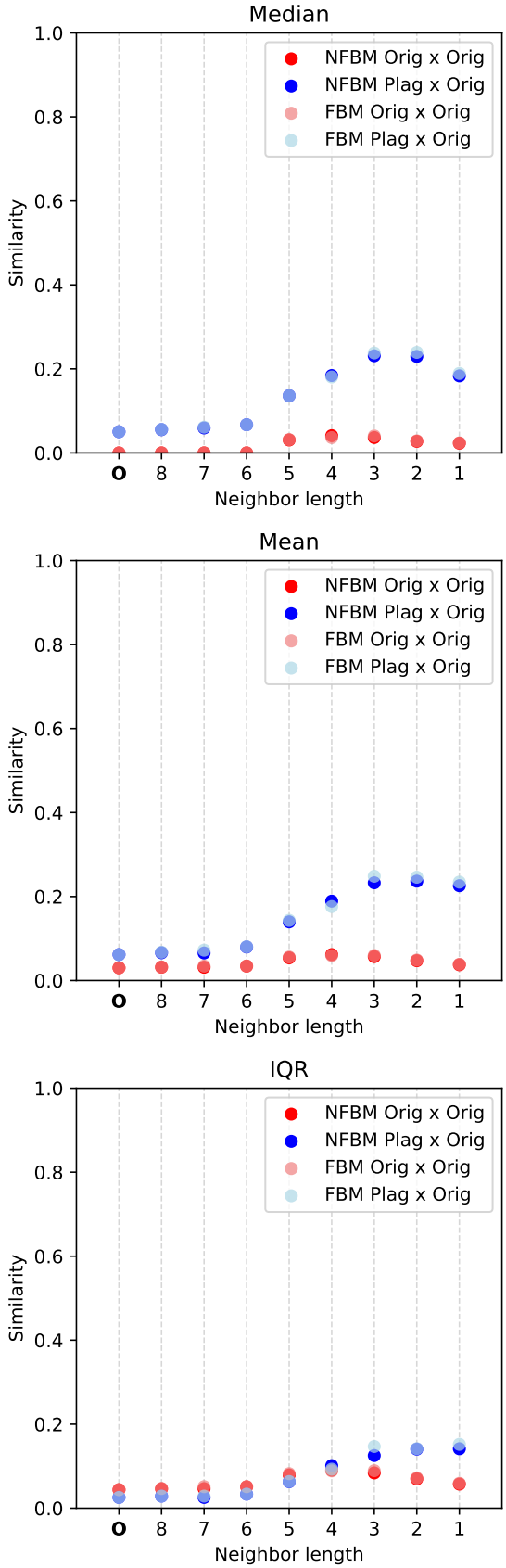


Figure 5.15: TicTacToe: No file boundaries merge vs. file boundaries merge

5.3.6 Default hyperparameters

After conducting experiments and analyzing the results, it is crucial to determine the optimal parameter combination (neighborLength × gapSize) to set as the default values for JPlag. We performed an extensive grid search across 17 distinct test sets, identifying the combinations where the disparity between true positives and false positives is most significant. In cases where multiple optimal combinations exist, we prioritize the more conservative option. Specifically, we select the combination with a higher neighbor length or a smaller gap size.

The results of our analysis, showcasing the 17 best combinations, are presented in Table 5.2.

Dataset	Testset	Neighbor Length	Gap Size
Ai-Soco	MossadSoco	2	10
Progpedia	Task19Insert	1	2
Progpedia	Task56Insert	1	2
Progpedia	Task19Reorder	8	6
Progpedia	Task56Reorder	8	5
Progpedia	Task19Combined	1	1
Progpedia	Task56Combined	1	1
TicTacToe	TicTacToe	2	6
Ai-Soco	Alteration10	2	3
Ai-Soco	Alteration20	2	9
Ai-Soco	Alteration30	3	20
Ai-Soco	Swapping10	2	6
Ai-Soco	Swapping20	3	13
Ai-Soco	Swapping30	3	14
Progpedia	Task19InsertMax	1	3
Progpedia	Task56InsertMax	1	4
TicTacTo	TicTacToeFBM	2	6

Table 5.2: Best parameter combination per test set

The values for each parameter were aggregated to determine the default combination for JPlag. We opted for the median as the aggregation method due to its resistance to outliers, unlike the mean.

The resulting default combination, ($neighborLength = 2 \times gapSize = 6$), was derived through this process and is now implemented as the standard configuration for JPlag. Experiments assessing the usability of Mossad and performance on AI-based obfuscation techniques were conducted based on this established default combination.

5.4 Threats to validity

We assess the validity of our evaluation based on the four categories as proposed by Runeson and Höst [34]. We will address the threats to validity, explore their mitigations, and provide explain our assumptions.

5.4.1 Internal validity

We identified and managed four threats to internal validity:

Adapting Mossad Mossad relies on the *Clang* compiler to check if submissions still compile after a random insertion and assert that the output remains the same. The output check is done by using the aggressive optimization level 3 for the compiler which removes dead code. The compile process is then halted after generating object files. When there is no difference between the two object files the output remains unaffected.

Recall that the *AI-Soco* dataset was collected from a competitive coding platform. Almost all submissions include the *bits/stdc++.h* library, which itself includes all other libraries. This is particularly helpful for the authors as they don't have to worry about missing includes. The problem is that *bits/stdc++.h* and its included libraries are part of the *GCC* collection. Therefore, *Clang* was unable to compile the source codes and Mossad did not work on the *AI-Soco* dataset.

To levitate this issue, we adapted Mossad to use the *G++* compiler instead, which was able to correctly compile the submissions. To ensure Mossad's functionality remains the same, we used the same optimization level and halted the process after object files were generated. The resulting assembly code is then converted into a readable format, from which we can check if both outputs are identical. Additionally, we manually inspected 10 randomly selected instances of generated plagiarism, in order to confirm that Mossad behaves the same after our changes.

Processing of datasets We opted to not process and clean the datasets, which means that there could be some inherent plagiarism that was not generated by us. This poses no threat to the validity as plagiarism of that nature would be assorted to the original × original category, due to not being marked by us. This means unidentified instances of plagiarism would only increase the false positive similarity scores, which would make the evaluation of our defence mechanism look worse, not better. We suspect that if there was inherent plagiarism in the datasets, they are few and far between, resulting in a negligible impact on the performance.

Cherry picking of submissions We ensured to not cherry-pick submissions for our evaluation. To this end, we choose the first 100 submissions from *AI-Soco* for the assessment of Mossad, Alteration and Swapping. From *Progpedia* we choose tasks 19 and 56, which are the same tasks that Brödel [4] evaluated his work with. For the *TicTacToe* dataset, we once let a Python script randomly choose submissions. Hence, we did not cherry-pick to increase the performance of our work artificially.

Wrong categorization The performance evaluation of our approach relies on the categorization of similarities to either true positive or false positive. To ensure that the evaluation pipeline works as intended, we calculated how many submissions there should be in each category for a given testset size and verified that the pipeline produced the correct labels.

5.4.2 External validity

We identified and managed two threats to external validity:

Selection of datasets We can only assess the performance of match merging based on the datasets used for the evaluation. To investigate the versatility of our defence mechanism across unseen datasets, we selected datasets that vary noticeably from one another, increasing the generalizability of our approach.

Our evaluation encompasses submissions written in both *Java* and *C++*, ranging from 1 to 10 files and spanning 80 to 400 lines of code. Moreover, these submissions originate from diverse backgrounds, including academia and competitive coding environments.

Potency of simulated attacks Alteration and Swapping can be realized as real attacks that operate on the source code. Brödel [4] already showed this with JPlag-GENs reorder mode, which essentially consists of chained line swaps. Alteration in a real scenario could for example change For-loops into While-Loops, or If-statements to Switch-case. As *real* attacks have to keep the output of a submission intact while obfuscating, they are very limited in their capabilities.

Our simulated attacks are not bound by this restriction and are much more potent as they can change submissions in any way. This makes defending against the simulated attacks more challenging than defending against real ones. Thus, our evaluation conducted on simulated Alteration and Swapping is relevant.

5.4.3 Construct validity

We identified and managed two threats to construct validity:

Methodical evaluation We meticulously assessed our approach's performance through a thoughtfully designed methodology outlined in the Goal-Question-Metric plan, as detailed in section 5.1, deviating from arbitrary evaluations.

Meaningful metrics We made sure to evaluate the performance of match merging based on meaningful metrics that have been used before in the context of plagiarism detection papers. Utilizing true positive and false positive similarity scores to access the performance of attacks on and defences of plagiarism detectors have been used by Devore-McDonald and Berger [9], Brödel [4] and Krieg [16]. Furthermore, Brödel and Krieg evaluate the effect of their defence mechanism on the usability of the plagiarism detector by measuring the change in runtime. Finally, Krieg measured how his defences integrated into the plagiarism detector hindered the usability of Mossad. His metrics of choice were changes in the amount of inserted lines and the runtime of Mossad.

5.4.4 Reliability

Finally, we identified and managed one threat to reliability:

Results cannot be reprocuded To ensure the reproducibility of our work we provide this reproduction package [29], which is hosted by Zenodo and includes:

- Testsets with generated plagiarism
- JPlag plagiarism detector with implemented match merging
- Results of full-grid evaluations
- Implementation of simulated obfuscation attacks

We cannot provide Mossad and JPlag-GEN as the authors did not give us permission, as the tools could cause harm. Furthermore, we can neither include the TicTacToe dataset, nor the created plagiarism from it, as students only gave permission to use their work chair internally.

Please be aware that Alteration and Swapping attacks yield random outcomes, which can vary across different systems, even when employing the same seed for the PRNG stream. These discrepancies likely stem from differences in the operating system, Java Virtual Machine, and CPU configurations.

5.5 Limitations

The core idea of Match merging is to merge neighboring matching blocks of tokens to revert automatic obfuscations. This method is inherently restricted to plagiarism detection systems that operate by comparing submissions in the form of token sequences.

Although there are other approaches for plagiarism detection, i.e. graph-based comparisons [17], these tools are primarily irrelevant as most of the state-of-the-art plagiarism detectors used for coding courses rely on tokenization and subsequence matching in the form of Greedy String Tiling or Winnowing [31, 28].

The performance of match merging is affected by two inherent problems of token-based plagiarism detectors:

1. Submissions whose token sequences are shorter than the minimum token match threshold are rejected and cannot be compared. This prevents needless computations as the comparison algorithm would never be able to find matches with submissions this short.

2. When comparing short submissions there is only little difference in structures which leads to a higher false positive similarities. Thus token-based plagiarism detectors work best on longer submissions.

6 Related works

This chapter provides an in-depth exploration of relevant literature, which is categorized into two distinct sections: plagiarism detection research and advancements in bioinformatics.

6.1 Plagiarism detection

Software plagiarism detection is a well-researched topic [21, 1] offering notable related works.

6.1.1 Brute force token deletion

In his bachelor's thesis *Preventing Code Insertion Attacks on Token-Based Software Plagiarism Detectors* [16] Krieg experimented with strategies to make JPlag more resilient against semantic-preserving insertion attacks. Among those was the idea of brute force token deletion.

Insertion attacks lower the similarity between two submissions by inserting meaningless lines which split up matches. The idea of Krieg was to raise the similarity by deleting as few tokens as necessary and thereby hoping to revert the obfuscation. The approach works around two integral parameters: The sliding window length k defines the length of continuous subsequences in both submissions that should be checked. The maximum deletion threshold n defines how many tokens can be removed from the sequence for one check.

Krieg's algorithm now checks for every combination of sliding windows if they can be made identical by deleting almost n tokens from either submission. If a combination fits the criteria the tokens are removed which inherently raises the similarity.

Brute force token deletion had two factors severely limiting its performance:

1. As the windows size k is substantially lower than the length of the individual submissions $|left|$ and $|right|$ the amount of combinations checked is almost quadratic with $|left| - k + 1 \times |right| - k + 1$. Additionally, for each of these combinations, the algorithm checks $k - n$ times if the windows can be made identically. This results in the approach becoming very costly as submissions increase in size.

2. Due to the exhaustive approach having low capacity, the algorithm also raised false positive similarities.

Match merging follows the same idea of removing tokens from the sequences but operates on matches rather than just on the tokenized sequences. This drastically decreases potential merges which keeps the runtime low. As we additionally combine already matched blocks with two parameters to build a heuristic we can increase the capacity of our approach which makes it able to raise true positive similarities while leaving false positives mostly unaffected.

My thesis builds upon and refines Krieg's original idea, representing a seamless continuation of his work.

6.1.2 Token sequence normalization

In *Preventing Automatic Code Plagiarism Generation Through Token String Normalization* [4] Brödel presented JPlag-GEN a novel automatic obfuscation tool against plagiarism detectors and the corresponding defense mechanism *token sequence normalization*. Recalling from chapter 2, JPlag-GEN features two distinct obfuscation modes, insertion of dead code and reordering of independent statements. We explore the concept of token string normalization and discover how it enhances JPlag's resilience against both obfuscation modes.

Brödel's countermeasure takes place in the tokenization step of the plagiarism detector pipeline. It receives the abstract syntax tree from the language module and converts it into a dependency graph, showing the order and data flow of statements. Figure 6.1 shows such a dependency graph for an example snippet. Dead code can be identified easily by observing which nodes have no indegree and are removed from the graph. Independent statements are identified and rearranged according to a standardized ordering given by the token types. This normalization renders obfuscation by reordering pointless. Afterwards, the cleaned and normalized dependency graph gets converted back into an abstract syntax tree and the language module continues its work. Comparisons between submissions are then performed on the normalized token sequence.

Brödel has shown that token string normalization is very effective in countering JPlag-GEN and its two modes. Although the defense mechanism is only available for *Java* submissions its functionality should be applicable to *C++* too, allowing the defense against Mossad.

Token string normalization outperforms match merging on test sets created with JPlag-GEN and this is due to his approach following strict rules which are tailormade for the attack, and match merging being based on a heuristic. However, this language-independent heuristic is what makes match merging a valuable defense mechanism as it is not bound to a specific attack and can mitigate plagiarism in many forms. Token string normalization cannot defend against other obfuscation attacks and its normalization rules for independent statements are a weak point that could be exploited by an attacker.

Token sequence normalization is a valuable contribution to plagiarism detection and should be utilized in conjunction with match merging to maximize resilience against obfuscation.

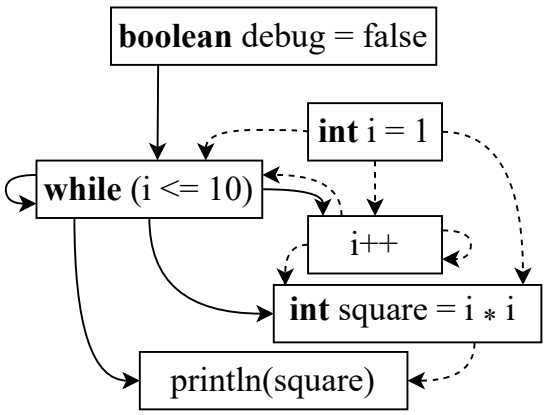


Figure 6.1: Example program dependency graph [4]

6.1.3 Compiler optimization

Devore-McDonald and Berger [9] proposed compiler optimization as a countermeasure against Mossad. The idea is simple. While obfuscating Mossad uses compiler optimization level 3 to ensure that insertion leaves the output unchanged. This aggressive optimization level removes dead code and unused variables. From the optimized compiled we can verify that each obfuscation step has no effect on the semantics.

Let $o(s_i)$ be the optimization from the compiler on the submission with insertion i, where $i = 0$ denotes the original then the following holds:

$$\forall i \in [0, n-1]\ o(s_i) = o(s_{i+1}) \Rightarrow o(s_0) = o(s_n)$$

We can compare the original submission with the final plagiarized one and due to every step being semantic preserving, both cleaned versions are identical.

If a plagiarism detector would compare submissions based on assembly object files generated with optimization level 3 then the impact of Mossad could be completely levered out. Unfortunately, state-of-the-art plagiarism detectors operate based on token sequences generated from the source code, which means they cannot easily utilize this countermeasure.

6.1.4 Semantics-based similarity comparison

In their paper Luo et al. [19] propose a novel code similarity scoring framework *CoP* that aims to be resilient against obfuscation. *CoP* was made for corporate use, aiding firms in proving software plagiarism and algorithm theft. The scoring pipeline of CoP has two distinct parts:

Semantically equivalent basic blocks In computer science, a basic block denotes a code snippet that has one entry point and one exit point. That means regardless of inputs the snippet will reach the same exit point. *CoP* wants to prove that two basic blocks, one from the original and one from the suspect, are semantically identical. To this end, they look at the binary of the submissions and assign both basic blocks specific registers as input. *CoP* now builds a logic formula for each operation in the blocks and through a theorem prover verifies that the output of both blacks points to the same registers and contains the same values. This method of comparing semantics is completely indifferent to dead code insertion and splitting up of operations.

Longest common path Building a ratio of identical basic blocks is not sufficient for an obfuscation-resilient similarity score, as potential thieves could add new paths to their program that either don't contribute to the result or are copied functions. To levitate this issue the authors base their score on the longest sequence of common blocks. While the original program consists of one linear path the suspect could have many. Through fuzzy matching *CoP* finds the longest path of semantically equivalent blocks in both submissions. The final score is the ratio of the length of the longest path to the length of the original.

With *CoP* Luo et al. build a framework that is resilient against single insertion, splitting up of computations and bloating the program with new unnecessary computation paths. Unfortunately, it cannot be easily applied to academic plagiarism detection because we don't know which submission is the original and which is plagiarism when comparing student submissions and the score of *CoP* is based on the length of the original.

6.1.5 Common code deletion

In his PhD thesis *Effect of source-code preprocessing techniques on plagiarism detection accuracy in student programming assignments* [30] Novak, among other techniques, proposed the idea of common code deletion. The idea is to remove common code directly from submissions before they are fed into the plagiarism detection pipeline. Novak considered the following types as common:

- package and import statements
- annotations
- setter methods
- simple 'setter' methods
- getter methods
- empty constructors
- simple constructors
- empty functions
- empty classes
- non-initialized class fields
- empty blocks
- leftover white spaces

These elements offer no meaningful distinction criteria for plagiarism detection and can even be used as insertions to obfuscate, fitting the semantic-preserving threat model. Removing them before the submissions enter the processing pipeline inherently makes plagiarism detectors more resilient but this method is bound to a specific set of common code. Any adversary could use insertions outside of this spectrum to circumvent this defence mechanism.

6.1.6 Various preprocessing techniques

In the survey *Preprocessing for Source Code Similarity Detection in Introductory Programming* [15] Karnalim et al. gathered commonly use preprocessing techniques for plagiarism detection. The three most interesting for the semantic-preserving threat model are:

- Removal of template code
- Removal of single-occurrence token
- Multiple variable declaration expansion

All three types are commonly used by obfuscation tools like Mossad, as these insertions alter the code, leave the semantics unaffected, and require little to no consideration. Removing them directly increases the resilience but just like *Common Code Deletion*, an adversary can alter their insertion strategies to go around the filtering.

6.2 Bioinformatics

The term *Indel* is a combination of Insertion and Deletion and refers to the variation of genome sequences for researching human diseases. Indel detection stands as a vital focus within the realm of bioinformatics, aiming to identify pairs of genome sequences wherein one party has been altered due to indels.

Fundamentally this purpose does not differ from plagiarism detection which also deals with finding pairs of submissions based on the token sequence regardless of small changes. Hence the two approaches presented in the next section are applicable to plagiarism detection as well.

6.2.1 Indel detection

Yang, Van Etten, and Dehm [41] proposed *TransIndel* a novel indel detection framework. It compares two genome sequences and tries to align them while being invariant to insertion or deletions.

Their algorithm operates as follows: It takes two input sequences, denoted as *read* and *target* and applies a specified window size to both. The algorithm then iterates through these sequences, comparing specific segments referred to as chimeric reads.

There a now two cases that can occur and both are visually displayed in Figure 6.2. If there are two possible alignments due to the target sequence containing an extra chimaeric read then is marked as deleted from the read sequence. If there is no possible alignment due to the read sequence containing an extra chimaeric read then is marked as inserted into the read sequence.

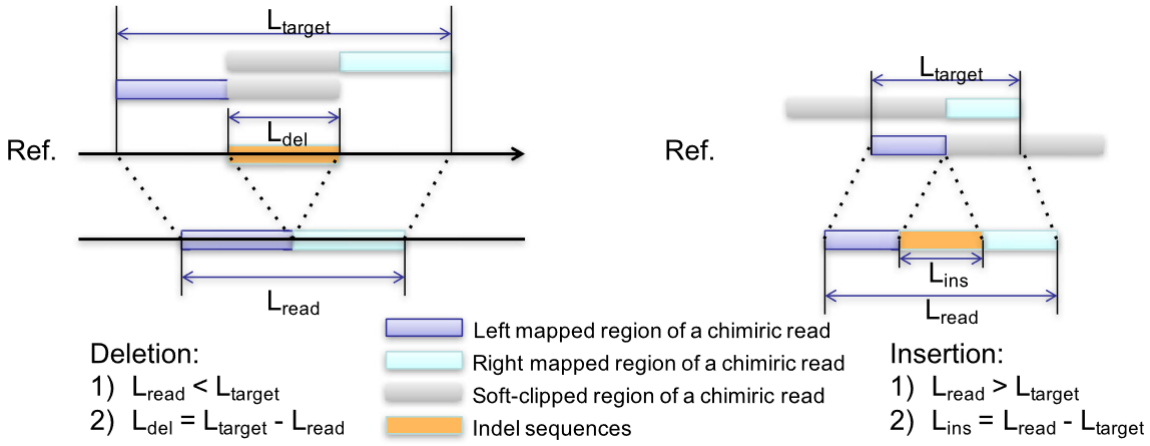


Figure 6.2: Overview of TransIndel [41]

The authors demonstrated that their approach was better than eight state-of-the-art indel detectors. Interestingly, the idea of inserting and deleting parts of sequences to align them has a strong resemblance with the core idea of match merging and backs up the soundness of our approach.

6.2.2 Needleman–Wunsch algorithm

The Needleman–Wunsch algorithm [26] is a dynamic programming algorithm that receives two strings as input and computes the globally best alignment between both sequences. In bioinformatics, it is used to align genome sequences and compute a similarity score.

The algorithm works as follows: We define a scoring system for comparing both sequences. For example, a match between both sequences yields 1 point, a mismatch between both -1 point and a gap -2 points. Gaps in this context stem from indels on the sequences. We now create a matrix and assign the columns with one sequence on the rows with the other. Both sequences can vary in length. Finally, we fill the first column and row with gap penalty points.

The matrix then gets filled out step by step adding and subtracting points depending on matches, mismatches and gaps. For matches and mismatches the value from the previous diagonal cell is taken as the old value. For a gap, the left or up value is taken depending on which submission has the gap. The algorithm ends when the matrix is completely filled and the similarity score for the defined system is in the lowest rightmost cell. Via backtracking and picking the path with the least gaps, we find the best alignment. Figure 6.3 shows an example execution on two genome sequences with the previously defined scoring system.

The Needleman-Wunsch algorithm is easily applicable to plagiarism detection when using two token sequences as input. A suitable scoring system would then yield pairwise similarity, where higher scores refer to more similar submissions.

		H	E	A	H	E	E
	0	-2	-4	-6	-8	-10	-12
P	-2	-2	-4	-6	-8	-10	-12
A	-4	-4	-4	-3	-5	-7	-9
H	-6	-3	-5	-5	-2	-4	-6
E	-8	-5	-2	-4	-4	-1	-3

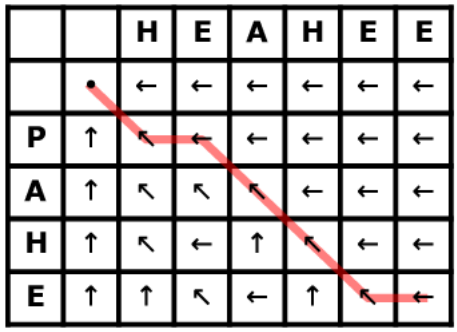


Figure 6.3: Example of Needleman-Wunsch algorithm on two genome sequences [27]

7 Future work

We conceptualized five starting points for future work that could enhance our contributions:

Experiment on more test sets

We conducted a full-grid search of 17 test sets to find a combination of neighbor length and gap size that overall yielded the best results, to serve as default values for JPlag. Experiments on more test sets would result in a combination that is more generally applicable. To this end, one can either use more datasets with the attacks presented in this thesis or come up with novel attacks and test match merging against the new test sets.

Use GPT4

Near the end of this thesis, OpenAI published a novel version of GPT. It remains to be seen if GPT4 outperforms GPT3.5 in aiding obfuscation or fully-generating submissions. There is currently no way of accessing GPT4 without paying a monthly fee for the ChatGPT variant or API access which is billed by the amount of processed words. Regardless, experimenting with GPT4 would yield valuable insights and if proven substantially more powerful than GPT3.5 could make a completely new approach to plagiarism detection necessary.

Extend merging heuristic

There is currently no intermediate token representation that is language-independent. This makes a heuristic based on token types inefficient as one would have to create rules for every language supported by the plagiarism detector.

If they were a language-independent representation we could exploit the fact that semantic-preserving obfuscation tools like Mossad or JPlag-GEN use multiple one-liners as insertions that are independent from one another, which restricts these tools to variable definitions and applications of functions.

A rule like could check if a gap only contains *VAR_DEF* and *APPLY* tokens:

$$\forall token \in gap : isType(token, APPLY) \vee isType(token, VAR\_DEF)$$

Token-type rules could also be combined with hyperparameters to increase the capacity of our heuristic.

Real Alteration attack

One could build a *real* Alteration attack that changes the source code while leaving the output unchanged. To this end, one could switch different loop types, replace if-statements with switch cases, extract methods or apply currying. We came up with three different approaches that could aid a potential researcher:

1. Use refactoring APIs from IDEs like *Eclipse* or *IntelliJ* programmatically.
2. Use a combination of regular expressions and bracket counting to search for replaceable content and know its scope.
3. Make changes to the Abstract-Syntax-Tree and then translate it back to the source code.

This proposed Alteration attack falls into the semantic-preserving category.

Extract base game from TicTacToe

We could not compare the fully-generated submissions for Tic-Tac-Toe with the original ones from students because the LLM was only capable of generating the base game task and all student submissions include the complete task, consisting of the base game and simple AI-controlled opponent.

One could manually remove the AI extension from the original submissions and then run a more thorough comparison of the similarity scores. Manually altering datasets has to be done with great care so as not to invalidate the results due to subjective decisions and personal preferences.

8 Conclusion

As the amount of computer science students steadily rises educational personnel have to rely on state-of-the-art source code plagiarism detection tools, to prevent students from submitting plagiarized coding assignments. These token-based plagiarism detectors are inherently resilient against obfuscations such as lexical changes and retyping. The recently disclosed obfuscation tools Mossad and JPlag-GEN pose a threat to academic integrity as they empower students to effortlessly change their submissions to deceive the plagiarism detectors. These tools automatically use insertions of dead code and reordering of independent statements to lower the similarity between original and plagiarism. Additionally, the recent rise of ChatGPT also raises concerns about its obfuscation capabilities and whether they can be mitigated. While defense mechanisms against obfuscation exist they are often only effective on one attack or are language-dependent, requiring tedious and error-prone reimplementation.

We developed a novel defence mechanism against automatic obfuscation attacks called match merging. It leverages the fact that all obfuscation attacks change the token sequence by inserting tokens, reordering tokens or changing their type. These changes split up continuous matches between two submissions and if the broken matches fall below the minimum length threshold the plagiarism detector discards them. Our approach reverts the effects of these attacks by merging neighboring matches based on a heuristic. The heuristic keeps false positives low and mainly operates based on the length of neighbors and the size of the gap between them. Our approach works on matches between token sequences and does not consider the specific tokens. This makes it inherently language-independent and completely agnostic to semantics, making it attack-independent.

To show the effectiveness of our approach, we evaluated six different attack types on three distinct datasets. In addition Mossad and JPlag-GEN, we considered attacks that alter and swap tokens and utilized large language models for obfuscating and fully generating submissions. The datasets contain real submissions from undergrad coding courses and competitive coding platforms and vary in length and language.

Against Mossad-generated plagiarism from the Ai-Soco dataset, we were able to raise the similarities of plagiarized submissions from 20 to 80 percent while leaving original submissions mostly unaffected. This clearly separates plagiarized from original, allowing for easy identification. As Mossad relies on a plagiarism detector to check similarities after every insertion we additionally evaluated changes to the amount of insertions and submission size, when the attack is hindered by our approach. Because match merging can compensate for changes up to a degree, Mossad needs many more insertions to reach its target which increases the size of submissions by an average of 250 percent. With such sizes, submissions are bound to raise suspicion even with low similarity scores.

We generated plagiarism with JPlag-GEN from two tasks from Progpedia and from one assignment from our university. For the first task, we raised the plagiarism similarities from 10 to 40 percent without affecting the false positive rate. The effectiveness of the second task was

even better raising plagiarized submissions from 15 to 60 percent. The last assignment consists of longer submissions where we raised the similarities of plagiarized submissions from 5 to 20 percent.

Alteration and Swapping can be tweaked to set the percentage of changed tokens and both have a similar impact on the scores. For 10 percent changes in the submissions, we raised the similarities of plagiarized submissions from 50 to 90 percent. For 20 percent changes we notice a higher increase from 20 to 80 percent. Lastly, for 30 percent changes submissions we get an increase from 0 to 65 percent. We conclude that our approach is resilient against semantic-preserving and semantic-agnostic attacks while leaving false positives mostly unaffected.

We conducted experiments on large language model evaluating their ability to obfuscate submissions based on prompts and generate complete submissions directly from the task description. For obfuscation, we created 15 different prompts varying in effort and knowledge about plagiarism detectors, 11 of which failed to deceive the detector. The remaining 4 prompts lowered the similarities to around 55 percent which our defense mechanism increased to over 65 percent. For fully generating submissions the LLM created submissions from a simple coding task and we measured the similarity between them. With match merging, we were able to increase similarities from 46 to 68 percent with no score falling below 40 percent. Thus we significantly improved the plagiarism detection capabilities against AI-based plagiarism.

Further experiments showed that the impact of match merging on the runtime of the plagiarism detector is negligible, only rising slowly and linearly the more aggressively it is used. This is important as educational personnel use plagiarism detection on up to a thousand submissions at once and a quadratic increase in runtime would drastically lower the usability of our approach.

In summary, this thesis shows that match merging effectively defends against a broad spectrum of obfuscating attacks. What sets our approach apart is its language and attack independence, with an underlying heuristic that keeps the scores of valid submissions mostly unaffected. Evaluations across various attacks and datasets demonstrate the general effectiveness of our method. Importantly, our approach introduces minimal runtime overhead, making it seamlessly compatible with other defence mechanisms.

Bibliography

[1] Mayank Agrawal and Dilip Kumar Sharma. "A state of art on source code plagiarism detection". In: *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)*. IEEE. 2016, pp. 236–241.

[2] Stella Biderman and Edward Raff. "Fooling MOSS Detection with Pretrained Language Models". In: New York, NY, USA: Association for Computing Machinery, 2022. ISBN: 9781450392365. DOI: 10.1145/3511808.3557079. URL: https://doi.org/10.1145/3511808.3557079.

[3] Romain Brixtel et al. "Language-Independent Clone Detection Applied to Plagiarism Detection". In: *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. 2010, pp. 77–86. DOI: 10.1109/SCAM.2010.19.

[4] Moritz Brödel. "Preventing Automatic Code Plagiarism Generation Through Token String Normalization". BA thesis. Karlsruhe Institute of Technology, 2023.

[5] *ChatGPT*. https://chat.openai.com/. Accessed: 2023-10-05.

[6] Xin Chen et al. "Shared information and program plagiarism detection". In: *IEEE Transactions on Information Theory* 50.7 (2004), pp. 1545–1551. DOI: 10.1109/TIT.2004.830793.

[7] *Codeforces*. https://codeforces.com/. Accessed: 2023-05-14.

[8] Peter A. Cotton Debby R. E. Cotton and J. Reuben Shipway. "Chatting and cheating: Ensuring academic integrity in the era of ChatGPT". In: *Innovations in Education and Teaching International* (2023). DOI: 10.1080/14703297.2023.2190148. URL: https://doi.org/10.1080/14703297.2023.2190148.

[9] Breanna Devore-McDonald and Emery D. Berger. "Mossad: Defeating Software Plagiarism Detection". In: *CoRR* abs/2010.01700 (2020). arXiv: 2010.01700. URL: https://arxiv.org/abs/2010.01700.

[10] Ali Fadel et al. *Authorship Identification of SOurce COde 2020 (AI- SOCO)*. Zenodo, Sept. 2020. DOI: 10.5281/zenodo.4059840. URL: https://doi.org/10.5281/zenodo.4059840.

[11] M. Joy and M. Luck. "Plagiarism in programming assignments". In: *IEEE Transactions on Education* 42.2 (1999), pp. 129–133. DOI: 10.1109/13.762946.

[12] M. Joy and M. Luck. "Plagiarism in programming assignments". In: *IEEE Transactions on Education* 42.2 (1999), pp. 129–133. DOI: 10.1109/13.762946.

[13] *JPlag repository*. https://github.com/jplag/JPlag. Accessed: 2023-05-14.

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

[15] Oscar Karnalim, Simon, and William Chivers. “Preprocessing for Source Code Similarity Detection in Introductory Programming”. In: New York, NY, USA: Association for Computing Machinery, 2020. ISBN: 9781450389211. URL: https://doi.org/10.1145/3428029.3428065.

[16] Pascal Krieg. “Preventing Code Insertion Attacks on Token-Based Software Plagiarism Detectors”. Abschlussarbeit - Bachelor. Karlsruher Institut für Technologie (KIT), 2022. DOI: 10.5445/IR/1000154301.

[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*. 2006, pp. 872–881.

[18] Brady D. Lund et al. “ChatGPT and a new academic reality: Artificial Intelligence-written research papers and the ethics of the large language models in scholarly publishing”. In: *Journal of the Association for Information Science and Technology* 74.5 (2023), pp. 570–581. DOI: https://doi.org/10.1002/asi.24750. URL: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/asi.24750.

[19] Lannan Luo et al. “Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection”. In: *IEEE Transactions on Software Engineering* 43.12 (2017), pp. 1157–1177.

[20] Rien Maertens et al. “Dolos: Language-agnostic plagiarism detection in source code”. In: *Journal of Computer Assisted Learning* 38.4 (2022), pp. 1046–1061.

[21] Vitor T Martins et al. “Plagiarism detection: A tool survey and comparison”. In: (2014).

[22] Jesse G Meyer et al. “ChatGPT and large language models in academia: opportunities and challenges”. In: *BioData Mining* 16.1 (2023), p. 20.

[23] *Moss*. https://theory.stanford.edu/~aiken/moss/. Accessed: 2023-05-14.

[24] William Hugh Murray. “Cheating in Computer Science”. In: 2010.October (2010). DOI: 10.1145/1865907.1865908. URL: https://doi.org/10.1145/1865907.1865908.

[25] Engineering National Academies of Sciences, Medicine, et al. *Assessing and responding to the growth of computer science undergraduate enrollments*. National Academies Press, 2018.

[26] Saul B. Needleman and Christian D. Wunsch. “A general method applicable to the search for similarities in the amino acid sequence of two proteins”. In: *Journal of Molecular Biology* 48.3 (1970), pp. 443–453. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(70)90057-4. URL: https://www.sciencedirect.com/science/article/pii/0022283670900574.

[27] An Nguyen et al. “Conformance Checking for a Medical Training Process Using Petri net Simulation and Sequence Alignment”. In: (Oct. 2020).

[28] Lawton Nichols et al. “Syntax-Based Improvements to Plagiarism Detectors and Their Evaluations”. In: New York, NY, USA: Association for Computing Machinery, 2019. ISBN: 9781450368957. DOI: 10.1145/3304221.3319789. URL: https://doi.org/10.1145/3304221.3319789.

[29] Niehues. *Reproduction package for: Intelligent Match Merging to Prevent Obfuscation Attacks on Software Plagiarism Detectors*. Zenodo, Nov. 2023. DOI: 10.5281/zenodo.10149536. URL: https://doi.org/10.5281/zenodo.10149536.

[30] Matija Novak. "Effect of source-code preprocessing techniques on plagiarism detection accuracy in student programming assignments". PhD thesis. University of Zagreb. Faculty of Organization and Informatics, 2020.

[31] Matija Novak, Mike Joy, and Dragutin Kermek. "Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review". In: 19.3 (2019). DOI: 10.1145/3313290. URL: https://doi.org/10.1145/3313290.

[32] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. *PROGpedia*. Zenodo, Dec. 2022. DOI: 10.5281/zenodo.7449056. URL: https://doi.org/10.5281/zenodo.7449056.

[33] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. *JPlag: finding plagiarisms among a set of programs*. Tech. rep. 1. 2000. DOI: 10.5445/IR/542000.

[34] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical software engineering* 14 (2009), pp. 131–164.

[35] Dylan Ryman, PK Imbrie, and Jeff Kastner. "Application of source code plagiarism detection and grouping techniques for short programs". In: *2021 IEEE Frontiers in Education Conference (FIE)*. IEEE. 2021, pp. 1–7.

[36] Timur Sağlam et al. "Token-Based Plagiarism Detection for Metamodels". In: New York, NY, USA: Association for Computing Machinery, 2022. ISBN: 9781450394673. DOI: 10.1145/3550356.3556508. URL: https://doi.org/10.1145/3550356.3556508.

[37] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. "Winnowing: local algorithms for document fingerprinting". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 2003, pp. 76–85.

[38] Rini van Solingen (Revision) et al. "Goal Question Metric (GQM) Approach". In: *Encyclopedia of Software Engineering*. 2002. DOI: https://doi.org/10.1002/0471028959.sof142.

[39] Guo Tao et al. "Improved Plagiarism Detection Algorithm Based on Abstract Syntax Tree". In: *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*. 2013, pp. 714–719. DOI: 10.1109/EIDWT.2013.129.

[40] Michael J Wise. "String similarity via greedy string tiling and running Karp-Rabin matching". In: *Online Preprint, Dec* 119.1 (1993), pp. 1–17.

[41] Rendong Yang, Jamie L Van Etten, and Scott M Dehm. "Indel detection from DNA and RNA sequencing data with transIndel". In: *BMC genomics* 19.1 (2018), pp. 1–11.

[42] Mengya Zheng, Xingyu Pan, and David Lillis. "CodEX: Source Code Plagiarism Detection Based on Abstract Syntax Tree." In: *AICS*. 2018, pp. 362–373.