Bachelor Informatica

University of Amsterdam

CRDS: Detecting Code Plagiarism Reordering Attacks using Abstract Syntax Trees

Philo Decroos

June 18, 2021

Supervisor(s): J. (Jelle) van Assema MSc

Abstract

With the increase in popularity of programming courses comes the prevalent problem of plagiarism. A selection of tools for automated detection of code plagiarism is widely used (MOSS [1] and JPlag [2] are examples). These tools efficiently detect most plagiarism in academic programming. However, interpretation and explanation of the results of these tools is still a difficult and time consuming task. Research into leveraging code structure information for plagiarism detection using Abstract Syntax Trees or Program Dependence Graphs has provided improvements over established tools [3, 4, 5, 6, 7]. Yet, this research has not been focused on the interpretation aspect of automated plagiarism detection. In this thesis, we present the Code Reordering Detection System (CRDS): a plagiarism detection tool that can effectively find many code reordering attacks in plagiarised code using an algorithm based on Abstract Syntax Trees. The algorithm is language independent and languages can easily be added to CRDS using grammar files. We find that CRDS effectively detects code reordering on every scale. The results produced by CRDS can be used as an extra layer on top of existing plagiarism detection tools, to provide insights in reordering modifications that were made by the plagiarising student. With this addition, we aim to make interpretation of the results of existing tools easier for teachers. We hope that the results presented in this thesis will encourage future research into similar and more sophisticated detection techniques.

Contents

1 Introduction **7**

2 Theoretical framework **9**
2.1 Characteristics of Code Plagiarism 9
2.1.1 Student motivation 9
2.1.2 Plagiarism attacks 10
2.1.3 Judging plagiarism 10
2.2 Prior Research 10
2.2.1 String-matching-based algorithms 11
2.2.2 Tree- and graph-matching-based algorithms 11
2.2.3 User interfaces 13

3 Methodology **15**
3.1 Design 15
3.1.1 Hashing Algorithm 15
3.1.2 Sub tree selection 15
3.1.3 Similarity detection algorithm 16
3.1.4 Reordering detection algorithm 16
3.2 Implementation 18
3.2.1 Hardware 18
3.2.2 Used tools 18
3.2.3 Framework 19
3.2.4 Pre-procesing step 19
3.2.5 Comparison step 20
3.2.6 Results 20
3.3 Data sets 20
3.3.1 Real-world data set 20
3.3.2 Synthetic data set 20
3.4 Generation of reorderings 21
3.4.1 Sub-statement 21
3.4.2 Statements 22
3.4.3 Functions 22
3.4.4 Conditionals 22
3.5 Experiments 22
3.5.1 Comparison to MOSS 22
3.5.2 Reordering detection 23
3.6 Evaluation 23

4 Results **25**
4.1 Comparison to MOSS 25
4.1.1 Real-world data 25
4.1.2 Influence of reordering 26
4.2 Reordering Detection 27

4.2.1 Synthetic data . . . 27
4.2.2 Real-world data . . . 28

5 Discussion **31**
5.1 AST-based similarity detection . . . 31
5.2 Existing vulnerabilities . . . 32
5.3 Reordering detection . . . 32
5.4 Weaknesses . . . 32
5.5 Ethical Aspects . . . 33
5.6 Future Research . . . 33

6 Conclusion **35**

Appendix A Full results of MOSS comparison **39**

Appendix B Full results of detection experiments **41**

CHAPTER 1

Introduction

Programming classes have become increasingly popular in recent years. With the increase of computer science students at universities [8], and popular online courses like Harvard's CS50 [9], large classes of sometimes thousands of students have become common. With education containing student assignments comes the problem of plagiarism. Programming courses are especially vulnerable to student plagiarism compared to courses with essay assignments [10].

To preserve academic integrity, detection of code plagiarism is important to teachers [11]. This process is time consuming when done by hand, and requires domain specific knowledge on the taught subjects and the assignments. Additionally, students will disguise plagiarism to prevent detection of plagiarism. Multiple attacks are known to be used to make code similarity difficult to detect. These attacks range from trivial modifications like changing identifier names to complex structural rearrangements of code [12]. Many of these attacks can be executed without understanding the source code [13].

To aid teachers in detecting plagiarism, automatic code plagiarism detection tools have been developed. Examples of tools that are widely used are MOSS [1] and JPlag [2]. These tools use fingerprinting or string matching algorithms that operate on token streams, and have been widely used for years. These token-based algorithms are robust against basic disguise attacks like changing comments, white space and identifier names. However, they are sensitive to some attacks, like adding useless statements or reordering of code [14].

Improvements on token-based algorithms have been proposed, and proven to be effective in improving similarity detection when more advanced attacks are executed. Newly developed techniques often use more structural information of source code [15], using a parser to create Abstract Syntax Trees (AST). Comparing the structural information represented by these more complex data structures removes some vulnerability to attacks that exists in the string-matching algorithms.

However, these improvements did not lead to new, more advanced plagiarism detection tools that are widely used. MOSS and JPlag are still the most popular choices in universities. A reason for this lack of practical usage could be that AST-based algorithms often have issues with performance: tree traversal is an operation that has high complexity, and comparing thousands of submissions with these algorithms can lead to very long execution times. A different reason could be that the improvements that these algorithms introduce are not significant enough for teachers to replace the tools they currently use. While AST-based tools improve similarity scores, the fact that they are not adopted by many teachers indicates that the effectiveness of the older tools might be sufficient.

Since interest among teachers towards techniques improving similarity detection is low, in this thesis, we employ Abstract Syntax Trees to address a different problem with existing tools. This problem lies in interpreting the results from plagiarism detection tools. The user interface of these tools is often quite limited. In most cases, the user interface consists of an "inverse code diff" only: code that is similar to code in a different file is highlighted. Interpretation of these similarity results is left to the teacher or assistant. When an attack like code reordering is executed, interpretation becomes time consuming. Because existing tools do not provide insights

in executed attacks, identifying them requires careful examination of the code. Arguing about a suspicion of plagiarism towards students or exam committees therefore requires experience and knowledge about the assignment.

We have used the structural information extracted from code using ASTs to provide additional explanations to the user about why code is similar. Instead of enhancing plagiarism detection to be insensitive to attacks, this thesis focuses on detecting reordering of source code. This specific attack was chosen because it occurs frequently [13], and it makes interpretation of similarity results tedious. By highlighting reorderings in a user interface, similarity can be made more obvious and explainable. Many reorderings found in a single submission could also increase suspicion of plagiarism over just code similarity, because it indicates that a student likely executed attacks on existing code. The research question we will answer in this project is:

> **How effectively can code reordering be detected using Abstract Syntax Trees?**

We will answer this question by presenting the Code Reordering Detection System (CRDS): a plagiarism detection tool that uses a new algorithm based on Abstract Syntax Trees to detect reordering of source code. In order to give a conclusive answer, five sub questions will be answered by experimenting on CRDS. Because the detection algorithm is based on existing AST-based similarity detection, we will first confirm that this detection technique is effective:

- *How does similarity detection based on Abstract Syntax Trees compare to the established tool MOSS?*

Next, we will examine the detection of reordering on four different scales. Reordering attacks occur in real submissions on every one of these scales [12]. This approach results in the following sub questions:

- *How effectively can CRDS detect sub-statement reorderings?*
- *How effectively can CRDS detect statement reorderings?*
- *How effectively can CRDS detect function reorderings?*
- *How effectively can CRDS detect reordering within conditional statements?*

We find that, in accordance with conclusions from previous research, AST-based similarity detection can compete with MOSS in plagiarism detection on real submissions. The existing algorithm performs significantly better than MOSS when reordering attacks are executed. The reordering detection algorithm introduced by CRDS can find reorderings on every scale. These reorderings are, however, only detected when the reordered code is not otherwise edited structurally. Because of this, reorderings between large code blocks will not be detected often in real submissions.

CHAPTER 2

Theoretical framework

2.1 Characteristics of Code Plagiarism

Cosma and Joy [11] define source-code plagiarism as follows.

> "Source-code plagiarism in programming assignments can occur when a student reuses source-code authored by someone else and, intentionally or unintentionally, fails to acknowledge it adequately, thus submitting it as his/her own work."

This definition includes the reuse of code of others and modifying this code, depending on the effort that is required for the modifications. Clearly, failing to acknowledge source-code reuse is stated as a condition to label reuse as plagiarism. When reuse is not allowed, but is acknowledged appropriately, reuse may be regarded as an incorrect solution to the assignment rather than plagiarism.

Details of the definition of plagiarism lie within the instructions of the assignment, and regulations stated by universities. Some assignments allow collaboration between students while others do not, which can make the definition vary among academics and students. When collaboration or reuse is permitted, general consensus is that proper acknowledgement of ownership is still needed to avoid committing plagiarism offences [11].

Not only reuse of source-code, but also reuse of design ideas can sometimes be regarded as plagiarism. However, in an academic context, students are often solving the same problem where a specific solution is expected. Because of this, most research into academic plagiarism focuses on direct source-code plagiarism [16].

2.1.1 Student motivation

Research into the perspective of the student has shown that many students perceive a programming assignment to be similar to a maths problem. They do not realise that there are many solutions to the same problem, and that plagiarism can be detected even if their programs generate the correct output [17]. In a survey among 313 Computer Science students executed by Chuda et al. [16], 33% of students admitted to having committed plagiarism at their faculty. Also, only 10% of students said to do all their assignments completely without looking at the work of others. The availability of digital technology has made code reuse even more tempting to students [18]. Collaboration is of course not always a cause of plagiarism, because collaborating and looking at code of others for inspiration about how to tackle an assignment is quite a common practice in programming in general. However, students may cross the line into direct copying of code of others for several direct reasons [16]:

- time pressure;
- disinterest in the course subject;
- tolerance for, or little interest in plagiarism by teachers;

- too challenging assignments.

Furthermore, plagiarism in programming courses is sometimes viewed among students as a minor offence compared to plagiarising essay assignments [17]. Both Chuda et al. [16] and Aasheim et al. [17] concluded that while plagiarism is often caused by student motivation, teachers play a large role in preventing plagiarism. Clearly specifying what is acceptable and what is not, making sure students are reminded of the rules and that their submissions will be checked, and not overloading the students with too difficult assignments might prevent many cases of plagiarism.

2.1.2 Plagiarism attacks

When students do decide to plagiarise code of others, they are likely to change the code to disguise their plagiarism. Previous research into detecting plagiarism has empirically listed many different attacks that are commonly executed by students [13, 12]. Some examples are:

- changing comments and identifier names;
- reordering statements or functions;
- adding useless code to the assignment;
- splitting code up into functions;
- replacing statements with equivalents that use different syntax.

Different techniques used for automatic plagiarism detection are vulnerable to different attacks. Research towards improving existing tools is often focused on making these tools robust against these attacks, so that students can not fool the detection algorithm.

For our plagiarism attack of focus, the reordering of code, Karnalim [12] stated four different specific methods of execution:

- rearranging method declarations;
- rearranging branching statements based on its condition validation sequence;
- rearranging loosely-coupled instructions;
- changing operand order in complex arithmetic operations.

2.1.3 Judging plagiarism

Teachers generally agree that a "zero tolerance policy" towards plagiarism is appropriate [11]. The reasoning for this, is that plagiarism is thought to be detrimental to students' education, because it removes the need for understanding the concepts that are taught through coding assignments. However, detecting plagiarism and avoiding false positives remains a challenging task for teachers, even with the help of detection tools. When similar code is found, the distinction between plagiarism and other reasons for similarity still needs to be made. Collaborating students, an easy to find source on the internet, or a specific exercise with little variation in possible solutions can all lead to similar code. It is up to a teacher to decide when similar code is suspicious, and most tools do not provide many clues to help with this.

2.2 Prior Research

Code plagiarism detection techniques can be divided into several categories [15]. Our focus is on structure-based techniques, where algorithms operate on underlying code structure, as opposed to direct text comparison or attribute-counting-based approaches. Within structure-based techniques a distinction can be made between string-matching, tree-matching and graph-matching algorithms. This distinction is mainly based on the representation of the code that is used to run the comparison algorithm on. Code is transformed before the comparison: either into a token stream using a lexer, or into an Abstract Syntax Tree or Program Dependence Graph using a parser.

2.2.1 String-matching-based algorithms

Winnowing

The winnowing [19] algorithm is a document comparison algorithm based on *n-gram* fingerprinting. While this algorithm was originally designed for document comparison, many desired properties of a text comparison algorithm are similar to those of code plagiarism detection. Documents are split into n-grams, and these n-grams are hashed and cross-compared between two files. This fingerprinting approach makes the comparison position independent. Winnowing uses a *noise threshold*, a variable minimum size of a sub string match, to ensure that small, coincidental matches are ignored.

MOSS [1] was first developed in 1994, and is one of the oldest plagiarism detection tools that is still widely used. By running source-code through a lexer and executing the winnowing algorithm on the produced tokens, MOSS applied winnowing in a way that works well for source code plagiarism detection [20].

More recently, a collaboration between Harvard University and the University of Amsterdam has resulted in Compare50 [21], an open source, extensible tool that also uses the winnowing algorithm on token streams.

Greedy String Tiling

Another very popular tool, JPlag [2], developed in 2000, uses a similar string-matching algorithm. The Greedy String-Tiling (GST) algorithm [22] it uses is similar to a Longest Common Subsequence (LCS) algorithm: the algorithm looks for the largest common sub strings in two token strings. Unlike LCS, Greedy String-Tiling is not dependent on sub string location within the compared strings. This makes plagiarism detection with GST also position independent.

Evaluation

Evaluation of these tools shows that they perform quite well in detecting plagiarism [23, 20]. Since these algorithms operate on token streams, they are robust against the most basic and common attacks [24], including:

- changes in white space;
- adding, removing or changing comments;
- changing identifier names.

Because winnowing and GST are position independent, partial plagiarism, embedded in different code, can also be detected. This makes executing attacks that will fool these techniques already quite a challenging task for the average student. However, while their use in practice shows their effectiveness, string-matching-based algorithms are sensitive to more advanced structure-based attacks. Local reordering of statements, moving code to functions, adding unused code, and replacing statements with semantic equivalents all reduce the effectiveness of string-matching-based algorithms [14, 24].

2.2.2 Tree- and graph-matching-based algorithms

Several improvements have been proposed for algorithms that are less sensitive to these attacks, using more structural information about the code. A way to extract structural data from a program is to use a parser to create parse trees or Abstract Syntax Trees (AST). An even more explicit and detailed representation of code is a Program Dependence Graph. This representation uses a graph data structure to explicitly represent the full control flow and data flow of a program. While these data structures make detection more robust, this improvement comes at the cost of performance: comparisons of trees and graphs are a very costly operation. Most of these tools therefore employ several performance improvements to make the execution times acceptable. Still, execution times are significantly larger than those of string-matching-based algorithms.

Abstract Syntax Trees

Abstract Syntax Trees are parse trees that are stripped from language specific syntax information. This information is not relevant for plagiarism detection, because we are looking at code structure rather than syntax. Figure 2.1 shows the AST generated by running the simple C function shown in Code 2.1 through a parser.

Code 2.1: A simple C function that adds two integers.

```
1    int add(int i, int j) {
2        return i + j;
3    }
```

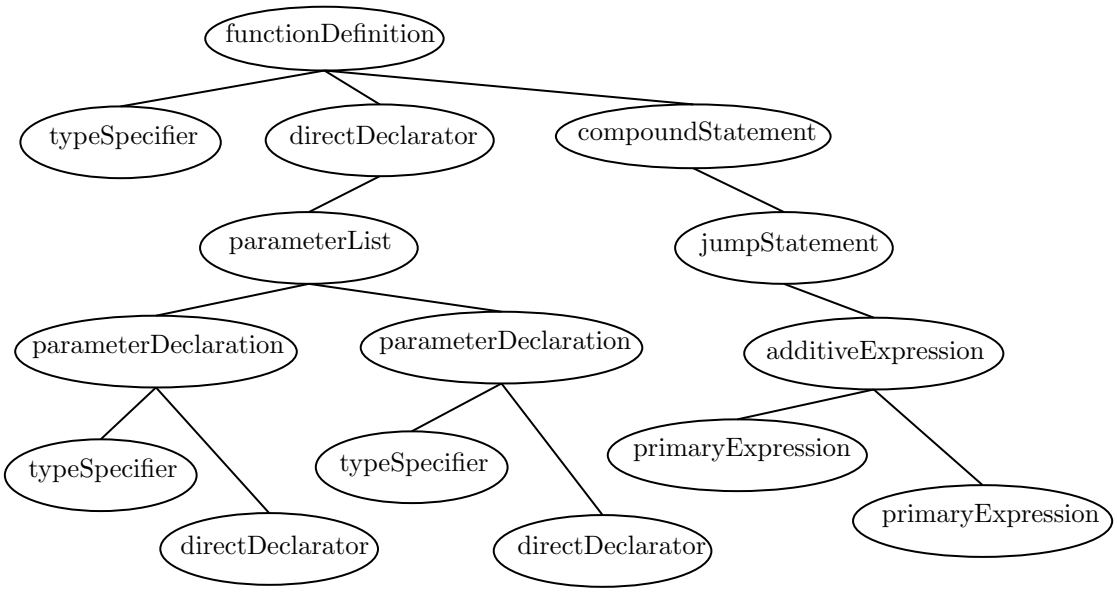

Figure 2.1: An Abstract Syntax Tree generated by a parser from the simple C function shown in Code 2.1. This tree contains structural information about the code, but the syntax is disregarded.

Cui et al. [3] introduced a plagiarism detection tool that uses ASTs. They use a lexer to create a token stream, and a parser generated with *Yacc* [25] to generate parse trees. Then, Abstract Syntax Trees are generated by selecting the relevant nodes from the parse tree. Similarity metrics are then calculated by comparing the syntax trees. To improve performance, the nodes of the AST are hashed, combining the node type and the hash value of all child nodes. This combination of hash values is done using addition: since addition is commutative, the sum of hash values of a node and its children remains the same when reordering the children. Because of this property, the hash value of a node uniquely represents the entire sub tree below this node, and is invariant under reordering within the sub tree. After the hashing step, all sub trees of the AST are classified by their number of child nodes. Hash values of sub trees with equal size are compared, and if they match, the lines of code that make up this sub tree are highlighted as similar in a user interface. This resulted in a tool the authors named Code Comparison System (CCS). This algorithm was later extended to also be robust against logic flow changes in conditional statements [26]. While the results were promising, the CCS tool does not seem to have been made available for use.

Fu et al. [4] took inspiration from Natural Language Processing and applied a kernel method to find similarities between Abstract Syntax Trees. This model has a mathematical nature, directly calculating a similarity score between two trees. This makes it less suited for user interface solutions since it only provides a single score between two files.

Kikuchi et al. [6] used AST representation for pre-processing only: source code submissions are divided into functions. They then cross-compared the token sequences of these functions

with a sequence alignment algorithm. This approach is less costly because the AST only needs one traversal, and the comparison itself is done with a less complex operation.

Nichols et al. [5] use ANTLR to generate syntax trees, and their approach is similar to that of Kikuchi et al. [6], extending the pre-processing further. They also divide the source code into functions. Then, the function sub trees are sorted by size and then traversed, producing linear token sequences. These sequences are then filtered, only keeping specific tokens specified by the user, and transforming all if/else statements into switch statements. These processed sequences are then compared using the Smith-Waterman [27] sequence alignment algorithm.

Program Dependence Graphs

GPLAG [7] is a tool that uses Program Dependence Graphs (PDG) for plagiarism detection. This representation contains more information than an AST: it includes the complete data- and control flow of a program. The authors of GPLAG claim that PDGs are even more robust against every type of plagiarism attacks than ASTs. However, PDGs are very large data structures and a lot of optimisation is needed to use them in practice without high execution times. Since this publication in 2006, most research has been focused on Abstract Syntax Trees.

2.2.3 User interfaces

Figure 2.2 shows two examples of a commonly used type of user interface. Almost every plagiarism detection tool has a user interface that is very similar to the ones shown. They often include a similarity score and an "inverse code diff": the two compared files are shown side by side, and parts of the code that are similar are highlighted.

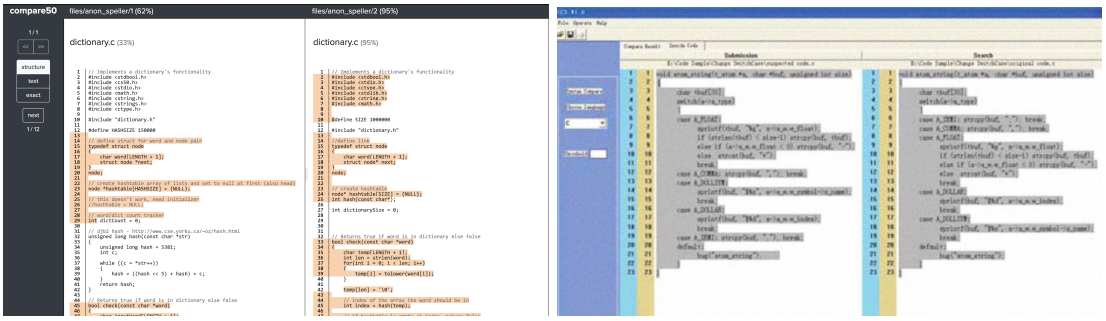

Figure 2.2: Two examples of user interfaces used by two different tools. On the left, the user interface provided by Compare50 [28]; on the right, the user interface that is shown in the article about CCS [3], the detection tool that uses ASTs. The output of both programs is an "inverse code diff", highlighting the code that is similar between two files.

Determining which segments of code are found to be similar by these tools is an easy task when using this type of user interface. To act on a suspicion of code plagiarism however, more than the output of these tools is needed. A teacher needs to be able to argue why it is very likely that code similarity is actually plagiarism. Currently, this task relies entirely on the expertise of the teachers, and is time consuming, especially when the plagiarised code is heavily edited.

Improvements on this type of interface are often on a larger scale: for example, Poon et al. [29] designed a tool that includes an extensive user interface about clustering (within classes and multiple submissions) and trends among multiple submissions of a single student. To the best of our knowledge, advanced techniques have not been applied to extend the standard user interfaces on a small scale, to explain *why* code blocks have been labelled as similar by the tool.

CHAPTER 3

Methodology

To answer our research question, *"How effectively can code reordering be detected using Abstract Syntax Trees?"*, we have created a basic plagiarism detection tool. We have named it Code Reordering Detection System (CRDS). It is meant as a "proof-of-concept" framework to implement our reordering detection algorithm, and to conduct experiments on it. Therefore, not much effort has been focused towards any user experience of the tool. It is able to read source code submissions from a folder, cross-compare them using our algorithm, and show the results in a simple user interface. These results are two-fold: it provides both a standard "inverse code diff" similarity interface, and a separate interface to show reordering of blocks of code.

3.1 Design

3.1.1 Hashing Algorithm

The CCS algorithm introduced by Cui et al. [3] seems promising: the authors already showed that their detection system was robust against some reordering attacks. Because of this, the hashing approach from CCS is used as a starting point. The hashing algorithm that CRDS implements is MD5 [30]. This is an older, 128-bit hashing function. An MD5 hash is represented as a 32-character hexadecimal number: for example, the MD5 hash of the word "hash" is *0800fc577294c34e0b28ad2839435945*. This hashing function is suitable for our use case, because its output size is small, making addition of hashes fast compared to advanced hashing algorithms like SHA-256. Although collisions are easy to find for MD5, our input space is limited to at most a few hundred node types, making the probability of finding collisions negligibly small.

After an Abstract Syntax Tree is generated from a source-code submission, it is traversed bottom-up and every node is hashed using the following procedure:

$$h_n = (\mathbf{hash}(type) + \sum_{c \in C_n} h_c) \bmod 2^{128}$$

where h_n is the computed hash value of node n; **hash** is the MD5 hashing function; *type* is the type of node (for example *functionDefinition*); and C_n is the collection of children of node n. By using addition to compute a node's hash value, this value represents the sub tree below the node, invariant under reordering of its children. By hashing only the *type* of the node, syntactic features of the code like identifier names and literal values are abstracted away. This way, comparison is based on structural information only.

3.1.2 Sub tree selection

Since the hash value of the root node represents the entire sub tree, and we expect these hashes to be unique, the only useful comparison to make is between sub trees of the exact same size. After both the source and target trees are hashed, all sub trees are extracted from the ASTs and

sorted by size. This way, all sub trees of the same size in both submissions can be looked up quickly.

3.1.3 Similarity detection algorithm

To find sub trees that are similar, we go through every sub tree of the source tree and look up all sub trees in the target tree of the same size. These are then cross-compared, and a pair of sub trees is decided to be similar if the hash values of their root nodes are equal. To avoid matching small code blocks that are very common or coincidentally equal, a minimum sub tree size parameter is used. Equal sub trees that are smaller than this minimum size will not be matched. The default value for this minimum size has been empirically set to 10 nodes, but could be varied by the user of CRDS to look at more and less fine grained similarity detection.

3.1.4 Reordering detection algorithm

With the CCS-based algorithm design so far, CRDS is able to identify similar code. This is the point where most plagiarism detection tools decide the result to be satisfactory. They build their user interface around this result, and teachers are expected to interpret the reasons for similarity and argue about why it is suspicious.

From this point however, more information about this similarity can be extracted to help the interpreter with these challenging tasks. CRDS extracts reordering in blocks of code that have otherwise remained the same. This means that the CCS-based similarity detection will already show these blocks as similar: after all, the concept of reordering is not meaningful when the reordered code blocks differ between the submissions.

Performance considerations

The reordering detection of CRDS is designed as an addition to the existing similarity detection. We will look for reorderings inside sub trees that have already been labelled as similar by their hash values. Because of this approach, an additional traversal of the submissions is not needed: the detection is merely an extra analysis next to the comparison of hash values. This means that the impact on performance is minimal.

The hashing approach for similarity detection is invariant under reordering. However, it would be preferred to avoid searching every sub tree that is similar, because we expect many direct similarities without reordering. To achieve this, an additional hash value is stored per node, that is not invariant under reordering. Instead of using addition of the children's hash values, they are appended to each other and the result is re-hashed. This way, a different order of children results in a different hash value. We only look for reorderings if these hash values are *not* equal.

Design

To find reordering in similar sub trees a heuristic approach is used. For every pair of sub trees, we check if the direct children are in the same order. If they are, the sub trees are not traversed further, because it is assumed that all child sub trees will be evaluated later in the execution.

By the design of the algorithm, this assumption is indeed correct if the children are themselves similar to each other. Since the entire sub tree is similar, their children will often be similar as well. However, our hashing approach is also invariant under vertical reordering inside the sub tree. For example, the two sub trees shown in Figure 3.1 will be labelled as similar, while they obviously are not. For our heuristic approach, this is not a problem. The sub trees below the root nodes will not be matched against each other, but they are different, so we are not interested in finding reordering within them.

If the direct children are not in the same order, an additional check is needed to see if the list of children can be reordered to be equal. It is possible, as is also visible in Figure 3.1, that the direct children of nodes with equal hash values are different. If this is not the case, a reordering has been detected.

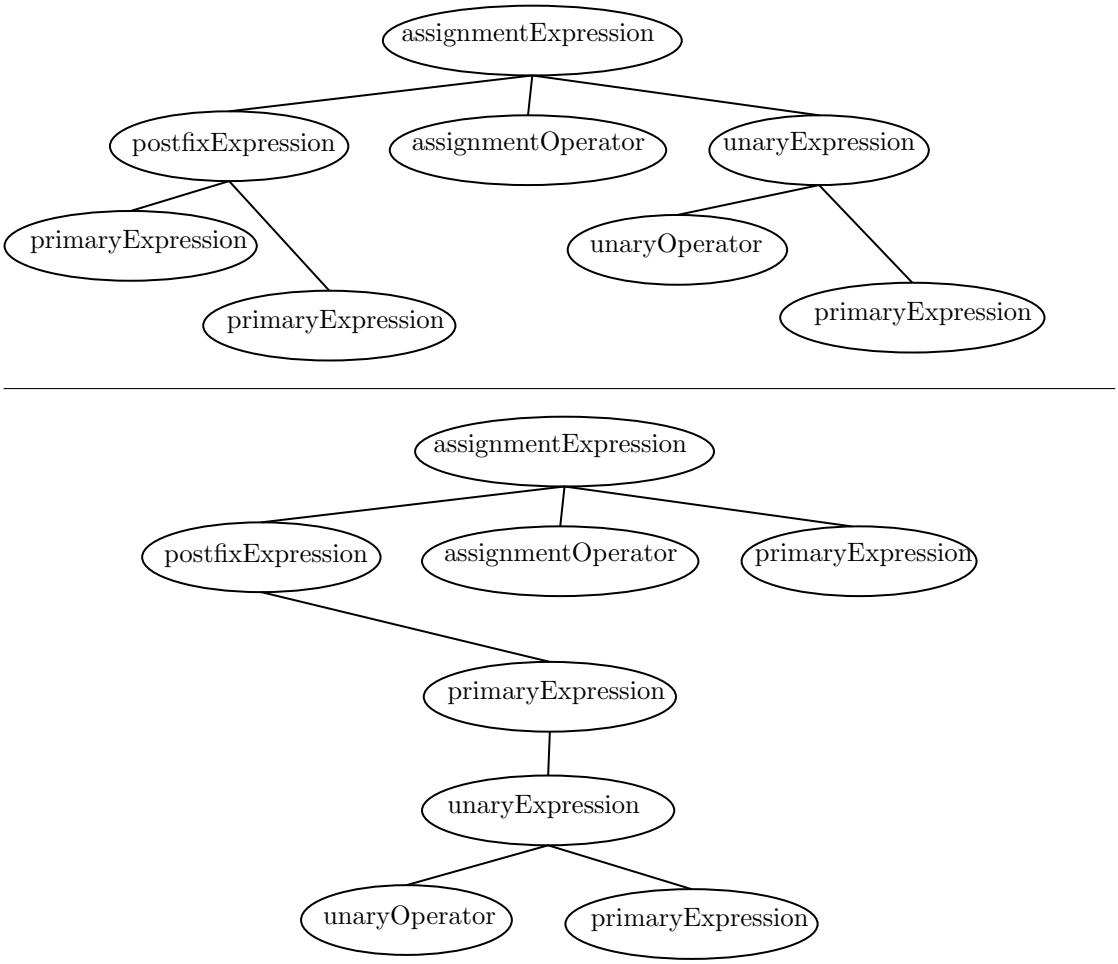

Figure 3.1: Two sub trees that are labelled as similar by CRDS, while they obviously are not. The hash value is invariant under reordering, both vertical and horizontal. This example also shows that while the root nodes have the same hash values, it is possible that the direct child nodes are different.

If a child is smaller than the minimum sub tree size, it will not be matched for similarity later in the execution. Yet, we are interested in finding reordering on such a small scale, for example within an arithmetic expression. It is possible not to use the size threshold for the detection. However, that would mean that we lose our restriction of only looking for reorderings within similar sub trees (sub trees below the size threshold are never labelled as similar individually). Instead, to avoid false positives due to noise, detection of these small-scale reorderings is only attempted within larger code blocks that are already labelled as similar. When the children of similar nodes are analysed for reordering, all children that are smaller than the threshold size are directly cross-compared by traversing them top-down. This way, we can find all reorderings on the smallest possible scale, but only when they are within a larger sub tree that is labelled similar.

Extracting results

Finding a reordering in our AST yields a list of same-level sub trees that were reordered. Showing all these children to a user is not useful, especially when there are many of them. Before being able to show reordering as a result in a user interface, it needs to be determined which sub trees were most likely moved within that list. To find an edit sequence from the source submission to the target submission, the algorithm shown in Algorithm 1 is used. For every child sub tree in

the source submission, it finds the first matching target sub tree, and registers a swap if these children are not in the same position. This algorithm yields an approximation of an optimal edit sequence between the source children and target children: finding the optimal transposition distance has been shown to be NP-hard [31]. Using the results of this algorithm, CRDS can highlight only the child trees that were most likely moved.

Algorithm 1 Reordering steps between lists of child sub trees

```
1: s ← list of source child nodes
2: t ← list of target child nodes
3: i ← 0
4: for all child_s ∈ s do
5:    if t[i] = child_s then
6:       remove t[i]
7:    else
8:       j ← 0
9:       for all child_t ∈ t do
10:         if child_s = child_t then
11:            add swap child_s ⟺ child_t to results
12:            remove t[j]
13:         end if
14:      end for
15:      j ← j + 1
16:   end if
17:   i ← i + 1
18: end for
```

3.2 Implementation

CRDS, our implementation of the presented algorithm design, was developed as an open source, publicly available "proof-of-concept" tool [32]. The implementation uses two traversals of the generated ASTs: a pre-processing step and a comparison step.

3.2.1 Hardware

For the development of CRDS and the execution of the experiments, a machine with the specifications shown in Table 3.1 was used.

Table 3.1: The specifications of the machine that was used in this thesis.

Specification	Value
Processor	Intel Core i5-7200U CPU, 2.50GHz × 4
RAM	8GB
System type	64-bit
OS	Ubuntu
OS version	20.10

3.2.2 Used tools

ANTLR

To process source code and generate Abstract Syntax Trees, we used the ANTLR [33] tool. ANTLR is a parser generator, that takes grammar files as input, and produces a parser for the language described by the grammar. For many programming languages, these grammar files are

openly available online. To add support for a new language to our tool, all that is needed is a grammar file to add a new parser.

Python

ANTLR is written in the Java programming language, and produces Java parsers by default. However, our solution is written in Python. This choice has been made because we have the ambition to integrate our algorithm into the Compare50 project, which is written in Python. ANTLR has functionality to create parsers written in Python, and this is suitable for most grammars without any modifications. Therefore, we found the drawbacks of using Python to be minimal.

3.2.3 Framework

For the implementation of CRDS, files are read from a submission directory to cross-compare them. Because source-code parsing and the generation of Abstract Syntax Trees are costly operations, this pre-processing step is executed once per submission. Because of this, the processed ASTs are stored in memory before the comparison operation. This way, execution times are acceptable, but RAM usage of CRDS grows rapidly with the amount of submissions. On the machine used for the implementation and experiments presented in this thesis, submissions of around 500 files can be read without problems caused by high RAM usage.

3.2.4 Pre-procesing step

Building Abstract Syntax Trees

When running the ANTLR tool with a grammar describing a programming language as input, it generates code for a lexer and a parser. To pre-process a submission, the source-code file is read and passed through the lexer, producing a token stream. This token stream is then passed to the parser, which generates a parse tree. ANTLR also generates a parse tree listener, that comes with a pre-built parse tree walker. With some modifications to this listener, CRDS walks the generated parse tree and builds the hashed AST from it. Both of these operations can be done within one full traversal of the parse tree.

Using the parse tree walker provided by ANTLR, the parse tree is traversed using depth-first search. During the downward pass of this traversal, the AST is built in a separate data structure. For each node in the parse tree, its type and location in the original source file are saved as a node in the AST. The tokens containing the syntax information are not stored, because the algorithm does not use this data.

A parse tree created by the ANTLR parsers contains many "wrapper" nodes: nodes that only have one child, and are a wrapper of the sub tree below it, without providing any additional information about the source code. These nodes are a more general specification of the code that is represented by the tree below it: for example, an if-statement is also a "statement". To save storage space, we filter out these wrapper nodes from the parse tree by excluding a node from the AST if it has exactly one child node.

Hashing and sorting Abstract Syntax Trees

During the backtracking of the depth-first search, additional data is stored in the AST nodes that depends on the children of the node. Both hash values can be calculated here, since the hash values of the children are known in the backtracking step. These hash values are stored in the AST node for the comparison.

The sub tree size is also kept during the upward pass and stored in the AST nodes. Every sub tree is then separately stored by size. For fast look-ups during the comparison step, a hash table data structure is used for storing the sub trees by size: with a look-up using the sub tree size as a key, a list of sub trees of that size can be obtained.

3.2.5 Comparison step

After the pre-processing step, the comparison is executed. For this step, the AST representations of all submissions are cross-compared. The sorted list of sub trees of the source submission is traversed. For every available sub tree size above the minimum size threshold, all same-size sub trees of the target submission are looked up. Then all source sub trees are cross-compared with all target sub trees.

Similarity detection

If two same-size sub trees have the same hash value, they are similar. The location in the source files of this similarity is stored for later use, to show the similarities in the user interface.

Reordering detection

If the second, more exact hash value is different between two similar sub trees, the reordering detection algorithm is executed. The lists of children of the root nodes are analysed. If we sort both, and they are not equal, we do not look further. If they are, we locate the reordering using the CRDS implementation of Algorithm 1. The file locations of any reordering that is found are stored as a result.

If any of the children are smaller than the minimum sub tree size, we traverse them recursively. We look for reorderings within these trees using the same algorithm. Because we can only compare sub trees of the same size, this recursion step is only executed if any same-size child sub trees are found in both the source and the target submissions.

3.2.6 Results

CRDS produces a similarity score for every pair of submissions. This similarity score is calculated as the fraction of lines in each file that was labelled similar. To show detailed results for one pair, HTML files are dynamically generated using a templating engine. CRDS generates three result files: a text file with the sorted similarity scores between every pair of submissions; a similarity "inverse code diff" user interface; and a similar highlighting user interface that shows the reorderings that were found.

3.3 Data sets

The experiments that have been set up to assess the effectiveness of CRDS and to answer the research question, are executed on two different data sets.

3.3.1 Real-world data set

To evaluate the effectiveness of CRDS on real students' submissions, experiments are conducted on the data set provided by Ljubovic [34]. This is a data set that consists of submissions of students during two introductory programming courses at the university of Sarajevo. One course was taught in C++, the other in C: since CRDS does not support C++ submissions yet, the submissions of the C course from 2016 are used for our experiments. This selection from the data set contains 18 assignments, each containing submissions created by up to 546 students.

What makes this data set especially useful, is that along with the submissions, information is provided about which students are suspected to have committed plagiarism. These suspicions are based on static analysis of the source code or failure by students to orally defend their submission.

3.3.2 Synthetic data set

To be able to experiment on specific plagiarism attacks, a synthetic data set is used. It consists of 20 highly varying code files that we created as solutions to assignments in previous years. There are 7 Python submissions and 13 C submissions in the data set. For our experiments, we will use these code files to execute different plagiarism attacks on. This way, it is possible to

evaluate the effectiveness of CRDS on different scales, under multiple circumstances and when specific attacks are used. This is difficult to achieve using real submissions, because they contain noise, and we can never be absolutely certain that plagiarism was actually committed.

3.4 Generation of reorderings

To make the synthetic data set usable and representative for testing CRDS, reorderings are introduced programmatically. To achieve this, the parse trees generated by ANTLR are used. Every code file in the data set is parsed, and traversed with the ANTLR-generated listener. When traversing the parse tree, we look for specific node types, of which the child nodes are reordered randomly. After all reorderings have been introduced, the source-code is regenerated from the parse tree. The entire synthetic data set is publicly available with CRDS [32].

The sub-questions, leading to an answer to the research question in this thesis, state four different scales of reordering we will experiment on. These scales are shown with a short description in Table B.5. For each scale, we reorder the entire data set in every possible place. Using this approach, for each of the four scales, a version of the entire data set is created with reorderings introduced.

These reorderings do not always leave the functionality of the code intact. Since source-code is sequential in nature, reordering without changing the output is only possible in specific cases. In AST representation, we do not have enough information to determine whether functionality remains the same: it represents code structure, not execution flow. Because of this, the introduction of reorderings will often change the functionality of the original code. Enforcing that functionality remains the same, as real plagiarism attacks often will, only limits the reorderings that are possible: it does not introduce new possibilities. Therefore, if our experiments prove the detection to be effective, it will also be effective when this enforcement is made.

Table 3.2: The different scales of reordering that are executed on the synthetic data set.

Scale	**Description**
Sub-statement	Reordering of parameters, arguments or arithmetic operands within a statement or expression
Statements	Reordering of single-line statements within the same scope
Functions	Reordering of functions
Conditionals	Reordering of conditional flow (cases in a switch statement or if-else statements with opposite conditional expressions)

3.4.1 Sub-statement

Sub-statement reorderings are reorderings within arithmetic expressions, function parameters and corresponding function calls. These attacks are executed often [12], as they are easy to execute without changing the program execution. Both Python and C have a node type for parameter lists, argument lists, multiplicative expressions and additive expressions: every time a node of one of these types is encountered in the AST, its children are reordered using a random shuffle function.

Because parameter names and literal values are not used in our AST representation, the reordered parameters and operands will not be detected if they only differ by name or value. CRDS will see them as equal, so there will be no detection of reordering unless the structure of the reordered parameters is different. Therefore, we do not expect to find all reorderings on such a small scale. To somewhat mitigate this drawback, we added a third hash value to CRDS for this specific case: names of parameters are hashed with the node type. The difference between running CRDS with and without using this new hash value will show how high the influence of this drawback is.

3.4.2 Statements

Reordering single statements within a scope is quite difficult to do without changing the code execution: the statements need to be loosely coupled, which most statements are not. When they are loosely coupled (multiple variable definitions, for example) these attacks are executed in practice by students [12]. As stated earlier, we cannot know if statements are loosely coupled in our AST representation, so our introduction of reorderings will almost always change the execution. Statements can be reordered under many different types of node: for any node that contains two or more simple (single-line) statements, these statements are reordered randomly.

3.4.3 Functions

Reordering of functions and methods is also a plagiarism attack that happens in real submissions [12]. In many programming languages, especially object oriented languages, method order is not relevant, making reordering easy to execute. While reordering functions may at first make similarity less obvious to a teacher when taking a look at the code, most existing tools will not be fooled by this attack, because the individual functions are often large enough to be matched individually. Still, finding reorderings on such a large scale can be a very strong indicator for plagiarism, making the detection useful. Using our AST, all children of a top-level node or Python class definition node are reordered if they are function definitions.

3.4.4 Conditionals

Switch-case statements can be reordered easily: as long as there is no fall-through behaviour between cases, they can be reordered randomly without changing the execution of the code. Reordering of if-else blocks is somewhat more complicated than all the other reordering attacks. To retain the program flow, when swapping the body of an if statement with the corresponding else statement, the condition also needs to be inverted. This means that in order to execute this reordering attack, the code needs to be edited.

Tao et al. [26] showed that normalising if-else blocks can make the AST-based algorithm that CRDS uses robust against these reorderings. Doing this, however, is quite a complex operation, and falls outside of the scope of this thesis. In order to experiment with reorderings on this scale, we ignore the conditional expressions completely, only evaluating reordering of the bodies of the if-else statements. These reorderings can easily be introduced using our AST approach.

3.5 Experiments

3.5.1 Comparison to MOSS

To evaluate the effectiveness of the similarity detection of CRDS, it is compared to the established string-matching-based tool MOSS [1]. To confirm that the AST-based similarity detection of CRDS is also effective for use on real data that contains noise, we compare the similarity results of the two tools using the real-world data set. Furthermore, previous studies showed that AST-based algorithms are more robust against reordering attacks than MOSS. Our synthetic data set with introduced reorderings is used to confirm this improvement introduced by using AST. With the results from these experiments, we will answer our first sub question:

> *How does similarity detection based on Abstract Syntax Trees compare to the established tool MOSS?*

Firstly, both CRDS and MOSS are run on the real-world data set. Unfortunately, MOSS is not an open source tool, and the only way to use it is via a web server. This server has a limit of the amount of files that can be cross-compared, limiting us to only 5 of the assignments from the data set that can be fully analysed by MOSS. Still, with a total of 573 submissions, we believe this data to be representative. For every assignment, the similarity scores produced by both tools are extracted. Using the data about plagiarising students that was provided with the

data set, the average similarity scores are calculated for student pairs that plagiarised and those that did not.

Secondly, with both tools, a similarity check is executed between every original file from the synthetic data set and every reordered variant of it. The similarity scores will show how much the effectiveness of similarity detection is affected by reorderings on different scales.

3.5.2 Reordering detection

To test the reordering detection introduced by CRDS, the synthetic data set is used, with the reorderings that were introduced on each scale. For each original source-code file, CRDS is first run with an exact copy of the submission, to find false positives. Obviously, reorderings should not be detected between two identical files. However, it is possible that we will find some small-scale false positives because of coincidental similarity between two different code blocks in a file, or because of the abstraction introduced by using Abstract Syntax Trees.

The comparison between the original submissions and all reordered versions of those submissions are executed with CRDS. The amount of reorderings found is recorded. We also analyse a sample of the false negatives, in order to understand why some reorderings are not detected.

To confirm whether reordering detection is useful for plagiarism detection in real-world submissions, we run the reordering detection algorithm on the real-life data set as well. We report how many reorderings are found within submissions that were likely plagiarised, versus submissions that were not. Additionally, by manually inspecting a representative sample of the reorderings that are found, we assess the rate of false positives found in real submissions.

With the results of these experiments, we are able to answer our sub questions about the detection of reordering:

- *How effectively can CRDS detect sub-statement reorderings?*
- *How effectively can CRDS detect statement reorderings?*
- *How effectively can CRDS detect function reorderings?*
- *How effectively can CRDS detect reordering within conditional statements?*

3.6 Evaluation

From the results of our experiments we expect to be able to properly answer our research question. We thoroughly examine the effectiveness of the reordering detection, by using a representative data set, introducing reorderings on different scales and by examining the relevance using real, noisy submissions. By using a combination of synthetic data and real-world data, we eliminate some weaknesses that lie in using only one of these options.

The real-world data set is very useful to test our algorithm on a lot of data. It can confirm the usability and effectiveness of CRDS on real submissions. It is also greatly fit to create a comparison with existing tools. But, even though there is information about suspected plagiarisers, identifying plagiarism and reordering attacks with complete certainty is not possible. Therefore, it is not fit for testing our reordering detection algorithm thoroughly.

To test our reordering detection algorithm, synthetic data is very useful because we have full control over it. When executing attacks, we know exactly what we expect our results to be. However, experimenting on self-created data could also introduce a bias, because this data might be constructed in a way that it will be detected more often. Also, it lacks noise and other modifications besides reordering. We reduce the influence of a possible bias by only executing attacks that were shown to be executed often by students, and by executing them randomly. Additionally, the reorderings found by CRDS in the real-life data set will make our results more reliable by proving that reorderings are also found when the submissions are noisy.

CHAPTER 4

Results

4.1 Comparison to MOSS

4.1.1 Real-world data

For every assignment in the real-world data set that MOSS was able to handle fully, the average similarity scores of plagiarised and non-plagiarised submission pairs are presented in Table 4.1. For every assignment, both MOSS and CRDS show a clear-cut difference between suspected plagiarism and original solutions. The average similarity scores over all submissions are shown in Figure 4.1. MOSS scores slightly higher in similarity between plagiarised submissions. Overall, both tools are clearly able to identify plagiarism, scoring suspicious pairs over 50% similar on average.

Table 4.1: The average similarity scores of CRDS and MOSS on the assignments of the real-world data set, separately calculated for plagiarised and non-plagiarised submissions.

Assignment	**CRDS**		**MOSS**	
	Plagiarism	**Not Plagiarism**	**Plagiarism**	**Not Plagiarism**
4.2	53.04 %	9.66 %	58.69 %	10.54 %
4.3	77.72 %	10.48 %	85.24 %	11.10 %
4.4	61.60 %	5.42 %	81.29 %	7.16 %
5.2	49.09 %	10.00 %	62.67 %	9.95 %
5.3	24.43 %	12.12 %	20.32 %	6.06 %

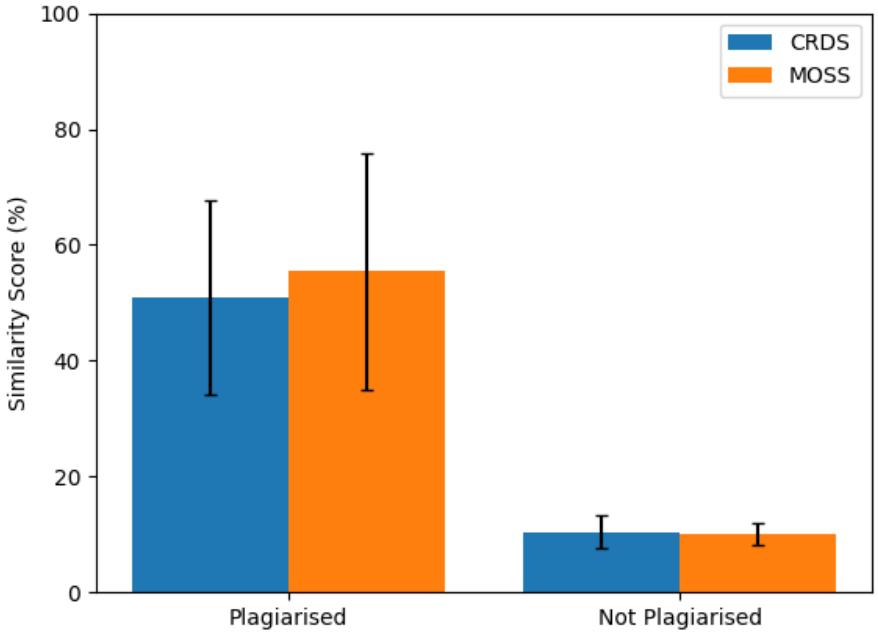

Figure 4.1: The average similarity scores for CRDS and MOSS, for plagiarised and non-plagiarised submissions in the real-world data set. The difference between suspected plagiarism and other submissions is clearly visible for both tools.

4.1.2 Influence of reordering

Table 4.2 shows the results of the comparison experiments run on the synthetic data set. The complete data making up these results is available in Appendix A. In accordance with previous research regarding AST-based similarity detection, CRDS is not affected at all by the reorderings that were introduced: all average scores remain above 94%.

The similarity score produced by MOSS is especially affected by small scale reorderings, with statement reorderings bringing the average similarity score down to 41.21%. The large scale reorderings, functions and conditionals, do not affect the similarity score much.

Table 4.2: The average similarity scores of CRDS and MOSS between the original code files in the synthetic data set and every reordered version of that original file.

Tool	**Similarity score (%)**			
Scale:	**Sub-statement**	**Statements**	**Functions**	**Conditionals**
CRDS	94.33	94.26	94.30	94.36
MOSS	59.53	41.21	86.21	88.53

4.2 Reordering Detection

4.2.1 Synthetic data

Four each of the four scales we tested our reordering detection on, we assess the effectiveness by examining the amount of true positives and false negatives. Some files yield false positives when CRDS is run with an exact copy of the file: to get the amount of true positives, we subtracted these false positives from the amount of detected reorderings. The averages of these results are presented for every scale in Table 4.3. Figure 4.2 shows a comparison between the four scales in detection effectiveness. The full results for every file are available in Appendix B.

Table 4.3: The average results of the reordering detection experiment on each scale.

Scale	False positives	Introduced	Detected	True positives	False negatives	Found (%)
Sub-statement:	1.68	31.37	17.79	16.11	15.26	56.62
Statements:	1.68	13.63	12.89	11.21	2.42	83.07
Functions:	1.88	1.06	2.94	1.06	0.00	100.00
Conditionals:	2.29	4.79	5.43	3.14	1.64	59.19

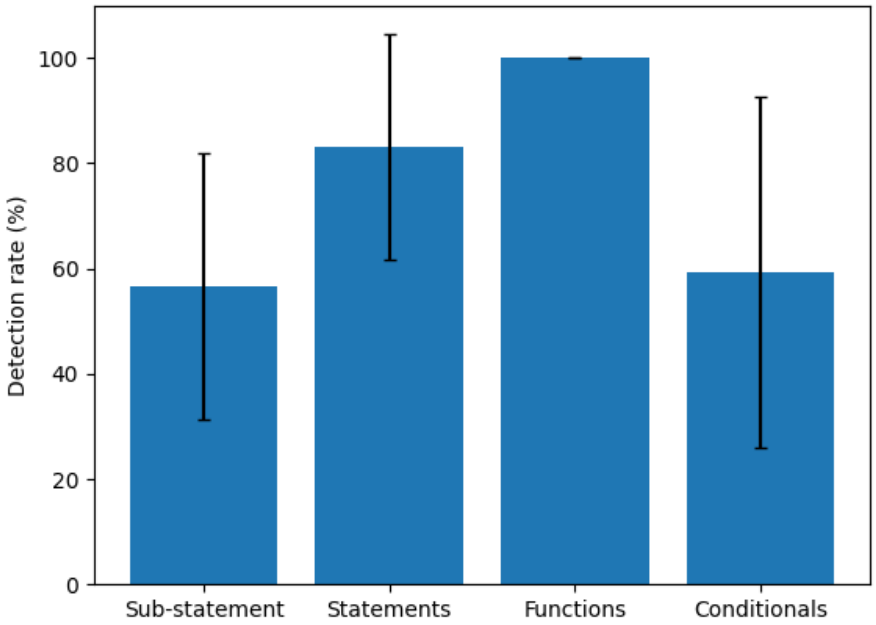

Figure 4.2: The average reordering detection rate for each of the four scales.

Notably, all function reorderings are detected. Statement reordering is also effective with a detection rate of 83%. While the detection rates of the sub-statement and conditionals scales are lower, none are below 50%: detection is possible on every scale.

Influence of parameter names

As discussed in 3.4.1, we expect sub-statement detection rates to be lowered by the fact that function parameter names are seen as equal by the AST because their names are not used. To test this hypothesis, an additional hash value was introduced to CRDS that includes the parameter names in the hash. This addition yields an average improvement of 7% in the detection of sub-statement reorderings over the original hash value. This improvement is visible in Figure 4.3.

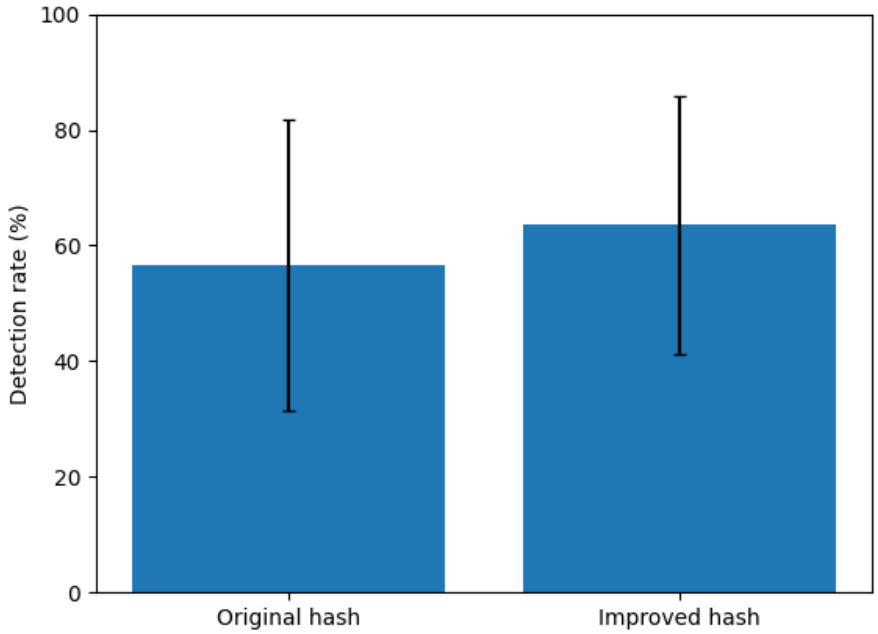

Figure 4.3: The difference in sub-statement reordering detection rate between the original hash value and the improved hash value that includes the names of function parameters.

4.2.2 Real-world data

Through the entire real-world data set of 6.6 thousand submissions across 18 assignments, 8327 reorderings have been found. Figure 4.4 shows the average amount of reorderings between plagiarised pairs of submissions, versus pairs that are not labelled as suspicious.

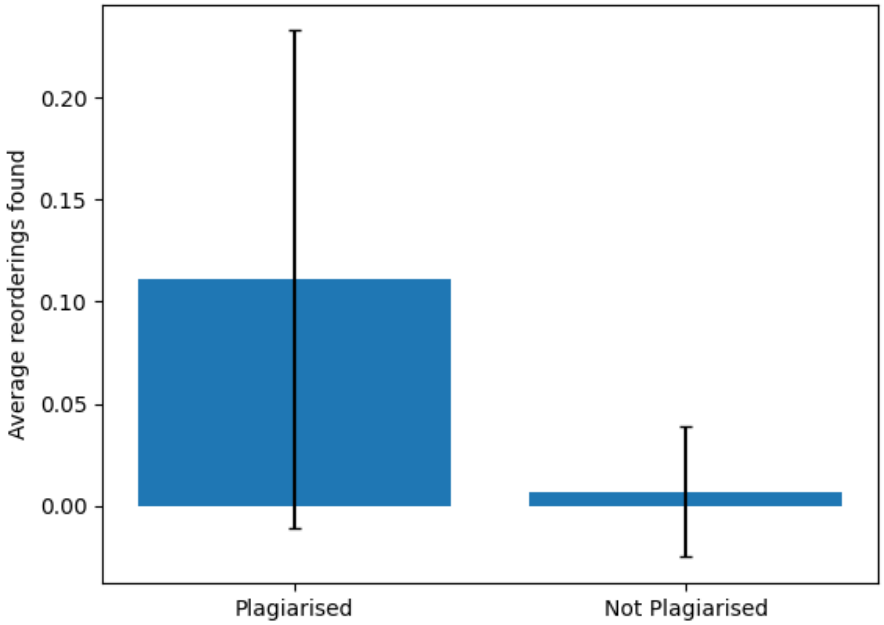

Figure 4.4: The average amount of detected reorderings between suspicious pairs of submissions and non-suspicious pairs of submissions.

From this figure, it is clear that we find more reorderings between submissions that are suspected of plagiarism. Still, however, the amount of reorderings found within these suspicious pairs, is only 0.11 on average. The standard deviation in these results is quite high, indicating that the distribution of the detected reorderings among the submission pairs is not uniform: they are grouped in a relatively small selection of submission pairs that often yield multiple reorderings.

CHAPTER 5

Discussion

5.1 AST-based similarity detection

To confirm the effectiveness of the plagiarism detection technique based on Abstract Syntax Trees that was introduced in previous research, we compared CRDS with the established and widely used tool MOSS. The results of this comparison show that the similarity detection based on ASTs can compete with winnowing, the string-matching-based algorithm used by MOSS. Figure 4.1 shows that the difference between the results from these two tools is minimal on a representative data set. We conclude that CRDS is effective in identifying similar code.

Automatic source-code plagiarism detection based on Abstract Syntax Trees was introduced in previous research as an improvement over string-matching-based algorithms like winnowing. It was shown to be more robust against reordering attacks [3, 5]. To confirm this improvement, we tested the similarity produced by CRDS and MOSS between code files and reordered versions. Table 4.2 shows that CRDS indeed performs significantly better than MOSS when code is heavily reordered. In fact, CRDS is not affected by these reorderings at all: the only reason that the scores produced by CRDS are not all 100% is the fact that lines that contain comments are not labelled as similar. This corresponds with the method used by MOSS to calculate its similarity scores. MOSS is especially affected by reorderings on a small scale: statement reordering reduces the average similarity score to 41%. These small scale reorderings break up the fingerprints that the winnowing algorithm uses. Reordering at a larger scale, namely functions and code blocks in conditionals, affects MOSS's scores a lot less. These larger code blocks are still matched individually because they are larger than the fingerprint size.

A nuance that needs to be made here, is that in our generated code files, reordering is applied in every possible place, without retaining code functionality. This will never happen in real plagiarised submissions, where reordering will be much less frequent. Because of this, and considering that MOSS's scores in Table 4.2 are not extremely low, we expect this limitation of MOSS to have little impact on its usage in practice. This may also be the reason that the improvement that is provided by using AST has not been widely adopted: the problem it solves may not be that much of a concern to teachers.

5.2 Existing vulnerabilities

While using Abstract Syntax Trees removes some vulnerability to reordering attacks, there are still certain other attacks that both CRDS and MOSS are vulnerable to [5, 12]. Complex structural attacks, like splitting a function into multiple small ones, or replacing code with semantic equivalents that use different syntax, can still fool these techniques. However, these attacks are quite difficult to execute and less frequently observed in practice [12, 13]. Adding a lot of dummy statements is also an effective attack as it breaks up MOSS's fingerprints and changes the hash value of the AST sub trees used by CRDS. Adding high amounts of dummy code however, can easily be spotted if grading is not entirely automated.

5.3 Reordering detection

Our belief is that the problem of interpreting and explaining code similarity results is much more pressing than improving robustness against very advanced plagiarism attacks. We used the structural information that is present in Abstract Syntax Tree representation to provide additional insights in the reasons for similarity. We implemented a heuristic reordering detection algorithm in CRDS and experimented on it using automatically generated reorderings in source-code. These reorderings were introduced on four different scales: functions, statements, conditionals, and sub-statement reordering.

Table 4.3 shows that reordering of functions in our data set is detected 100% of the time. Functions are large isolated code blocks, that are often not similar to each other. This makes detection of reordering very effective.

When reorderings happen on a smaller scale, within statements, we see the detection rate drop to 56%. As we already hypothesised earlier, this is because on a small scale, the abstraction introduced by the transformation to AST removes information needed to find the reorderings. As is visible in Figure 4.3, adding identifier names to the hash value can increase this detection rate somewhat. The detection of single line statements worked well in both the synthetic data set and the real-world data set. This scale seems to be just large enough to remain distinguishable after the abstraction is introduced.

5.4 Weaknesses

By inspecting the false negatives from the reordering detection on the synthetic data, as well as the detected reorderings in the real-world data set, we empirically found a vulnerability to the approach used by CRDS. By definition, finding a reordering is only possible between two equal sets of sub trees. This means that any small structural change, like the addition or removal of a statement, makes finding reorderings on a level above this change impossible.

In combination with the weakness of too much abstraction on a small scale, this limits detection of reordering in real submissions. In real submissions, most reorderings found are on statement scale: on this scale, chances are highest that no changes were made within the sub trees, but the scale is large enough to retain its differences after the abstraction is applied.

5.5 Ethical Aspects

As the automatic detection of source-code plagiarism becomes more advanced, confidence in the results generated by these tools can grow with teachers. This may lead to over-reliance on these results for raising suspicion about plagiarism. When the output of a tool becomes leading, false positives and incorrect results can be the cause of punishment for innocent students. Also, code similarity is not synonymous with plagiarism: some assignments are very specific, causing many solutions to be similar. It is therefore imperative that suspicion from automatic plagiarism detection tools is never treated as absolute: teachers must always use their own judgement to raise suspicion about plagiarism. Automatic detection tools should never be more than tools to aid teachers in this task.

As explained in Section 2.1, definitions and understanding about plagiarism can vary between students and teachers. To avoid unnecessary cases of plagiarism, that are not caused by any unethical internal motivations of students but are rather accidental, it is important that guidelines about collaboration and code reuse are clearly stated by universities.

5.6 Future Research

Before CRDS can be used in practice, some improvements are needed. Firstly, a user friendly interface needs to be designed to actually make reordering results useful to the interpreter. Secondly, a performance improvement needs to be made. RAM usage is currently not acceptable for analysing large sets of submissions. Many other optimisations could be made to CRDS: pre-processing to remove unused code could make CRDS robust to this attack. Also, to mitigate the influence of very small changes, it would be interesting to try out using a parameter for a minimum size for a sub tree to be included in the hash value of its parent. This way, small, local changes would not be reflected in parent nodes, which might make detection of large scale reorderings more frequent in practice.

CHAPTER 6

Conclusion

In this thesis, we used Abstract Syntax Trees to create a plagiarism detection tool we named Code Reordering Detection System (CRDS). We researched the following question:

> **How effectively can code reordering be detected using Abstract Syntax Trees?**

We implemented a heuristic algorithm for detecting reorderings of similar source code between submissions. We then experimented on this algorithm, guided by the following five sub questions that will lead to an answer to our research question.

- *How does similarity detection based on Abstract Syntax Trees compare to the established tool MOSS?*
- *How effectively can CRDS detect sub-statement reorderings?*
- *How effectively can CRDS detect statement reorderings?*
- *How effectively can CRDS detect function reorderings?*
- *How effectively can CRDS detect reordering within conditional statements?*

We found that AST-based similarity detection can compete with MOSS when it comes to detecting plagiarism on real students' submissions. AST-based similarity detection outperforms MOSS when many reordering attacks are executed on the plagiarised code.

Our findings show that sub-statement reorderings can be detected by CRDS. On such a small scale however, reorderings can be missed because of the abstraction created by the AST-based representation. Because of the same reason, false positives on a small scale are quite common. On a larger scale, false positives are far less common. Reordering detection is very effective with statements and functions. However, by definition, reordering detection on a large scale is only possible when all code blocks on that scale are not otherwise structurally edited. Therefore, statement reordering is the most common finding in real submissions. To effectively detect reordering in conditional statements, an extension of the algorithm would be needed: currently, CRDS is not able to identify changing conditional statements.

We conclude that CRDS can detect most reorderings effectively on any scale, when code is not otherwise structurally edited on that scale. In practice, when submissions are noisy, we conclude that mainly small-scale reordering is detected. With some improvements to the usability, performance and prevention of false positives, CRDS has potential to be used to provide extra explanation of similarity results to the interpreter of plagiarism detection tools.

Bibliography

[1] Stanford University. *MOSS A System for Detecting Software Similarity*. URL: `http://theory.stanford.edu/~aiken/moss/` (visited on 04/06/2021).

[2] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. *JPlag: Finding plagiarisms among a set of programs*. Citeseer, 2000.

[3] Baojiang Cui et al. "Code comparison system based on abstract syntax tree". In: *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*. IEEE. 2010, pp. 668–673.

[4] Deqiang Fu et al. "WASTK: a weighted abstract syntax tree kernel method for source code plagiarism detection". In: *Scientific Programming* 2017 (2017).

[5] Lawton Nichols et al. "Syntax-based improvements to plagiarism detectors and their evaluations". In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 2019, pp. 555–561.

[6] Hiroshi Kikuchi et al. "A source code plagiarism detecting method using alignment with abstract syntax tree elements". In: *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE. 2014, pp. 1–6.

[7] 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.

[8] UCAS. *UCAS undergraduate sector-level end of cycle data resources 2020*. URL: `https://www.ucas.com/data-and-analysis/undergraduate-statistics-and-reports/ucas-undergraduate-sector-level-end-cycle-data-resources-2020` (visited on 04/30/2021).

[9] Harvard University. *CS50 Introduction to Computer Science*. URL: `https://online-learning.harvard.edu/course/cs50-introduction-computer-science` (visited on 04/06/2021).

[10] Carolyn Duffy Marsan. "Why computer science students cheat". In: *PC World, April April* (2010).

[11] Georgina Cosma and Mike Joy. "Towards a definition of source-code plagiarism". In: *IEEE Transactions on Education* 51.2 (2008), pp. 195–200.

[12] 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.

[13] Oscar Karnalim. "Python source code plagiarism attacks on introductory programming course assignments". In: *Themes in Science and Technology Education* 10.1 (2017), pp. 17–29.

[14] Jurriaan Hage, Peter Rademaker, and Nike van Vugt. "A comparison of plagiarism detection tools". In: *Utrecht University. Utrecht, The Netherlands* 28.1 (2010).

[15] Oscar Karnalim, William Chivers, et al. "Similarity Detection Techniques for Academic Source Code Plagiarism and Collusion: A Review". In: *2019 IEEE International Conference on Engineering, Technology and Education (TALE)*. IEEE. 2019, pp. 1–8.

[16] Daniela Chuda et al. "The issue of (software) plagiarism: A student view". In: *IEEE Transactions on Education* 55.1 (2011), pp. 22–28.

[17] Cheryl L Aasheim et al. "Plagiarism and programming: A survey of student attitudes". In: *Journal of information systems education* 23.3 (2012), pp. 297–314.

[18] Trip Gabriel. "Plagiarism lines blur for students in digital age". In: *The New York Times* 1.8 (2010).

[19] 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.

[20] Thomas Lancaster and Fintan Culwin. "A comparison of source code plagiarism detection engines". In: *Computer Science Education* 14.2 (2004), pp. 101–112.

[21] Harvard CS50. *Compare50, a fast and extensible plagiarism-detection tool*. URL: `https://cs50.readthedocs.io/projects/compare50/en/latest/` (visited on 04/06/2021).

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

[23] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. "Finding plagiarisms among a set of programs with JPlag". In: *J. UCS* 8.11 (2002), p. 1016.

[24] Vítor T Martins et al. "Plagiarism detection: A tool survey and comparison". In: (2014).

[25] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. URL: `http://dinosaur.compilertools.net/yacc/index.html` (visited on 06/04/2021).

[26] 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*. IEEE. 2013, pp. 714–719.

[27] Temple F Smith, Michael S Waterman, et al. "Identification of common molecular subsequences". In: *Journal of molecular biology* 147.1 (1981), pp. 195–197.

[28] Compare50. *Compare50 Docs*. URL: `https://cs50.readthedocs.io/projects/compare50/en/latest/` (visited on 05/06/2021).

[29] Jonathan YH Poon et al. "Instructor-centric source code plagiarism detection and plagiarism corpus". In: *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. 2012, pp. 122–127.

[30] Ronald Rivest and S Dusse. *The MD5 message-digest algorithm*. 1992.

[31] Laurent Bulteau, Guillaume Fertin, and Irena Rusu. "Sorting by transpositions is difficult". In: *SIAM Journal on Discrete Mathematics* 26.3 (2012), pp. 1148–1180.

[32] P. Decroos. *CRDS*. `https://github.com/philok55/CRDS`. 2021.

[33] Terence J. Parr and Russell W. Quong. "ANTLR: A predicated-LL (k) parser generator". In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810.

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

APPENDIX A

Full results of MOSS comparison

Table A.1: The similarity scores calculated by MOSS for the synthetic data set with multi scale reorderings.

File name	**Similarity score (%)**			
Scale:	**Sub-statement**	**Statements**	**Functions**	**Conditionals**
mm.c	33	19	94	88
scheduler-MemoryEfficient.c	78	18	90	64
shell.c	65	21	82	56
assign_1_3.c	68	0	98	99
simulate.c	18	82	98	81
array.c	13	0	74	90
heap.c	19	59	87	91
list.c	99	35	85	76
set.c	55	45	50	96
stack.c	85	42	57	99
tree.c	59	53	75	79
bezier.c	31	32	93	92
shaders.c	38	27	83	99
client.py	77	62	97	97
lab4.py	39	42	99	93
Lab5.py	96	70	98	99
lab6.py	87	54	95	91
server.py	80	52	96	93
sudoku.py	91	70	87	99
Average:	60	41	86	89

Table A.2: The similarity scores calculated by CRDS for the synthetic data set with multi scale reorderings.

File name	**Similarity score (%)**			
Scale:	**Sub-statement**	**Statements**	**Functions**	**Conditionals**
mm.c	91.86	91.86	91.86	91.86
scheduler-MemoryEfficient.c	92.92	92.92	92.92	92.92
shell.c	95.45	95.44	95.44	95.47
assign_1_3.c	92.31	92.31	92.31	92.31
simulate.c	93.42	93.42	93.42	93.42
array.c	96.47	96.47	96.47	96.47
heap.c	92.56	92.56	92.56	92.57
list.c	96.8	96.8	96.8	96.81
set.c	84.19	83.68	83.68	83.68
stack.c	88.71	88.71	88.71	88.71
tree.c	96.65	96.65	96.65	96.66
bezier.c	89.73	89.73	89.73	89.73
shaders.c	91.44	91.44	91.44	91.44
client.py	99.91	99.85	99.91	99.91
lab4.py	98.97	98.84	98.97	98.97
Lab5.py	99.41	99.41	99.41	99.56
lab6.py	99.83	99.81	99.83	99.91
server.py	99.09	98.77	99.09	99.41
sudoku.py	92.48	92.31	92.5	93.06
Average:	94.33	94.26	94.30	94.36

APPENDIX B

Full results of detection experiments

Table B.1: The results of the reordering detection experiment on sub-statement scale.

File name	False positives	Introduced	Detected	True positives	False negatives	Found (%)
mm.c	0	28	20	20	8	71.43
scheduler-MemoryEfficient.c	6	7	12	6	1	85.71
shell.c	0	24	8	8	16	33.33
assign_1_3.c	0	17	8	8	9	47.06
simulate.c	0	103	48	48	55	46.60
array.c	0	10	10	10	0	100.00
heap.c	0	75	49	49	26	65.33
list.c	18	6	20	2	4	33.33
set.c	0	6	6	6	0	100.00
stack.c	0	2	2	2	0	100.00
tree.c	2	36	24	22	14	61.11
bezier.c	0	39	17	17	22	43.59
shaders.c	0	48	36	36	12	75.00
client.py	0	9	3	3	6	33.33
lab4.py	0	29	13	13	16	44.83
Lab5.py	0	22	6	6	16	27.27
lab6.py	6	89	40	34	55	38.20
server.py	0	26	9	9	17	34.62
sudoku.py	0	20	7	7	13	35.00
Average:	1.68	31.37	17.79	16.11	15.26	56.62

Table B.2: The results of the reordering detection experiment on sub-statement scale, when function parameter names are included in the comparison operation.

File name	**False positives**	**Introduced**	**Detected**	**True positives**	**False negatives**	**Found (%)**
mm.c	0	28	20	20	8	71.43
scheduler-MemoryEfficient.c	6	7	10	4	3	57.14
shell.c	0	24	8	8	16	33.33
assign_1_3.c	0	17	8	8	9	47.06
simulate.c	0	103	49	49	54	47.57
array.c	0	10	10	10	0	100.00
heap.c	0	75	49	49	26	65.33
list.c	18	6	24	6	0	100.00
set.c	0	6	6	6	0	100.00
stack.c	0	2	2	2	0	100.00
tree.c	2	36	27	25	11	69.44
bezier.c	0	39	19	19	20	48.72
shaders.c	0	48	37	37	11	77.08
client.py	0	9	4	4	5	44.44
lab4.py	0	29	13	13	16	44.83
Lab5.py	0	22	9	9	13	40.91
lab6.py	6	89	58	52	37	58.43
server.py	0	26	12	12	14	46.15
sudoku.py	0	20	11	11	9	55.00
Average:	1.68	31.37	19.79	18.11	13.26	63.52

Table B.3: The results of the reordering detection experiment on statements scale.

File name	**False positives**	**Introduced**	**Detected**	**True positives**	**False negatives**	**Found (%)**
mm.c	0	10	10	10	0	100.00
scheduler-MemoryEfficient.c	6	10	12	6	4	60.00
shell.c	0	24	22	22	2	91.67
assign_1_3.c	0	7	7	7	0	100.00
simulate.c	0	11	5	5	6	45.45
array.c	0	8	8	8	0	100.00
heap.c	0	9	9	9	0	100.00
list.c	18	18	34	16	2	88.89
set.c	0	2	2	2	0	100.00
stack.c	0	3	3	3	0	100.00
tree.c	2	20	20	18	2	90.00
bezier.c	0	7	6	6	1	85.71
shaders.c	0	8	8	8	0	100.00
client.py	0	4	1	1	3	25.00
lab4.py	0	8	7	7	1	87.50
Lab5.py	0	22	18	18	4	81.82
lab6.py	6	63	57	51	12	80.95
server.py	0	18	10	10	8	55.56
sudoku.py	0	7	6	6	1	85.71
Average:	1.68	13.63	12.89	11.21	2.42	83.07

Table B.4: The results of the reordering detection experiment on functions scale.

File name	**False positives**	**Introduced**	**Detected**	**True positives**	**False negatives**	**Found (%)**
mm.c	0	1	1	1	0	100.00
scheduler-MemoryEfficient.c	6	1	7	1	0	100.00
shell.c	0	1	1	1	0	100.00
assign_1_3.c	0	1	1	1	0	100.00
simulate.c	0	1	1	1	0	100.00
array.c	0	1	1	1	0	100.00
heap.c	0	1	1	1	0	100.00
list.c	18	1	19	1	0	100.00
set.c	0	1	1	1	0	100.00
stack.c	0	1	1	1	0	100.00
tree.c	2	1	3	1	0	100.00
bezier.c	0	1	1	1	0	100.00
shaders.c	0	1	1	1	0	100.00
Lab5.py	0	1	1	1	0	100.00
lab6.py	6	2	8	2	0	100.00
server.py	0	1	1	1	0	100.00
sudoku.py	0	1	1	1	0	100.00
Average:	1.88	1.06	2.94	1.06	0.00	100.00

Excluded (contained only one function):	
client.py	
lab4.py	

Table B.5: The results of the reordering detection experiment on conditionals scale.

File name	**False positives**	**Introduced**	**Detected**	**True positives**	**False negatives**	**Found (%)**
mm.c	0	2	1	1	1	50.00
scheduler-MemoryEfficient.c	6	2	8	2	0	100.00
shell.c	0	11	9	9	2	81.82
simulate.c	0	6	6	6	0	100.00
array.c	0	2	2	2	0	100.00
heap.c	0	2	1	1	1	50.00
list.c	18	7	24	6	1	85.71
tree.c	2	9	6	4	5	44.44
bezier.c	0	2	1	1	1	50.00
shaders.c	0	1	0	0	1	0.00
lab4.py	0	3	2	2	1	66.67
Lab5.py	0	2	0	0	2	0.00
lab6.py	6	12	14	8	4	66.67
server.py	0	6	2	2	4	33.33
Average:	2.29	4.79	5.43	3.14	1.64	59.19

Excluded (contained no switch or if-else statements):	
assign_1_3.c	
set.c	
stack.c	
client.py	
sudoku.py	