Oscar Karnalim

TF-IDF-INSPIRED DETECTION FOR CROSS-LANGUAGE SOURCE CODE PLAGIARISM AND COLLUSION

Abstract *Several computing courses allow students to choose which programming language they want to use for completing a programming task. This can lead to cross-language code plagiarism and collusion in which the copied code file is rewritten in another programming language. In response, this paper proposes a detection technique that is able to accurately compare code files written in various programming languages but with limited effort in accommodating such languages at the development stage. The only language-dependent feature used in the technique is a source code tokenizer; no code conversion is applied. The impact of coincidental similarity is reduced by applying a TF-IDF-inspired weighting in which rare matches are prioritized. Our evaluation shows that the technique outperforms common techniques in academia for handling language-conversion disguises. Furthermore, it is comparable to these techniques when dealing with conventional disguises.*

Keywords source code plagiarism and collusion, cross-language detection, TF-IDF, computing education, information retrieval

Citation Computer Science 21(1) 2020: 113–136

1. Introduction

In some computer science courses, students are free to choose their preferred programming language for completing assignments. This can lead to an issue when plagiarism or collusion occurs; most existing detection techniques are not specifically designed to handle such cases in a cross-language manner where the involved source code files are written in different programming languages [2, 21].

Some detection techniques have addressed the issue by considering the source code as raw text [7, 50], removing any needs for language-specific components. Even though this kind of approach is applicable, it may lack effectiveness [43, 44]. Occasionally, a given source code can be inaccurately tokenized since text grammars are different from source code grammars.

To maintain the tokenization (and detection) accuracy, two language-dependent solutions exist: converting one code to another programming language [33, 39], and converting both codes to a particular intermediate format [5, 34]. These solutions are backed with the claim that most programming languages share the same features (and these features are reversible across the languages). However, the claim only works when the programming languages are equally developed. For an extreme example, Pascal does not have the list comprehension that can be found in Python, since such comprehension has just recently been found useful and Pascal's development is slower than Python's. Another possible issue related to such a conversion is that the mapping programming syntax from one language to another can be demanding; some programming languages are extremely different in terms of lexical and structural representation (e.g., R and Java).

This paper proposes a language-dependent technique that accurately parses source code tokens but with less effort in accommodating new programming languages; it only requires one source code tokenizer per language, which can be easily obtained with the help of ANTLR [38] and its predefined grammars[1]. Code conversion (which complicates the accommodation of new programming languages on existing language-dependent techniques) is excluded and the token strings are compared directly, assuming that some crucial tokens for raising suspicion (e.g., identifiers, keywords, constants, and arithmetic operators) are not affected by programming language conversion. Considering only non-coincidental matches can be used for raising suspicions, a TF-IDF-inspired weighting [13] is also applied. This accentuates the impact of such matches by assigning a higher score to the rare ones.

Among language-dependent techniques, our technique seems to be the first of its type that requires only ANTLR tokenizers for detecting cross-language source code plagiarism and collusion. It also seems to be the first one that uses TF-IDF-inspired weighting in accentuating the impact of non-coincidental matches between tokens from different programming languages. Due to its language dependency, the parsing mechanisms are more accurate than those that ignore language-specific features (e.g., treating source code as text).

[1]https://github.com/antlr/grammars-v4

2. Related works

Source code plagiarism and collusion occur when a person steals another person's source code and claims it as theirs [11,19,47]. Several automated detection techniques have been proposed to capture such a misbehavior [49]. They commonly compare the source code files in a pairwise manner in which pairs with high similarity degrees are considered to be suspicious. It is true that a high similarity can occur due to various reasons besides plagiarism and collusion [55]. For example, some students may use the same library to solve a particular task. Hence, the suspected pairs need to be observed further by examiners on most occasions [35].

Based on how they measure similarity, these detection techniques can be classified into two categories: attribute counting-based and structure-based techniques [3]. The former measures similarity by counting how many similar characteristics are shared between given source code files, while the latter focuses more on comparing their structures.

Ottenstein's work [37] is argued to be the earliest among the attribute counting-based techniques. It relies on four Halstead metrics [22] to determine source code similarity. Arguing that these metrics were not representative, the metrics was then extended by considering more factors [3, 16] in which some of them are contextual (like source code tokens) [4, 23]. The similarity measurement was also updated by adopting algorithms from other domains such as information retrieval [12, 18, 26], clustering [1,24,36], and classification [50,56]. Occasionally, algorithms from some of these domains were applied altogether [14,46].

Structure-based techniques are argued to be at least as effective as attribute counting-based techniques [52], even though they are considerably slower [49]. These techniques rely on various structures for comparison. Some of the structures are source code token strings [8,29,41], abstract syntax trees [20,30,53], parse trees [48], program dependency graphs [32], and low-level token strings [25,42].

Since attribute counting-based and structure-based techniques have their own characteristics and combining them may lead to more benefits [25], hybrid techniques have been introduced. A structure-based technique can be more time-efficient if the source code pairs are limited to those that share a high attribute counting-based similarity degree [9,49]. On the contrary, an attribute counting-based technique can be more effective if its results are combined with a structure-based technique's result [15,40,45].

Most detection techniques assume that all inputted source code files are written in the same programming language [2,21]. This can be an issue when more than one programming language is allowed to complete a particular assessment. Plagiarism and collusion may involve two or more programming languages, which is outside the coverage of these techniques.

Some detection techniques deal with such an issue in a language-independent manner (i.e., no language-specific components are required) by considering a given source code as text. For example, Flores et al. [18] parse given source code files to 3-gram characters and, therefore, measure the similarity with Cosine correlation. Another example is the work of Brixtel et al. [7], which determines similarity through multiple text granularity levels (starting from the character level to the document level).

To capture a shared semantic across source code files, another work by Flores et al. [17] relies on latent semantic analysis (LSA) [13] in which all of the code files are simply fed to the LSA. This kind of technique was also performed by Ullah et al. [51].

Classification algorithms is also applicable for language-independent techniques in which one instance refers to a source code pair and its target class defines whether the pair is suspicious. Ullah et al. [50] consider source code words as learning features and, therefore, raise the suspicion for some pairs with the help of multinomial logistic regression [6]. Yasaswi, Purini, and Jawahar [56] classify the suspicion through a character-level language model and a support-vector machine [10]. Language-independence can also be applied by encapsulating an existing text-based similarity-detection tool. For instance, Allyson et al. [4] rely on a text-based similarity-detection tool by applying five normalization rules to the source code beforehand. Petrik, Chuda, and Steinmüller utilize a UNIX *diff* command to determine similarity. They firstly compare given source code files in multiple representations; after this, the similarity degree is calculated by averaging the results.

It is true that treating source code as text enables the detection of cross-language plagiarism and collusion with minimal effort. Nevertheless, this treatment may reduce the detection accuracy [44]; the source code can be inaccurately parsed since source code grammars are different from text grammars. For instance, statement *countMAX+=1;* can be considered to be one word according to text grammars since no spaces are involved between the tokens.

To maintain accuracy, several detection techniques tokenize the source code files and convert them to a particular intermediate representation. Arwin and Tahaghoghi [5] generate such a representation with the help of GNU compiler collection. Rabbani and Karnalim [42] utilize .NET Common Intermediate Language as the intermediate representation for source code files written in various .NET programming languages (e.g., C# and Visual Basic).

Converting to a particular intermediate representation can be easy if the conversion tools (e.g., GNU compiler collection and .NET compiler) cover all of the code files' programming languages. However, since most conversion tools cover only a limited number of languages, it is possible that one or more of the used programming languages are not covered. When this occurs, such a conversion should be created from scratch, which can be demanding.

3. Method

Cross-language source code plagiarism and collusion can be detected either by ignoring language-specific features or converting the code files to a particular intermediate representation. The former guarantees the ease of integrating new programming languages but with a trade-off in the accuracy of the token parsing. The latter keeps such accuracy but requires a considerable amount of effort in integrating new programming languages. There is a need to propose a detection technique that can somehow balance these two; it should be easy to integrate new programming languages while keeping the token parsing's accuracy.

This paper proposes a cross-language source code plagiarism- and collusion-detection technique. Unique to this, it is language-dependent (which makes it more accurate in tokenizing source code and detecting similarity) but with limited effort in covering new programming languages. It only requires a source code tokenizer, which can be easily generated with ANTLR [38] and its provided grammars[2]. Figure 1 shows how our proposed technique works in five stages.

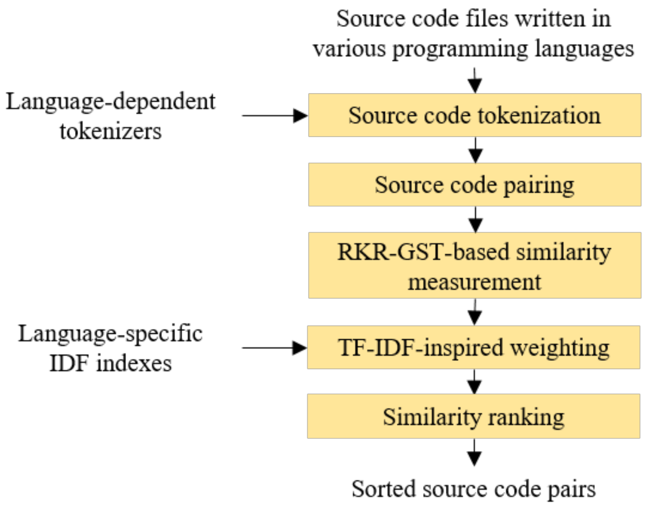

Source code files written in various programming languages
Language-dependent tokenizers
Source code tokenization
Source code pairing
RKR-GST-based similarity measurement
Language-specific IDF indexes
TF-IDF-inspired weighting
Similarity ranking
Sorted source code pairs

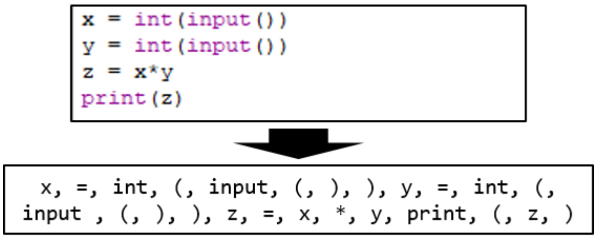
Figure 1. Our proposed cross-language source code plagiarism- and collusion-detection technique. It accepts source code files as its input and, therefore, generates suspected source code pairs through five consecutive stages

At first, the source code files are tokenized with their corresponding tokenizers and then compared directly with the Running Karp-Rabin Greedy String Tiling (RKR-GST) algorithm. Such a direct comparison is based on the assumption that some crucial tokens for raising suspicion (e.g., identifiers, keywords, constants, and arithmetic operators) are represented in the same way across programming languages. The similarity degree is then calculated with the help of a TF-IDF-inspired weighting [13], which prioritizes rare matches to accentuate the impact of non-coincidental similarity. Finally, all source code pairs are sorted based on their similarity degree in descending order and displayed as the result.

[2]https://github.com/antlr/grammars-v4

3.1. Source code tokenization

This stage converts inputted source code files to token strings in which one code file corresponds to one string. At first, it detects the programming language of each code file by checking the file extension. For instance, a code file's language is C# if its file extension is *.cs*. Afterwards, each code file is tokenized with its corresponding language-dependent tokenizer; all non-semantic-preserving tokens such as comments and whitespaces are removed. An example of this sub-stage can be seen in Figure 2. If the file extension is not recognizable, the code file will be excluded from the comparison.



```
x = int(input())
y = int(input())
z = x*y
print(z)
```

```
x, =, int, (, input, (, ), ), y, =, int, (,
input , (, ), ), z, =, x, *, y, print, (, z, )
```

Figure 2. Example of converting Python code file to token string

To mitigate the number of non-recognizable code files, it is important to provide tokenizers for most of the frequently used programming languages. Currently, our detection technique covers ten languages: C, C++, C#, Java, Javascript, Kotlin, Pascal, Python, R, and Scala. Other programming languages can be accommodated with the help of ANTLR [38] and its provided grammars.

We are aware that tokenization can be featured with additional preprocessing steps such as stop words or template code removal. However, these steps are not implemented in our proposed technique; they are commonly language-dependent, and creating them separately per programming language can be demanding. This will contradict the fact that our technique takes the benefits of language dependency with a minimum effort. Furthermore, their generalized benefit (which is mainly for accentuating the impact of non-coincidental matches, assuming these matches rarely occur) can still be obtained with the help of our TF-IDF-inspired weighting (which will be described later).

3.2. Source code pairing

At this stage, each inputted code file is paired with other code files, resulting in $K(K-1)/2$ comparisons (where K refers to the number of inputted code files). In such a manner, all possible pairwise combinations are listed.

The pairing mechanism starts by storing all of the inputted code files in an array. The array is then iterated from the first to second-to-last element; each element will be compared to the other elements that are positioned after that element. The last element is not included in the iteration, as it has no elements following it.

To illustrate this, let us assume that we have three inputted code files (labeled *code1*, *code2*, and *code3*). With our pairing mechanism, these files are initially stored in an array in that order. The iteration starts with *code1*, with the resulting *code1-code2* and *code1-code-3* as comparison pairs. It is then continued to *code2*, which adds *code2-code3* to the comparison pairs. The iteration stops prior to processing the last element, which is *code3*. As a result, three comparison pairs are generated: *code1-code2*, *code1-code-3*, and *code2-code3*.

3.3. RKR-GST-based similarity measurement

For each comparison pair from the second stage, the matched regions will be generated at this stage, involving the token strings obtained from the first stage. These strings are compared to each other using Running Karp-Rabin Greedy String Tiling (RKR-GST) [54]. For sensitivity, RKR-GST's minimum matching length is set to two, which means that all of the matched regions whose lengths are higher than or equal to two will be listed.

As the token strings can come from different programming languages, it is expected that the resulting matches are limited to crucial tokens for raising suspicions like identifiers, keywords, constants, and arithmetic operators. These tokens' representation is seldom changed as the result of programming language conversion.

3.4. TF-IDF-inspired weighting

Suspicion can only be raised when the matches are non-coincidental. Assuming that rare matches are non-coincidental, they are prioritized at this stage by weighting each token occurrence with the token's inverse document frequency (IDF) score, which is proportional to the token's rarity.

The IDF score for each token is calculated as in (1). It divides the total number of code files in collection (N) with the number of code files containing given token (N_t); both N and N_t are incremented by one just to avoid division by zero.

$$IDF(t) = \frac{N+1}{N_t+1} \tag{1}$$

Each programming language will have its corresponding IDF score set that is stored as a language-specific index. The index is a hash map containing a key-score tuple, where "key" refers to the token mnemonic, and "score" refers to the number of code files containing that token.

Per comparison pair, its similarity degree will be calculated as in (2) after each token is weighted with their corresponding language-specific IDF score; *c1* and *c2* are the code files, while M represents the matched regions. It averages the matched proportion of the code files. Matched proportion (mp) for each code file is determined with (3) by summing the IDF score of matched region (c_m) toward code file (c) and then dividing it with the total IDF score of that code file. The IDF score for a particular region (or the whole parts) of the code file can be calculated as in (4). It simply sums up the IDF scores of all tokens covered by that region; each token's IDF score is taken from its respective language-specific IDF index.

$$sim(c1, c2, M) = \frac{mp(c1, M) + mp(c2, M)}{2} \tag{2}$$

$$mp(c, M) = \frac{\sum_{m \in M} idf\_ts(c_m)}{idf\_ts(c)} \tag{3}$$

$$idf\_ts(c) = \sum_{t \in c} idf(t) \tag{4}$$

It is worth noting that this kind of weighting can also be beneficial in detecting conventional source code plagiarism and collusion in which both the original and copied code files are written in the same programming language. The impact of rare matches (which appears to be non-coincidental) will be accentuated.

3.5. Similarity ranking

The comparison pairs are sorted based on their resulting similarity degrees at this stage. Pairs with high similarity degrees are placed at the top, which means that they are quite suspicious.

3.6. Language-specific IDF index generation

TF-IDF-inspired weighting depicts the need for language-specific IDF indexes, which can be generated from either an internal or external dataset. If the former is chosen, the indexes will be generated toward the inputted source code files; this will be performed between the source code tokenization and the pairing. Otherwise, the indexes will be built separately with different source code files; this should be done prior to the execution of our technique.

Language-specific IDF indexes are generated in a threefold manner. First – the given source code files (either from an internal or external dataset) are grouped based on their corresponding programming languages (via their file extensions). Second – each of the files is converted to a token string with its corresponding language-specific tokenizer. Third – per the programming language, distinct tokens are listed, and each is assigned with its IDF score calculated as in (1), where N is the number of code files, and N_t is the number of code files that contain the token.

4. Evaluation

This section describes how the proposed technique was evaluated, which datasets and metrics were used on this evaluation, and what findings could be concluded. The datasets and metrics are explained first, followed by the evaluation methodology and results.

4.1. Evaluation datasets

Three datasets were used in this evaluation. They are denoted as introductory, design pattern, and same-language datasets. The first two are cross-language datasets where the similarity disguises are about rewriting the source code files in other programming languages. In the final one, the similarity disguises are conventional, focusing on lexical, structural, and/or semantic change (but without programming language conversion).

The introductory dataset was created by rewriting seven original code files in a Java dataset [28] adapted from Liang's book [31] to eight other programming languages: C++, C#, Javascript, Kotlin, Pascal, Python, R, and Scala. These code files cover seven introductory materials: the output, input, branching, looping, function, array, and matrix. The dataset results in 63 code files (7 original and 54 copied files); it evaluates the proposed technique in dealing with programming language conversion on introductory courses.

The design pattern dataset was also created by rewriting several Java source code files in other programming languages. Twenty-three design pattern code files from Tutorialspoint[3] were considered as the original code files, which were then translated in four other programming languages: C++, C#, Python, and Kotlin. As a result, the dataset contains 115 code files in total (23 original and 92 copied files); it evaluates the proposed technique in dealing with programming language conversion on advanced courses (as design patterns are seldom used by novice programmers).

The same-language dataset is the Java dataset used for creating the introductory dataset [28]. Each original code file was copied and modified according to the last six levels of similarity disguise taxonomy [16] (the first level was excluded, as it depicted no disguises) by nine teaching assistants. To guarantee that all of the copied code files were at their correct disguise level, any copied code files that violated their corresponding disguise level rules were removed. Per the original code file, the non-copied ones were then created by 15 other teaching assistants. The assistants were asked to solve the same problem tasks but without any access to either the original or copied code files. As a result, each original code file has up to 54 copied code files and 15 non-copied code files, resulting in 467 code files in total. This dataset evaluates the impact of the proposed technique while dealing with conventional similarity disguises.

Similarity disguise taxonomy [16] maps the disguises to seven levels, where each level covers each of its lower level's coverage. Rephrased by Karnalim and Budi [27], these levels are as follows:

- Level 0: Verbatim copy,
- Level 1: Comment and whitespace modification,
- Level 2: Identifier renaming (e.g., variable or function name),
- Level 3: Component declaration relocation (e.g., variable or function),

[3]https://www.tutorialspoint.com/design_pattern/

- Level 4: Method (or function) structure change,
- Level 5: Program statement replacement,
- Level 6: Logic change.

4.2. Evaluation metrics

Two metrics are used in this evaluation: the similarity degree of the suspected pairs, and mean average precision (MAP) [13]. On the similarity degree of the suspected pairs, a detection technique is considered effective if it can enhance the similarity degree of the suspected pairs, as these pairs should be perceived as identical regardless of their disguises. Per the dataset, the suspected pairs are generated by pairing each original code with its copied code files. This results in 56 pairs for the introductory dataset, 92 pairs for the design pattern dataset, and 355 pairs for the same-language dataset.

MAP [13] is an Information Retrieval metric for measuring effectiveness calculated as in (5), where Q are the queries, and *aprec(q)* is the average precision for a particular query *q*. Average precision is measured as in (6); TP are true positives and *prec(tp)* is the precision for the top-M results (in which M is the position of tp). The precision for top-M results is the number of true positives on the top-M positive results normalized by *M*.

$$MAP = \frac{\sum_{q \in Q} aprec(q)}{|Q|} \tag{5}$$

$$aprec(o) = \frac{\sum_{tp \in TP} prec(tp)}{|TP|} \tag{6}$$

In our context, the queries refer to the original code files, while the true positives refer to their copied code files and the positive results refer to the union of their copied and non-copied code files. For each original code file on the introductory and design pattern datasets, the non-copied code files are the copied files of other original code files. As a result, the introductory dataset has 7 original files with 8 copied and 49 non-copied files each. The design pattern dataset has 23 original files; each of them is featured with 4 copied and 88 non-copied files.

We are aware that the effectiveness can be measured with other popular metrics: precision, recall, and f-score; however, these are not included for three reasons. First – MAP covers precision to some extent, as the former is derived from the latter (but with more sensitivity to rank position). Second – recall is not applicable, as our technique involves no minimum threshold for suspicion. Unlike many detection techniques, all of the comparison pairs are listed in descending order based on their similarity degrees, giving more freedom to the examiners for deciding when they need to stop observing pairs from the top of the list. We believe that hypothetically incorporating the minimum threshold specifically for this metric is unnecessary; on most occasions, recall is inversely proportional to precision (or MAP in our case). Third – the f-score is also not applicable since it is the harmonic mean between precision and recall (while recall is not applicable in our context).

4.3. Evaluation methodology

This evaluation aims to answer three research questions:

- R1: Is the proposed technique more effective than common techniques in academia for dealing with programming language-conversion disguises?
- R2: Do external IDF indexes enhance the effectiveness of the proposed technique in dealing with programming language-conversion disguises?
- R3: Is the proposed technique comparable to common techniques in academia for dealing with conventional similarity disguises?

The first research question was addressed by comparing the proposed technique (with internal IDF indexes on board, referred as *idfw*) to common techniques in academia [49] based on the similarity degree of the suspected pairs and MAP. These common techniques in academia were slightly modified to deal with cross-language source code plagiarism and collusion. They work in a similar fashion to our proposed technique except for the fact that TF-IDF-inspired weighting is replaced with either average normalization (whose calculation can be seen in 7) or maximum normalization (whose calculation can be seen in 8); *c1* and *c2* are the numbers of tokens for both code files, and T are the numbers of matched tokens. The common technique with average normalization is referred to as *avgn*, while the counterpart is referred to as *maxn*. In terms of the evaluation datasets, the introductory and design pattern datasets were used, as they exclusively applied programming language-conversion disguises.

$$avgsim = \frac{2 * T}{c1 + c2} \tag{7}$$

$$maxsim = \frac{T}{min(c1, c2)} \tag{8}$$

To address the second research question, the proposed technique was further derived to two sub-techniques. The first one relies on the internal dataset for generating IDF indexes (referred as *idfw*), while the other relies on the external dataset (referred to as *idfw_ext*). These sub-techniques were then compared to each other on the introductory and design pattern datasets, which applied programming language-conversion disguises. This comparison utilized the same metrics as in the first research question: the similarity degree of the suspected pairs and MAP.

Considering the fact that we were only able to build the external dataset for eight programming languages (Java, C++, C#, Javascript, Kotlin, Pascal, Python, and R), only source code files written in these languages were considered while addressing the second research question. However, we do believe that these files are still sufficient to draw some findings, as only the introductory dataset is affected with this reduction (with seven copied files excluded).

Each external dataset was built based on ten GitHub trending projects for that language on February 7, 2019. The projects were selected based on their descending rank order, with projects whose sizes are larger than 20 MB being skipped.

The third research question was addressed in a similar fashion to the first research question. It compared the proposed technique (*idfw*) to common techniques in academia (*avgn* and *maxn*), with the similarity degree of the suspected pairs and MAP as the evaluation metrics. However, the comparison was performed on the same-language dataset instead of the introductory and design pattern datasets; this dataset exclusively applied conventional similarity disguises.

4.4. R1 findings

Figure 3 shows that our proposed technique (*idfw*) generates a higher similarity degree range as compared to common techniques in academia (*avgn* and *maxn*). On average, it is 8.1% higher than *avgn* while outperforming *maxn* by 1.4%. Such a phenomenon also occurs on the design pattern dataset (see Fig. 4). *idfw*'s average score is 7.1% higher than *avgn* and 3.8% higher than *maxn*. This is to be expected, as *idfw* accentuates the impact of rare matches, and some of the copied code files share such rare matches with their original code files.

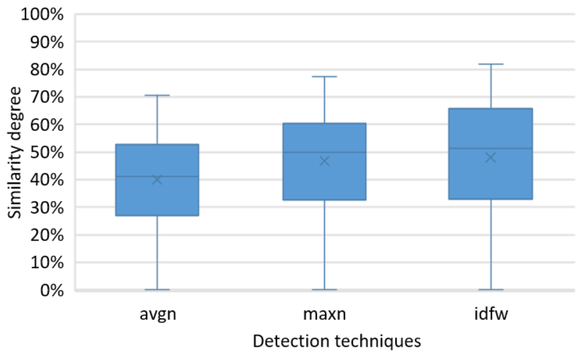


Figure 3. Similarity degree distribution for *avgn*, *maxn*, and *idfw* toward introductory dataset

When the copied code files are grouped together based on their programming language, Figure 5 shows that *idfw* outperforms *avgn* in all programming language categories except Pascal on the introductory dataset. In that language, *idfw*'s averaged similarity degree is 3.8% lower. Further observations show that , Pascal shares fewer crucial tokens with Java (for raising suspicion) as compared to other programming languages. For example, Pascal uses "<>" to check inequality, while Java uses "! =".

Figure 5 shows that, on the introductory dataset, *idfw* is able to outperform *maxn* in five programming languages (C#, Javascript, Python, R, and Scala) while performing worse on the remaining languages. Such an inconsistency is expected, as *maxn* is able to ignore some mismatched tokens once the number of matched tokens is equal to the shorter token string size.

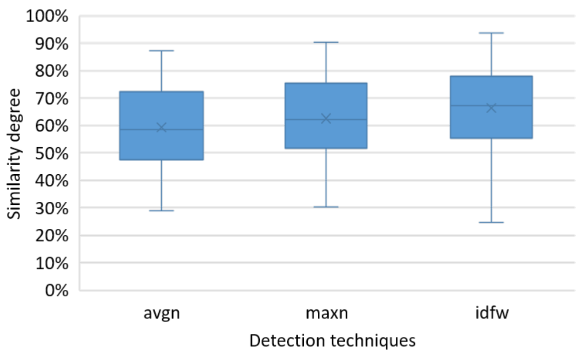


Figure 4. Similarity degree distribution for *avgn*, *maxn*, and *idfw* toward design pattern dataset

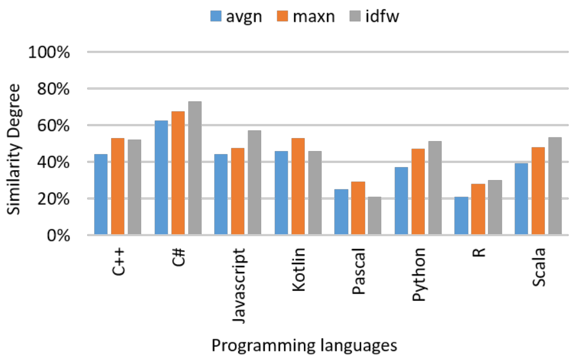


Figure 5. Similarity degree distribution for *avgn*, *maxn*, and *idfw* toward introductory dataset per programming language

On the design pattern dataset, Figure 6 shows that *idfw* outperforms both *avgn* and *maxn* in most programming languages except Kotlin. Kotlin's object-oriented syntax employs additional tokens such as *internal* and *override*. *idfw* considers these new tokens as mismatches since they are Kotlin-specific.

As depicted in Figure 7, *idfw* yields higher MAP than *avgn* on both the introductory and design pattern datasets (by 9.7% and 4.7%, respectively). *idfw* prioritizes rare matches by assigning them with higher IDF scores; such rare matches are frequently found on copied code files.

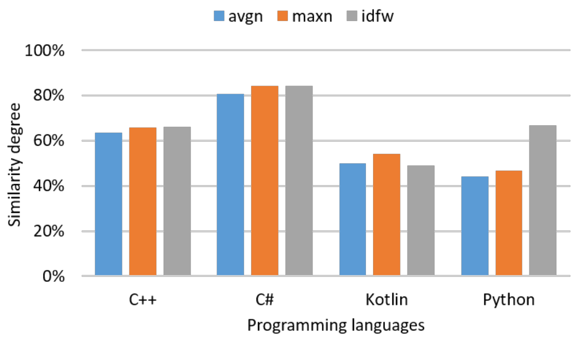


Figure 6. Similarity degree distribution for *avgn*, *maxn*, and *idfw* toward design pattern dataset per programming language

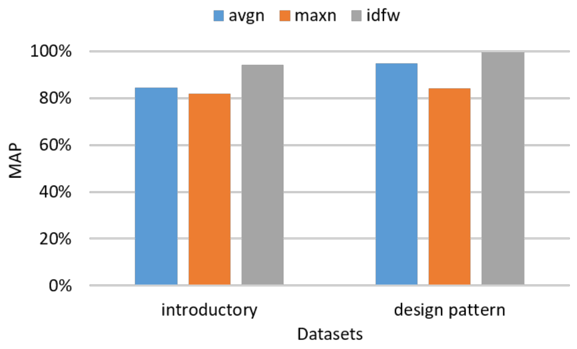


Figure 7. MAP for *avgn*, *maxn*, and *idfw* on introductory and design pattern datasets

When compared to *maxn*, *idfw* is far more effective; its MAP is 12.1% higher on the introductory dataset and 15.4% higher on the design pattern dataset. Both differences are even greater than those involving *avgn*; one possible reason behind this is that *maxn* occasionally excludes some crucial tokens from consideration (for raising suspicion) once the number of matched tokens exceeds the shorter token string size.

4.5. R2 findings

The use of external IDF indexes boosts the similarity degree of suspected source code pairs on the introductory dataset. Figure 8 shows that, on average, our technique with external IDF indexes (*idfw_ext*) outperforms the same technique with internal IDF indexes (*idfw*) by 16.2%. This positive impact also occurs on the design pattern dataset, even though the resulting difference is smaller. According to Figure 9, *idfw_ext* yields an 8.8%-higher similarity degree than *idfw*. External IDF indexes are larger in size as compared to their internal counterparts. Such a large difference therefore increases the weights of identifiers, constants, and arithmetic operators – tokens that are rare and crucial for raising suspicion in both datasets (as language conversion does not affect these tokens).

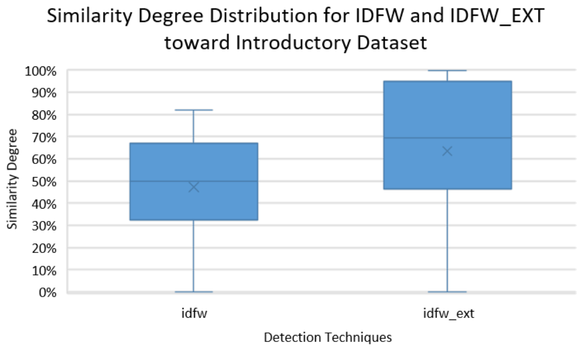


Figure 8. Similarity degree distribution for *idfw* and *idfw_ext* toward introductory dataset

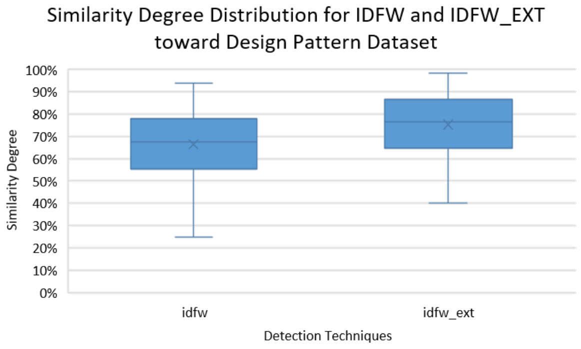


Figure 9. Similarity degree distribution for *idfw* and *idfw_ext* toward design pattern dataset

When the dataset is grouped per programming language, external IDF indexes are still favorable than their internal counterparts (see Figs. 10 and 11). Only Kotlin shows a reduction with such kind of usage on the introductory dataset. Further observation shows that Kotlin's external IDF indexes are built from object-oriented code; this does not suit the introductory dataset's procedural code.

Nevertheless, an increased similarity degree does not always entail higher MAP. As seen in Figure 12, *idfw_ext* yields lower MAP than *idfw*. Its MAP is 1.1% and 0.4% lower on the introductory and design pattern datasets, respectively. A possible reason behind this is that the external indexes share different term-importance values than the evaluation datasets, resulting in inaccurate weight proportion. For example, identifier *counter* may be rare on the evaluation dataset but common on the external indexes.

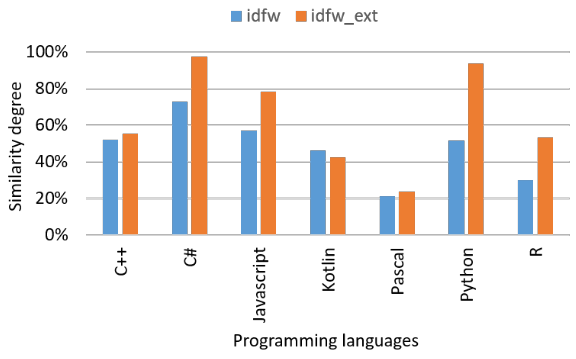


Figure 10. Similarity degree distribution for *idfw* and *idfw_ext* toward introductory dataset per programming language

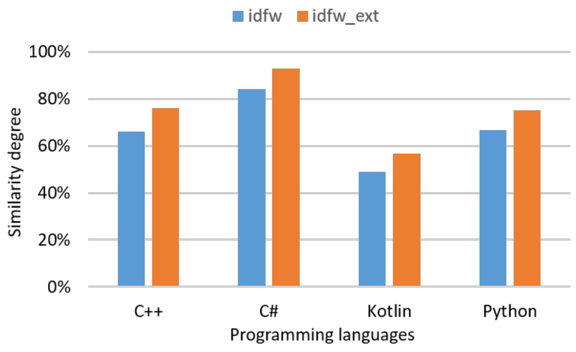


Figure 11. Similarity degree distribution for *idfw* and *idfw_ext* toward design pattern dataset per programming language

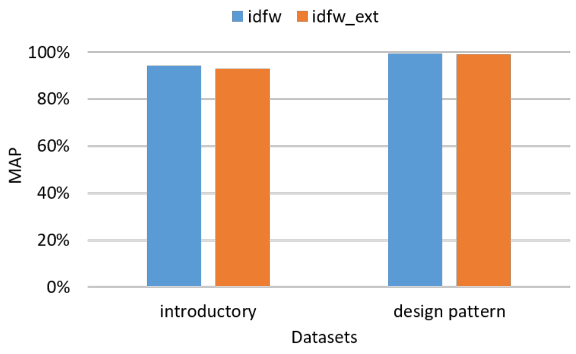


Figure 12. MAP for *idfw* and *idfw_ext* on introductory and design pattern datasets

4.6. R3 findings

In general, our proposed technique (*idfw*) generates a lower similarity degree than common techniques in academia (*avgn* and *maxn*) when dealing with conventional similarity disguises (see Fig. 13). It only leads to a 29.7% averaged similarity degree, while *avgn* and *maxn* generate 81.2 and 86.5%, respectively. This finding is natural since most of the matched tokens are keywords on the same-language dataset; in addition, their impact is mitigated by TF-IDF-inspired weighting due to their frequent occurrences.

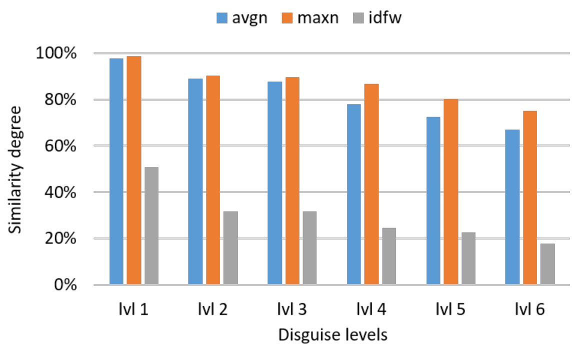


Figure 13. Similarity degree distribution for *avgn*, *maxn*, and *idfw* per plagiarism level toward same-language dataset

Even though *idfw*'s similarity degree is significantly lower than both *avgn* and *maxn*, it still shares the same characteristic with these two. The similarity degree is inversely proportional to the disguise level; higher disguise levels are harder to detect, as they share fewer tokens.

When perceived from a disguise-level perspective, Figure 13 depicts that the greatest difference between *idfw* and *avgn* occurs on Level-2 (which concerns identifier renaming). On both *idfw* and *avgn*, the renamed identifiers are considered to be mismatches toward their originals, as they have different mnemonics. However, *idfw* weighs such mismatches to a greater degree due to their rare occurrences.

Compared to *maxn*, the greatest difference occurs on Level-4 (see Figure 13). This level is about function structure change; most of its implementation is about replacing method calls with their corresponding contents or vice versa. *maxn* is in favor of this since the level leads to higher token size differences between original and plagiarized code; this difference is exclusively ignored by *maxn*.

According to Figure 13, Level-1 is where *idfw* shares the smallest difference toward *avgn* and *maxn*. This is expected, as that level's disguises do not affect the token string. These only disguise the focus on comments and whitespaces, which are automatically discarded at the tokenization stage.

The *idfw*'s low similarity degree does not mean that it is less accurate than *avgn* and *maxn* for dealing with conventional similarity disguises. Figure 14 shows that *idfw* is comparable to *avgn* and *maxn*. On average, it is still more accurate than *avgn* (with a 1.4% difference) even though it is less accurate than *maxn* (with a 1.1% difference).

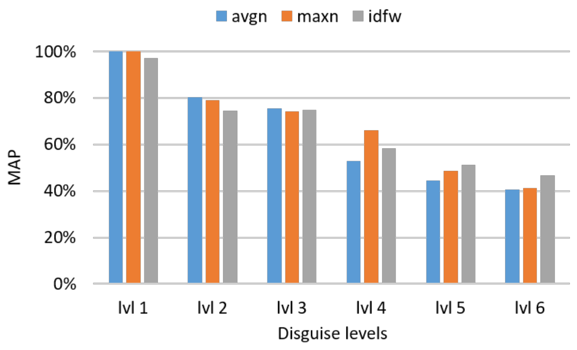


Figure 14. MAP for *avgn*, *maxn*, and *idfw* per plagiarism level toward same-language dataset

On the first disguise level, *idfw* is the only technique that cannot generate 100% MAP. Several copied files share fewer tokens with their originals than non-copied ones, which prevents them from being placed at the top of the results. Level-2 is also a level that does not favor *idfw*, as its disguises focus on identifier renaming. The renamed identifiers are considered to be mismatches and are weighted higher on *idfw* due to their rarity. On other levels, *idfw* commonly outperforms *avgn* and/or *maxn*.

5. Conclusion and future work

This paper proposes a cross-language source code plagiarism- and collusion-detection technique. Despite the fact that the technique is language-dependent, it can accommodate new programming languages with a minimum amount of effort. For each language, it only needs a source code tokenizer; this can be generated with the help of ANTLR [38]. The technique works by applying language-dependent tokenizers, comparing the token strings with RKR-GST in a pairwise manner, and sorting the pairs based on their similarity degrees (calculated with the help of a TF-IDF-inspired weighting).

According to our evaluation, several findings can be concluded. First, for dealing with language-conversion disguises, our proposed technique is more effective than common techniques in academia. Second, the use of external IDF indexes can enhance the resulting similarity degree but not the accuracy (MAP) in dealing with language-conversion disguises. Third, when handling conventional similarity disguises, our proposed technique is comparable to common techniques in terms of MAP, but it generates a lower similarity degree. Fourth, the technique is favorable on advanced disguise levels.

In future work, we plan to handle identifier renaming disguise (which is one of the weaknesses of our technique). All identifiers will be renamed based on their first-occurrence order (as inspired by [25]). Furthermore, we also plan to evaluate a technique on similarity disguises that combines language-conversion and conventional disguises. It is possible that, in addition to converting the code to another programming language, the culprit also applies some other modification.

References

[1] Acampora G., Cosma G.: A fuzzy-based approach to programming language independent source-code plagiarism detection. In: *The 2015 IEEE International Conference on Fuzzy Systems*, pp. 1–8. IEEE, 2015. https://doi.org/10.1109/FUZZ-IEEE.2015.7337935.

[2] Agrawal M., Sharma D.K.: A state of art on source code plagiarism detection. In: *The 2nd International Conference on Next Generation Computing Technologies*, pp. 236–241. IEEE, Dehradun, 2016. https://doi.org/10.1109/NGCT.2016.7877421.

[3] Al-Khanjari Z.A., Fiaidhi J.A., Al-Hinai R.A., Kutti N.S.: PlagDetect: a Java programming plagiarism detection tool, *ACM Inroads*, vol. 1(4), pp. 66–71, 2010. https://doi.org/10.1145/1869746.1869766.

[4] Allyson F.B., Danilo M.L., José S.M., Giovanni B.C.: Sherlock N-Overlap: invasive normalization and overlap coefficient for the similarity analysis between source code, *IEEE Transactions on Computers*, vol. 68, 2018. https://doi.org/10.1109/TC.2018.2881449.

[5] Arwin C., Tahaghoghi S.M.M.: Plagiarism detection across programming languages. In: *The 29th Australasian Computer Science Conference – Volume 48*, pp. 277–286, Australian Computer Society, Hobart, 2006. https://dl.acm.org/citation.cfm?id=1151730.

[6] Böhning D.: Multinomial logistic regression algorithm, *Annals of the Institute of Statistical Mathematics*, vol. 44(1), pp. 197–200, 1992. https://doi.org/10.1007/BF00048682.

[7] Brixtel R., Fontaine M., Lesner B., Bazin C., Robbes R.: Language-independent clone detection applied to plagiarism detection. In: *The 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pp. 77–86. IEEE, Timisoara, 2010. https://doi.org/10.1109/SCAM.2010.19.

[8] Budiman A.E., Karnalim O.: Automated Hints Generation for Investigating Source Code Plagiarism and Identifying The Culprits on In-Class Individual Programming Assessment, *Computers*, vol. 8(1), pp. 1–20, 2019. https://doi.org/10.3390/computers8010011.

[9] Burrows S., Tahaghoghi S.M.M., Zobel J.: Efficient plagiarism detection for large code repositories, *Software: Practice and Experience*, vol. 37(2), pp. 151–175, 2007. https://doi.org/10.1002/spe.750.

[10] Cortes C., Vapnik V.: Support-vector networks, *Machine Learning*, vol. 20(3), pp. 273–297, 1995. https://doi.org/10.1007/BF00994018.

[11] Cosma G., Joy M.: Towards a Definition of source-code plagiarism, *IEEE Transactions on Education*, vol. 51(2), pp. 195–200, 2008. https://doi.org/10.1109/TE.2007.906776.

[12] Cosma G., Joy M.: An approach to source-code plagiarism detection and investigation using Latent Semantic Analysis, *IEEE Transactions on Computers*, vol. 61(3), pp. 379–394, 2012. https://doi.org/10.1109/TC.2011.223.

[13] Croft W.B., Metzler D., Strohman T.: *Search engines: information retrieval in practice*, Addison-Wesley, 2010.

[14] Domin C., Pohl H., Krause M.: Improving plagiarism detection in coding assignments by dynamic removal of common ground. In: *The 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pp. 1173–1179. ACM Press, San Jose, 2016. https://doi.org/10.1145/2851581.2892512.

[15] Engels S., Lakshmanan V., Craig M.: Plagiarism Detection Using Feature-Based Neural Networks. In: *The 38th SIGCSE Technical Symposium on Computer Science Education*, vol. 39, pp. 34–38, ACM Press, 2007. https://doi.org/10.1145/1227504.1227324.

[16] Faidhi J.A.W., Robinson S.K.: An empirical approach for detecting program similarity and plagiarism within a university programming environment, *Computers & Education*, vol. 11(1), pp. 11–19, 1987. https://doi.org/10.1016/0360-1315(87)90042-X.

[17] Flores E., Barrón-Cedeño A., Moreno L., Rosso P.: Cross-language source code re-use detection using Latent Semantic Analysis, *Journal of Universal Computer Science*, vol. 21(13), pp. 1708–1725, 2015. http://www.jucs.org/jucs_21_13/cross_language_source_code.

[18] Flores E., Barrón-Cedeño A., Moreno L., Rosso P.: Uncovering source code reuse in large-scale academic environments, *Computer Applications in Engineering Education*, vol. 23(3), pp. 383–390, 2015. https://doi.org/10.1002/cae.21608.

[19] Fraser R.: Collaboration, collusion and plagiarism in computer science coursework, *Informatics in Education*, vol. 13(2), pp. 179–195, 2014. https://doi.org/10.15388/infedu.2014.01.

[20] Fu D., Xu Y., Yu H., Yang B.: WASTK: a weighted abstract syntax tree kernel method for source code plagiarism detection, *Scientific Programming*, vol. 2017, pp. 1–8, 2017. https://doi.org/10.1155/2017/7809047.

[21] Halak B., El-Hajjar M.: Plagiarism detection and prevention techniques in engineering education. In: *The 11th European Workshop on Microelectronics Education*, pp. 1–3. IEEE, Southampton, 2016. https://doi.org/10.1109/EWME.2016.7496465.

[22] Halstead M.H.: An experimental determination of the "purity" of a trivial algorithm, *ACM SIGMETRICS Performance Evaluation Review*, vol. 2(1), pp. 10–15, 1973. https://doi.org/10.1145/1041606.1041608.

[23] Inoue U., Wada S.: Detecting plagiarisms in elementary programming courses. In: *The 9th International Conference on Fuzzy Systems and Knowledge Discovery*, pp. 2308–2312. IEEE, 2012. https://doi.org/10.1109/FSKD.2012.6234186.

[24] Jadalla A., Elnagar A.: PDE4Java: Plagiarism Detection Engine for Java source code: a clustering approach, *International Journal of Business Intelligence and Data Mining*, vol. 3(2), p. 121, 2008. https://doi.org/10.1504/IJBIDM.2008.020514.

[25] Karnalim O.: A Low-Level Structure-Based Approach for Detecting Source Code Plagiarism, *IAENG International Journal of Computer Science*, vol. 44(4), pp. 501–522, 2017. http://www.iaeng.org/IJCS/issues_v44/issue_4/IJCS_44_4_11.pdf.

[26] Karnalim O.: Source code plagiarism detection with low-level structural representation and information retrieval, *International Journal of Computers and Applications*, 2019. https://doi.org/10.1080/1206212X.2019.1589944.

[27] Karnalim O., Budi S.: The effectiveness of low-level structure-based approach toward source code plagiarism level taxonomy. In: *The 6th International Conference on Information and Communication Technology*, pp. 130–134. IEEE, Bandung, 2018. https://doi.org/10.1109/ICoICT.2018.8528768.

[28] Karnalim O., Budi S., Toba H., Joy M.: Source code plagiarism detection in academia with information retrieval: dataset and the observation, *Informatics in Education*, vol. 18(2), pp. 321–344, 2019. https://doi.org/10.15388/infedu.2019.15.

[29] Kermek D., Novak M.: Process model improvement for source code plagiarism detection in student programming assignments, *Informatics in Education*, vol. 15(1), pp. 103–126, 2016. https://doi.org/10.15388/infedu.2016.06.

[30] Kikuchi H., Goto T., Wakatsuki M., Nishino T.: A source code plagiarism detecting method using alignment with abstract syntax tree elements. In: *The 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pp. 1–6. IEEE, Las Vegas, 2014. https://doi.org/10.1109/SNPD.2014.6888733.

[31] Liang Y.D.: *Introduction to Java programming, comprehensive version (9th Edition)*, Pearson, 2013.

[32] Liu C., Chen C., Han J., Yu P.S.: GPLAG: detection of software plagiarism by program dependence graph analysis. In: *The 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 872–881, ACM Press, Philadelphia, 2006. https://doi.org/10.1145/1150402.1150522.

[33] Malabarba S., Devanbu P., Stearns A.: MoHCA-Java: a tool for C++ to Java conversion support. In: *The 21st international conference on Software engineering*, pp. 650–653, ACM Press, Los Angeles, 1999. https://doi.org/10.1145/302405.302918.

[34] Maletic J.I., Collard M.L.: Exploration, analysis, and manipulation of source code using srcML. In: *The 37th International Conference on Software Engineering*, pp. 951–952, ACM, Florence, 2015. https://dl.acm.org/citation.cfm?id=2819225.

[35] Mišić M.J., Protić Ž.J., Tomašević M.V.: Improving source code plagiarism detection: lessons learned. In: *The 25th Telecommunication Forum*, pp. 1–8, IEEE, Belgrade, 2017. https://doi.org/10.1109/TELFOR.2017.8249481.

[36] Ohmann T., Rahal I.: Efficient clustering-based source code plagiarism detection using PIY, *Knowledge and Information Systems*, vol. 43(2), pp. 445–472, 2015. https://doi.org/10.1007/s10115-014-0742-2.

[37] Ottenstein K.J.: An algorithmic approach to the detection and prevention of plagiarism, *ACM SIGCSE Bulletin*, vol. 8(4), pp. 30–41, 1976. https://doi.org/10.1145/382222.382462.

[38] Parr T.: *The definitive ANTLR 4 reference*, Pragmatic Bookshelf, 2013.

[39] Piñeiro C., Abuin J.M., Pichel J.C.: Perldoop2: A big data-oriented source-to-source Perl-Java compiler. In: *The 15th International Conference on Dependable, Autonomic and Secure Computing*, pp. 933–940, IEEE, Orlando, 2017. https://doi.org/10.1109/DASC-PICom-DataCom-CyberSciTec.2017.156.

[40] Poon J.Y.H., Sugiyama K., Tan Y.F., Kan M.Y.: Instructor-centric source code plagiarism detection and plagiarism corpus. In: *The 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, pp. 122–127, ACM Press, Haifa, 2012. https://doi.org/10.1145/2325296.2325328.

[41] Prechelt L., Malpohl G., Philippsen M.: Finding Plagiarisms among a Set of Programs with JPlag, *Journal of Universal Computer Science*, vol. 8(11), pp. 1016–1038, 2002. http://dx.doi.org/10.3217/jucs-008-11-1016.

[42] Rabbani F.S., Karnalim O.: Detecting source code plagiarism on .NET programming languages using low-level representation and adaptive local alignment, *Journal of Information and Organizational Sciences*, vol. 41(1), pp. 105–123, 2017. https://doi.org/10.31341/jios.41.1.7.

[43] Ragkhitwetsagul C., Krinke J., Clark D.: Similarity of source code in the presence of pervasive modifications. In: *The 16th International Working Conference on Source Code Analysis and Manipulation*, pp. 117–126, IEEE, Raleigh, 2016. https://doi.org/10.1109/SCAM.2016.13.

[44] Ragkhitwetsagul C., Krinke J., Clark D.: A comparison of code similarity analysers, *Empirical Software Engineering*, vol. 23(4), pp. 2464–2519, 2018. https://doi.org/10.1007/s10664-017-9564-7.

[45] Rosales F., García A., Rodríguez S., Pedraza J.L., Méndez R., Nieto M.M.: Detection of plagiarism in programming assignments, *IEEE Transactions on Education*, vol. 51(2), pp. 174–183, 2008. https://doi.org/10.1109/TE.2007.906778.

[46] Sidorov G., Ibarra Romero M., Markov I., Guzman-Cabrera R., Chanona--Hernández L., Velásquez F.: Measuring similarity between Karel programs using character and word n-grams, *Programming and Computer Software*, vol. 43(1), pp. 47–50, 2017. https://doi.org/10.1134/S0361768817010066.

[47] Simon, Cook B., Sheard J., Carbone A., Johnson C.: Academic integrity: differences between computing assessments and essays. In: *The 13th Koli Calling International Conference on Computing Education Research*, pp. 23–32, ACM Press, Koli, 2013. https://doi.org/10.1145/2526968.2526971.

[48] Song H.J., Park S.B., Park S.Y.: Computation of program source code similarity by composition of parse tree and call graph, *Mathematical Problems in Engineering*, vol. 2015, pp. 1–12, 2015. https://doi.org/10.1155/2015/429807.

[49] Sulistiani L., Karnalim O.: ES-Plag: efficient and sensitive source code plagiarism detection tool for academic environment, *Computer Applications in Engineering Education*, vol. 27(1), pp. 166–182, 2019. https://doi.org/10.1002/cae.22066.

[50] Ullah F., Wang J., Farhan M., Habib M., Khalid S.: Software plagiarism detection in multiprogramming languages using machine learning approach, *Concurrency and Computation: Practice and Experience*, p. e5000, 2018. https://doi.org/10.1002/cpe.5000.

[51] Ullah F., Wang J., Farhan M., Jabbar S., Wu Z., Khalid S.: Plagiarism detection in students' programming assignments based on semantics: multimedia e-learning based smart assessment methodology, *Multimedia Tools and Applications*, 2018. https://doi.org/10.1007/s11042-018-5827-6.

[52] Verco K.L., Wise M.J.: Software for detecting suspected plagiarism: comparing structure and attribute-counting systems. In: *The 1st Australasian Conference on Computer Science Education*, pp. 81–88, ACM Press, Sydney, 1996. https://doi.org/10.1145/369585.369598.

[53] Wang L., Jiang L., Qin G.: A search of verilog code plagiarism detection method. In: *The 13th International Conference on Computer Science & Education*, pp. 1–5, IEEE, Colombo, 2018. https://doi.org/10.1109/ICCSE.2018.8468817.

[54] Wise M.J.: Yap3: improved detection of similarities in computer program and other texts. In: *The 27th SIGCSE Technical Symposium on Computer Science Education*, vol. 28, pp. 130–134, ACM Press, Philadelphia, 1996. https://doi.org/10.1145/236452.236525.

[55] Yang F.P., Jiau H.C., Ssu K.F.: Beyond plagiarism: an active learning method to analyze causes behind code-similarity, *Computers & Education*, vol. 70, pp. 161–172, 2014. https://doi.org/10.1016/J.COMPEDU.2013.08.005.

[56] Yasaswi J., Purini S., Jawahar C.V.: Plagiarism detection in programming assignments using deep features. In: *The 4th IAPR Asian Conference on Pattern Recognition*, pp. 652–657, IEEE, Nanjing, 2017. https://doi.org/10.1109/ACPR.2017.146.

Affiliations

Oscar Karnalim
Maranatha Christian University, Surya Sumantri Street No. 65, Bandung, 40164, Indonesia, oscar.karnalim@it.maranatha.edu, ORCID ID: https://orcid.org/0000-0003-4930-6249

Received: 01.08.2019
Revised: 05.11.2019
Accepted: 07.11.2019