UDC 004.415.3
Original Scientific Paper

Detecting Source Code Plagiarism on .NET Programming Languages using Low-level Representation and Adaptive Local Alignment

Faqih Salban Rabbani *faqih.salban@gmail.com*
Faculty of Information Technology
Maranatha Christian University, Indonesia

Oscar Karnalim *oscar.karnalim@it.maranatha.edu*
Faculty of Information Technology
Maranatha Christian University, Indonesia

Abstract

Even though there are various source code plagiarism detection approaches, only a few works which are focused on low-level representation for deducting similarity. Most of them are only focused on lexical token sequence extracted from source code. In our point of view, low-level representation is more beneficial than lexical token since its form is more compact than the source code itself. It only considers semantic-preserving instructions and ignores many source code delimiter tokens. This paper proposes a source code plagiarism detection which rely on low-level representation. For a case study, we focus our work on .NET programming languages with Common Intermediate Language as its low-level representation. In addition, we also incorporate Adaptive Local Alignment for detecting similarity. According to Lim et al, this algorithm outperforms code similarity state-of-the-art algorithm (i.e. Greedy String Tiling) in term of effectiveness. According to our evaluation which involves various plagiarism attacks, our approach is more effective and efficient when compared with standard lexical-token approach.

Keywords: source code plagiarism detection, source code similarity, low-level language, .NET programming language, adaptive local alignment

1. Introduction

Source code plagiarism is a major issue which emerges in Programming course [1]. Students can easily obtain their colleague's work, modify it, and then submit it as their work in no time. Even though plagiarism can be conducted easily, detecting this illegal behavior is not a trivial task. Plagiarism can be conducted by every student, even the smartest one who knows programming concept well [2, 3]. Thus, plagiarism attacks incorporated on student assignments may be varied. It may be the simplest one such as verbatim copy (conducted by weak student) or the most complicated one which incorporates various logic and structure changes (conducted by smart students). Detecting these plagiarism variants may require considerable effort and time since lecturer should enlist all possible plagiarism attacks and check each possible source code pair. However, since the plagiarists still should be penalized, automatic source code plagiarism detection is developed to extenuate lecturer's work. Lecturers are not required to check each source code manually. Instead, they can feed all student source codes into the system and take its result as a consideration for detecting plagiarism.

In general, there are three major approaches for detecting source code plagiarism: text-based, attribute-based, and structure-based approach [4, 5]. Text-based approach determines similarity by considering source code as a raw text; attribute-based approach determines similarity based on source code attributes (e.g. number of operator and operand); and structure-based approach determines similarity based on source code structure. Among these approaches, structure-based approach is the most popular one due to its effectiveness [6]. This approach is quite sensitive to instruction order which is important on determining source code plagiarism.

This paper proposes a structure-based source code plagiarism detection which is focused on .NET programming language. All .NET source codes are converted into .NET Common Intermediate Language (CIL) and their similarity is determined based on Adaptive Local Alignment (ALA) algorithm. .NET CIL is a compiled form of all .NET source codes which is generated after compilation phase. This form is more compact than the source code itself since it only contains semantic-preserving information. On the other hand, ALA is incorporated as our similarity algorithm since it outperforms Greedy String Tiling (GST) algorithm in terms of sensitivity and specificity [7]. GST algorithm is a state-of-the-art algorithm which had been frequently incorporated on various structure-based plagiarism detection approaches [8, 9, 10, 3]. To our knowledge, there are no related works which combine both .NET CIL and adaptive local alignment for detecting source code plagiarism. Beside proposing a source code plagiarism detection approach, we also evaluate the impact of our approach when compared with standard token-based approach in terms of effectiveness and efficiency. This evaluation is conducted based on various plagiarism attacks from Karnalim's dataset [3] and evaluated based on both single and multiple attack schemas.

2. Related Works

Plagiarism is an act for reusing other people's work without explicitly acknowledging the author beforehand [9, 11, 12]. In Computer Programming education, it becomes more serious problem since most source codes are written electronically and most students understand a lot about copy-and-paste technique [1]. Students can easily replicate and plagiarize other student's work in a no time. In addition, even though plagiarism can be detected manually, it is not recommended to be conducted due to its considerable effort. Programming assignments are usually given every week and each of them consists of at least a dozens of source codes [9]. According on these reasons, automatic plagiarism detection is highly desirable to be developed on this domain.

Based on Sraka & Kaucic's work [2], there are various reasons for conducting source code plagiarism. The most common reason is the incapability for solving the given task by themselves. They plagiarize other people's work since they are weak in programming and have low computational thinking. However, this finding does not mean that smart students are never involved on this illegal behavior. Some smart students may plagiarize based on two reasons: 1) They do not have sufficient time to do it; and 2) They are too lazy to do it by themselves. Therefore, detecting source code plagiarism is not a trivial task. Plagiarism attacks may vary from the verbatim copy into the most complicated one such as logic modification and code encapsulation. Most smart students incorporate high-level plagiarism attack since they understand a lot about program structure and semantic.

In general, there are three major approaches for detecting source code plagiarism. These approaches are text-based, attribute-based, and structure-based approach [4, 5]. Text-based approach is the only approach which is programming-independent since it treats source code as raw text. The work of Heintze [13], Brixtel et al [14], Hoad & Zobel [15], and Cosma & Joy [16] are several works which fall into this category. Heintze [13] incorporates fingerprinting method for detecting plagiarism. Each involved source code is translated into a compact collection of integers (i.e. fingerprint) and two source codes are recognized as plagiarized to each other *iff* both fingerprints are similar. Brixtel et al [14] incorporates multiple level plagiarism recognition which vary from character to corpus level. These multiple recognitions are incorporated to keep its sensitivity toward various plagiarism attacks. Hoad & Zobel [15] and Cosma & Joy [16] incorporate information retrieval approach for detecting plagiarism. Hoad & Zobel [15] incorporates ranking approach which is based on information retrieval concept [17]. The most similar source code is expected to have the highest score when its plagiarized source code is given as a query. On the other hand, Cosma & Joy [16] incorporates Latent Semantic Analysis (LSA) to find source code similarity. Source codes are treated as natural language documents and fed into LSA to deduct their similarity. Nevertheless, most text-based approaches ignore programming language structure and semantic which are important for detecting high-level plagiarism attacks.

Attribute-based approach considers two source codes are plagiarized to each other *iff* both source codes yield similar key properties. Some earlier work with this approach are conducted by Donaldson et al [18], Halsthead [19], and Halsthead [20]. Based on their work, four key properties are incorporated for measuring similarity. These properties are the number of unique operators, the number of unique operands, the total number of operators, and the total number of operands. However, since these properties do not adequately represent the source code itself, several further works incorporate additional properties such as the number of variables, methods, loops, conditional statements, and method invocations [6]. Several works are also focused on programmer style properties such as the position of the brackets and comments [21, 22]. When incorporating programmer style, plagiarism is detected *iff* programmer style properties on the current submission are extremely different with previously submitted assignments.

In order to compare the similarity between given key properties, several attribute-based approaches do not rely on exact-match similarity measurement. Instead, they incorporate more sophisticated mechanisms such as machine learning [21, 22, 23, 24, 25] and information retrieval approach [26]. Bandara & Wijayarathna [21] combines three classification algorithm to detect plagiarism: Naïve Bayes, K-Nearest Neighbor (KNN), and AdaBoost Meta-learning algorithm. Source code key properties are fed into combined algorithm for detecting plagiarism and a source code is considered as a plagiarized code *iff* its classification result yields different author. For example, a source code submitted by *A* is considered as a plagiarized code since it is recognized as *B*'s source code based on classification result. Other classification approaches [22, 23, 24] also incorporate similar plagiarism detection mechanism with Bandara & Wijayarathna [21]. Yet, they differ on incorporated classification algorithm and involved key properties. Lange & Mancoridis [22] incorporates KNN and genetic algorithm; Ohno & Murao [23] incorporates coding-style statistics; and Engels et al [24] incorporates feature-based neural networks. Based on the fact that classification requires a considerable amount of training dataset, Jadalla & Elnagar [25] and Ramirez-de-la-Cruz [26] incorporates other mechanisms instead of classification. Jadalla & Elnagar incorporates clustering approach wherein plagiarized source codes are clustered as a single cluster based on their respective key properties. Whereas, Ramirez-de-la-Cruz incorporates cosine similarity from Information Retrieval approach to measure source code similarity based on high-level features.

Even though attribute-based approach yields more accurate result than text-based approach, it is important to note that two different source codes may yield similar key properties. For example, source code of bubble and insertion sort may yield similar number of loops (i.e. two loops) even though both of them are different in terms of how to solve the problem. In addition, as student computational thinking is developed during the course, their programming style may be changed significantly over time. Thus, programming-style-based properties may not reliable enough since it may cause a tremendous number of true positives. Some students may be detected as plagiarists since they change their programming style on their own.

Structure-based approach is the most sophisticated approach among three major approaches for detecting source code plagiarism [6]. It is relatively robust to various plagiarism attacks since this approach is typically focused on the ordinal structure of the given source code. This finding is also strengthened by the fact that structure-based approach is frequently used on publicly available plagiarism detection systems such as JPlag [8], MOSS [27], Sim [28], Plaque [29], YAP [30], Plaggie [31], FPDS [32], and Marble [33]. In general, most structure-based approaches detect plagiarism by converting source codes into lexical token sequences and compare them based on a particular similarity algorithm. This pattern is typically popular since it is frequently incorporated on most publicly available plagiarism detection systems. It is also incorporated on many related research works about source code plagiarism detection such as Lim et al [34], Shah et al [35], Kustanto & Liem [9], Ji et al [36], Djuric & Gasevic [6], and Pawelczak [37]. Nevertheless, not all structure-based approaches rely on standard lexical token sequence. Several works incorporate unique yet effective mechanism to exploit source code structure further.

Several works incorporate additional preprocessing to generate more declarative lexical token sequence [38, 39, 40]. Chilowics et al [38] incorporates function factorization when generating lexical token sequence. All function calls are replaced with their respective function contents before compared. In such manner, their approach is resistant from encapsulation-based attacks such as method inlining and outlining. Ellis & Anderson [39] replaces lexical token sequence with inorder form of generated parse tree. Each source code is converted into a parse tree and its contents are translated into token sequence by applying inorder traversal. Chilowicz et al [40] also incorporates parse-tree approach. Yet, their work generates token sequence based on fingerprinting mechanism instead of inorder traversal.

Instead of incorporating lexical token sequence, several works incorporate low-level codes as a resource for detecting plagiarism. Source codes are translated into their respective low-level form and compared using a particular similarity algorithm. There are several reasons why low-level codes is more effective than standard lexical token sequence for detecting source code plagiarism [3]: 1) Low-level codes only contain semantic-preserving instructions since these codes had been optimized at compilation phase. In other words, similarity is purely determined based on semantic structure; 2) Most syntactic sugars are translated into their natural semantic. For example, *for* and *while* traversals are translated into similar *goto* sequences. This mechanism may prevent syntactic-sugar-based plagiarism attacks automatically; 3) All comments are completely removed so that plagiarism attacks which relies on comment modification are unavailing; and 4) Local variable identifiers are technically renamed based on their respective order. This mechanism may prevent all identifier-renaming plagiarism attacks.

In general, there are four works which incorporates low-level code for detecting source code plagiarism: the work of Ji et al [36], Karnalim [3], Juričić [41], and Juričić et al [10]. The first two works are focused on Java programming language whereas the other two are focused on .NET programming language. Ji et al approach [36] is quite similar with standard lexical-token-sequence approach except that it replaces token sequence with bytecode sequence (bytecode is a low-level representation of Java programming language). Their work is extended by Karnalim [3] by incorporating several additional mechanisms such as method inlining, recursive handling, instruction generalization, and instruction interpretation. On the other hand, Juričić and Juričić et al approach [41, 10] detect source code similarity based on .NET Common Intermediate Language (CIL) which is the low-level representation of most .NET programming languages. Two source codes are considered as plagiarized to each other *iff* their CIL sequences are similar. The only difference between Juričić's and Juričić et al's work is their incorporated similarity algorithm. Juričić incorporates Levensthein distance whereas Juričić et al incorporates Greedy String Tiling (GST) algorithm.

In this paper, we propose two major contributions which are: 1) we extend Juričić et al approach by replacing GST with Adaptive Local Alignment. According to Lim et al [34], this algorithm outperforms GST in terms of sensitivity and specificity; and 2) we evaluate the effectiveness and efficiency of our approach for handling various plagiarism attacks in more detailed manner. We evaluate its impact per plagiarism attack where each plagiarism case only consists of one plagiarism attack. As in previous work about CIL-based plagiarism detection [10, 41], all evaluations are conducted based on generic dataset where each case may contain one or more plagiarism attacks. They do not evaluate its impact when handling various plagiarism attacks separately (i.e. each case only consists of one plagiarism attack). In addition, plagiarism attacks incorporated in previous works are not explicitly enlisted in terms of its attack type and occurrence frequency per case. Thus, we cannot know which plagiarism attacks are best handled by CIL approach.

3. Methodology

When comparing source codes for detecting plagiarism, our approach adapts pairwise approach where each source code is paired with all other source codes to measure its similarity. In general, similarity value for each source code pair is measured by following flowchart given on Figure 1. It is extended from Juričić et al's work [10] by incorporating different similarity algorithm and a particular mechanism to handle uncompilable source codes. As seen on the given flowchart, our source code similarity measurement only works on .NET source codes (e.g. C# and Visual Basic) since we rely on .NET CIL sequence. At the first phase, both source codes will be compiled into their respective .NET CIL executable file. Afterwards, these executable files are fed into .NET CIL disassembler to generate readable text that consist of .NET CIL information. Then, CIL sequences from both readable texts are filtered and their similarity is measured using Adaptive Local Alignment. Nevertheless, since not all source codes on student assignments are compilable, our approach also incorporates lexical tokens for handling the uncompilable ones. When at least one of the given source code is uncompilable, source code similarity will be measured based on standard lexical token sequence. In our work, standard lexical token sequence is generated by ANTLR [42] and grammar listed on ANTLR GitHub repository [43].

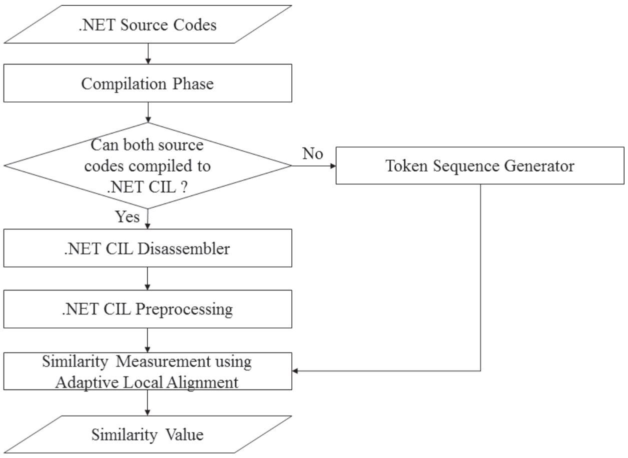
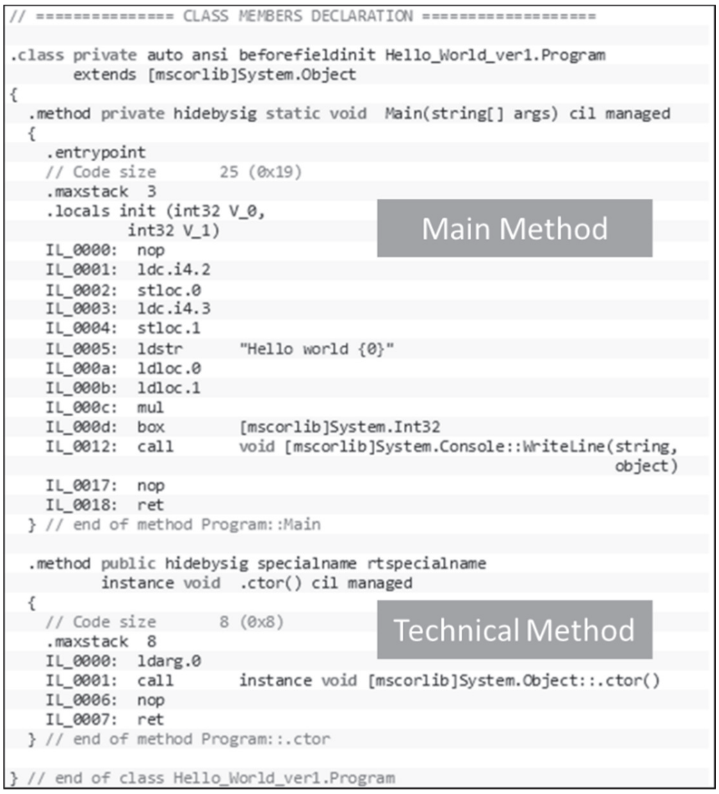


Figure 1. Flowchart for Determining Source Code Similarity

3.1. Compilation Phase and .NET CIL Disassembler

Compilation phase is conducted programmatically so that lecturer is not required to open each student source code on IDE and compile them manually. Lecturer is only required to provide student source codes and let our proposed approach to do the rest. After compiled, each source code is converted into .NET CIL executable form. Even though this form is our target representation for detecting plagiarism, extracting .NET CIL sequence from its executable form directly requires a considerable effort since it is represented as file binaries. Thus, ILDASM [44] is incorporated as a parser to read and extract .NET CIL sequence. ILDASM is a disassembler tool included in Visual Studio or Framework SDK which is able to read .NET CIL information of given executable file and store it on a readable text [41].

By incorporating ILDASM, each executable file is converted into readable text which contains .NET CIL information. The example of readable text generated by ILDASM can be seen on Figure 2. It is generated based on Hello World program written in C#. As seen on Figure 2, Hello World program generates two methods when compiled into CIL. The first one represents the original main method whereas the latter one represents technical method. Technical method is an auto-generated method which is required for executing CIL-based program.



```
// ============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit Hello_World_ver1.Program
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  Main(string[] args) cil managed
  {
    .entrypoint
    // Code size       25 (0x19)
    .maxstack  3
    .locals init (int32 V_0,
             int32 V_1)
    IL_0000:  nop
    IL_0001:  ldc.i4.2
    IL_0002:  stloc.0
    IL_0003:  ldc.i4.3
    IL_0004:  stloc.1
    IL_0005:  ldstr      "Hello world {0}"
    IL_000a:  ldloc.0
    IL_000b:  ldloc.1
    IL_000c:  mul
    IL_000d:  box        [mscorlib]System.Int32
    IL_0012:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                  object)
    IL_0017:  nop
    IL_0018:  ret
  } // end of method Program::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       8 (0x8)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  nop
    IL_0007:  ret
  } // end of method Program::.ctor

} // end of class Hello_World_ver1.Program
```

Figure 2. Readable Text Generated by ILDASM for Hello World Program

3.2. .NET CIL Preprocessing

As seen in Figure 2, readable text generated by ILDASM contains various information about the given source code. However, since our approach only relies on CIL sequence, all unrelated information is discarded. We only extract CIL lines which are usually started with IL as its prefix and merges them as a big chunk of CIL sequence. In other words, all instructions from various methods and classes are merged into one CIL sequence. We do not apply semantic-preserving preprocessing which is suggested by Ji et al [36] and Karnalim [3] since it may require additional processing time. In addition, most semantic-preserving preprocessing phases rely on instruction-specific features and file structure. These aspects may be changed or updated on the further release of .NET CIL.

Each CIL line generally consists of three components which are instruction position, mnemonic, and attributes. Instruction position represent CIL position on its method container. It typically starts with 0 for each method and each instruction may take more than 1 instruction slot. As seen in technical method on Figure 2, *IL* instruction starts with 0 and instruction *IL_0001* takes 5 instruction slots since its successor starts from 6 (*IL_0006*). *IL_0001* represents method call so that it takes more slots to store its method information and parameter invocation. Nevertheless, not all instruction position is marked with non-negative integers. Instead, several instructions are marked with alphabet to represent sub-instruction. As seen in main method on Figure 2, several instructions after *IL_0005* are marked with alphabet (*IL_000a*, *IL_000b*, *IL_000c*, and *IL_000d*).

Mnemonic represents instruction semantic where each mnemonic may require zero or more attributes to conduct its functionality. These attributes are typically placed after each mnemonic on CIL line. For example, *ldc* on *IL_0003* from main method has two attributes which are *.i4* and *.3*. *ldc* represents pushing a value into runtime stack; *.i4* represents 32-bit integer as its pushed value type; and *.3* represents the value of pushed 32-bit integer is 3. Nevertheless, it is important to note that not all instruction attributes start with dot mark. As seen in IL_0005 from main method, it incorporates string as its attributes. The complete list of CIL instructions and their respective semantic can be seen on [45].

After CIL lines are extracted, we filter each CIL line by only considering its mnemonic and return it as our preprocessed result. For example, if readable text on Figure 2 is fed into our preprocessing phase, its result will be *nop, ldc, stloc, ldc, stloc, ldstr, ldloc, ldoloc, mul, box, call, nop, ret, ldarg, call, nop,* and *ret*. Instruction position and attributes are excluded based on following reasons: 1) Instruction order has been implicitly incorporated on generated CIL sequence as sequence order. Therefore, instruction position is unnecessary; 2) Instruction attributes are over-technical and incorporating them on CIL sequence may yield over-sensitive plagiarism detection; and 3) Excluding instruction attributes automatically generalizes data type usage so that it can handle data-type-based plagiarism attacks.

3.3. Similarity Measurement using Adaptive Local Alignment

The similarity between two CIL sequences are measured using Adaptive Local Alignment proposed by Lim et al [34]. This algorithm is extended from Local Alignment [7] by incorporating token frequencies for weighting each token. Each token is assigned with its inverse frequency so that token with low frequency is assigned with high score and token with high frequency is assigned with low score. In such manner, when two token sequences share rare tokens, its similarity score will be increased significantly. We believe that this weighting mechanism is inspired from real-case plagiarism detection. If two source codes incorporate similar "unique" pattern, then their chance to be detected as a plagiarism case will be higher.

Lim et al only incorporates keywords and operators on their token sequence. From our perspective, we believe that identifiers are excluded from their approach since identifier renaming is frequently popular among plagiarists and incorporating them for detecting plagiarism may yield over-sensitive result. However, according to the fact that all identifiers, at some extent, are renamed on CIL [45], identifier renaming is unavailing when handled with our approach. Thus, our work incorporates identifiers in addition to keywords and operators on token sequence for detecting plagiarism. We believe that identifiers may strengthen the sensitivity of our approach for detecting plagiarism.

All setting parameters required for Adaptive Local Alignment are adapted from Lim et al's work [34]. Matched token is scored by its respective inverse frequency; mismatched token and gap are scored by multiplying its respective inverse frequency with -1; α is assigned as 0.6; and β is assigned as 0.4. In addition, to normalize resulted similarity value, we incorporate minimum matching similarity which detail can be seen in (1). *sim(A,B)* represents normalized similarity value where *A* and *B* are compared CIL sequences. It is calculated by dividing total weight of shared subsequence (i.e. *sim_weight(A,B)*) with minimum size from both CIL sequences (i.e. *min(weight(A),weight(B))*). *weight(A)* and *weight(B)* represent the total weight of *A* and *B* respectively.

$$sim(A,B) = \frac{sim\_weight(A,B)}{min(weight(A),weight(B))} \quad (1)$$

4. Evaluation

4.1. Evaluating Our Approach toward Various Plagiarism Attacks Separately

This evaluation is conducted to determine the strengths and weaknesses about our CIL approach when handling various plagiarism attacks separately. This evaluation is conducted based on Karnalim's dataset [3] which is rewritten in C#. Karnalim's dataset is selected for our evaluation since it enlists possible plagiarism attacks in general. These plagiarism attacks are generated based on lecturer assistants who have a lot of experience for handling plagiarism attempts. The detail of how these plagiarism attacks are collected can be read in [3]. However, instead of taking the whole dataset, we only take plagiarism cases which level is 3 or higher based on Faidhi & Robinson specification [46]. Level 0 to 2.5 are completely excluded since we intend to measure the impact of our approach only on plagiarism attacks that affect low-level form (CIL representation). Level 0 to 2.5 attacks do not affect CIL since their attacks are automatically removed and handled by the compiler when generating CIL. In short, our evaluation dataset consists of 42 plagiarism cases from Karnalim's dataset where each plagiarism case is enlisted as level 3 or higher plagiarism attack based on Faidhi & Robinson specification [46].

In this evaluation, standard lexical-token approach is incorporated as a baseline for measuring the improvement of our approach. It is selected as our baseline due to its popularity on various plagiarism detection works. However, to make it comparable with our CIL approach, we incorporate similar measurement algorithm (i.e. Adaptive Local Alignment) as its similarity measurement. For convenience, our CIL approach will be referred as CILS whereas standard token-based approach will be referred as STDS at the rest of this paper.

Based on the fact that each plagiarism case on Karnalim's dataset are considered as true-positive plagiarism, the effectiveness of our approach is increased proportionally to its average similarity result. The closer its average similarity result to 100%, the more effective that approach is. In other words, we can compare CILS and STDS approach effectiveness simply based on their average similarity. An approach with higher average similarity is considered to be more effective for handling plagiarism attacks. Similarity result for each plagiarism case using CILS and STDS can be seen on Figure 3. Horizontal axis represents plagiarism cases whereas vertical axis represents similarity value. From 42 cases on our dataset, CILS outperforms STDS on 32 cases. It only loses on 10 cases which are *3008*, *5004*, *5008*, *5009*, *5012*, *5016*, *5017*, *6003*, *6010*, and *6011*. When discovered further, CILS similarity on these cases is extremely lower due to its small token size. CILS tends to have extremely smaller size of tokens than STDS since it only considers semantic-preserving instructions. Consequently, when normalized using (1), even one mismatched token or gap may yield significant similarity drop. Mismatched token and gap are two terminologies which are defined on Adaptive Local Alignment. Mismatched token represents token subsequence which are not shared on both sequences and gap represents the distance required to generate the matched one.

To strengthen our finding about CILS loss, the detail of mismatched tokens and gaps toward these cases are given on Figure 4. However, to simplify the detail, we merge the number of mismatched tokens and gaps as penalties. According to Lim et al [34], the number of penalties is calculated based on (2) where gap's impact is doubled. As seen in Figure 4, the number of CILS penalties is quite similar to the number of STDS penalties. It only differs significantly on *5008*, *5016*, *5017*, *6003*, and *6011*. Yet, out of 5 significant cases, 4 of them yield lower number of penalties on CILS. It only yields higher penalties on *5017* since involved plagiarism attack on this case is over-technical. *5017* replaces standard loop for traversing a collection with *for-each*. Even though standard loop and *for-each* is quite similar in term of lexical source code structure, it yields significant difference when converted into CIL. *for-*

each is intended to handle traversal for all collection so that its CIL representation is more complex than the standard loop itself.

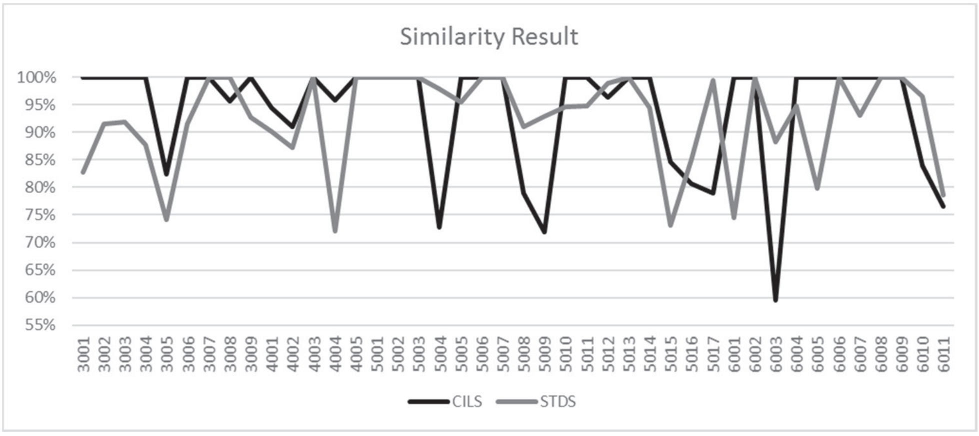


Figure 3. Similarity Result based on Karnalim's Dataset

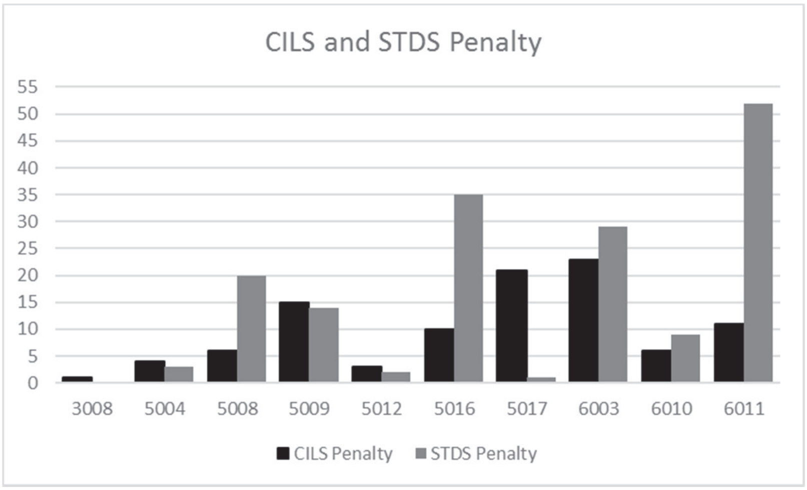


Figure 4. The number of CILS and STDS Penalties on Plagiarism Cases where CILS Similarity is lower than STDS

$$penalty(A,B) = mismatch(A,B) + 2*gap(A,B) \qquad (2)$$

In general, CILS outperforms STDS based on following facts: 1) CILS outperforms STDS in most cases (32 of 42 cases); 2) Most CILS losses are caused by its limited token size. When compared in term of penalty, CILS are still more beneficial than STDS; 3) In term of average similarity, CILS outperforms STDS by 1.408% difference. CILS yields 93.893% average similarity whereas STDS yields 92.485%.

Efficiency between CILS and STDS is measured based on the number of involved processes for executing the given approach. We incorporate approximate estimated time instead of the real one based on following reasons: 1) Real execution time is greatly affected by hardware and operating system dependency; 2) Using real execution time for comparison purpose only works well *iff* running programs take a considerable amount of time; and 3) Each source code on our dataset is small in size. Thus, it may yield faulty result when measured using real execution time since running time for each source code is considerably fast. It will be greatly affected by hardware and operating system dependency.

Approximate estimated time for CILS and STDS can be defined as (3) and (4) respectively where A and B are token size for each given sequence. Each phase on Figure 1 except similarity measurement takes linear complexity for both sequences. These phases take $A+B$ processes each. On the contrary, similarity measurement itself takes $A*B$ processes since its algorithm is based on matrix-like representation. $T_{CILS}(A,B)$ is calculated based on 4 steps on CILS which are: compilation phase, .NET CIL disassembler, .NET CIL preprocessing, and similarity measurement. We assume that all source codes are compilable to generate worst case for its estimated time. A_0 and B_0 are the number of token in source code before compiled. Both of them are required to measure the number of processes on compilation phase since source code compilation takes linear complexity based on the given source code. On the other hand, $T_{STDS}(A,B)$ is calculated based 2 steps on STDS: generating token sequence and similarity measurement. Generating token sequence takes linear number of processes whereas similarity measurement takes $A*B$ processes.

$$T_{CILS}(A,B) = (A_0+B_0) + 2*(A+B) + (A*B) \tag{3}$$
$$T_{STDS}(A,B) = (A+B) + (A*B) \tag{4}$$

Approximate estimated time for each plagiarism case using CILS and STDS can be seen on Figure 5. Horizontal axis represents plagiarism cases whereas vertical axis represents approximate estimated time based on T_{CILS} and T_{STDS}. As seen in Figure 5, CILS involves smaller number of processes than STDS in all cases. It reduces about 94.552% number of processes when compared with STDS. This significant difference is natural since CILS involves lower token size than STDS. Average token size of CILS is 80.691% lower than STDS. Thus, even though approximate estimated time equation for CILS takes higher complexity than STDS, CILS is still more efficient than STDS due to its lower token size.

4.2. Evaluating Our Approach toward Multiple Plagiarism Attacks

This evaluation is conducted to measure the impact of our proposed approach for handling real-case plagiarisms which typically incorporate various plagiarism attacks at once. In order to do that, we ask three respondents to plagiarize three C# source codes with any possible plagiarism attacks (3 respondents * 3 source codes = 9 plagiarism cases). C# source codes utilized in this evaluation are UVa750, UVa10003, and UVa11450 which are taken from Competitive Programming 3 book [47].

However, since these source codes are originally written in Java, they are rewritten to C# by the first author. In addition, to keep our dataset consist of various plagiarism attacks, we limit our respondents to lecturer assistants. Lecturer assistants should have a lot of experience for handling plagiarism attempts so that they may incorporate various plagiarism attacks at once.

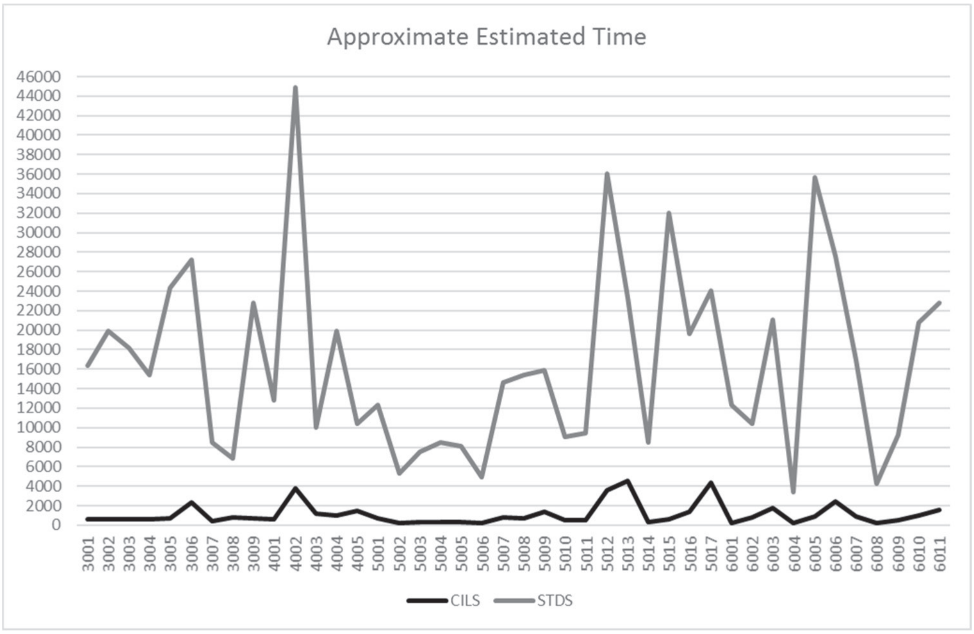


Figure 5. Approximate Estimated Time based on Karnalim's Dataset

Similarity result for each plagiarism case using CILS and STDS can be seen on Figure 6. Horizontal axis represents similarity value and vertical axis represents plagiarism case conducted by our respondents. In general, CILS characteristics on multiple plagiarism attacks is similar with its characteristics when handling plagiarism attacks separately. CILS similarity may significantly drop even though it only suffers small mismatches and gaps. However, even though CILS similarity is lower than STDS in many cases, its deficiency is still small since its average similarity is only 1.54% lower than STDS.

When perceived from the number of penalties, STDS yields higher number of penalties than CILS for handling multiple plagiarism attacks. As seen in Figure 7, STDS yields higher penalties in all cases than CILS. This finding is quite different with our previous finding about CILS-STDS penalty on single plagiarism attacks. As in single plagiarism attacks, the number of CILS and STDS penalties is quite similar to each other. When discovered further, STDS tends to yield more penalties on longer source code due to more delimiter tokens incorporated on source code. These tokens may obfuscate similar pattern and reduce the effectiveness of Adaptive Local Alignment.

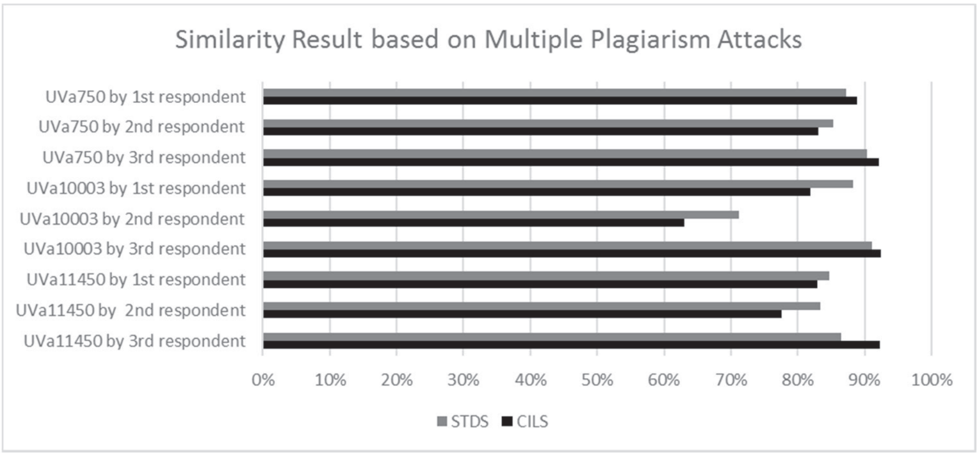


Figure 6. Similarity Result based on Multiple Plagiarism Attacks

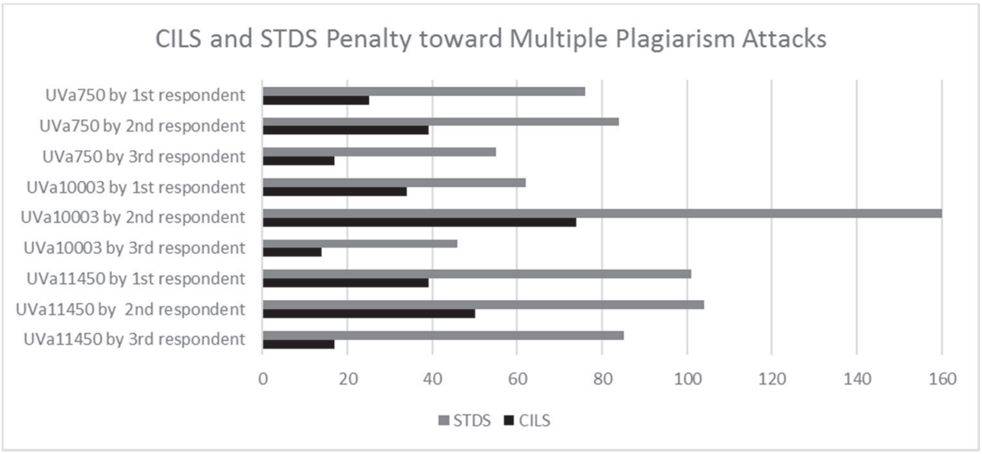


Figure 7. The number of CILS and STDS Penalties toward Multiple Plagiarism Attacks

In terms of efficiency, CILS still outperforms STDS since the number of involved tokens in CILS is extremely smaller than STDS. The detail of approximate estimated time from both approaches can be seen on Figure 8. It is calculated based on T_{CILS} and T_{STDS} defined in (3) and (4) respectively. When compared based on the average number of involved processes, CILS involves 86.09% processes lower than STDS. Thus, it can be stated that CILS is more efficient than STDS in terms of involved processes.

5. Conclusion and Future Works

In this paper, we have extended Juričić et al approach [10] by incorporating Adaptive Local Alignment (ALA) as a replacement of their Greedy String Tiling (GST). This replacement is conducted based on Lim et al's work [34] which states that ALA performs better than GST in terms of sensitivity and specificity. In addition, we also evaluate the impact of our approach (CILS) when compared to standard lexical-token approach (STDS). In general, CILS is more resistible for handling various plagiarism attacks separately. It outperforms STDS with 1.408% difference. Even though CILS may yield lower similarity than STDS due to its smaller token size, its number of penalties is still lower than STDS, especially on longer source code. However, CILS is weak against plagiarism cases which are focused on API and technical mechanism. These plagiarism cases yield significant change when converted into intermediate form. In terms of efficiency, CILS is more efficient than STDS since it reduces a vast amount of processes due to its limited token size. It reduces about 80.691% processes when compared with STDS based on Karnalim's dataset [3].

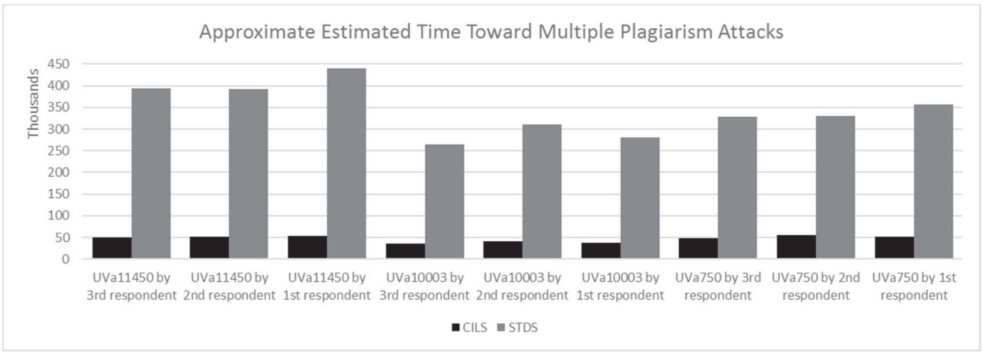


Figure 8. Approximate Estimated Time on Multiple Plagiarism Attacks

When handling multiple plagiarism attacks, CILS yields lower accuracy than STDS. It yields 1.54% lower similarity based on our dataset. When discovered further, its deficiency is caused by implemented normalization in (1). This finding strengthens our previous finding about CILS effectiveness. CILS similarity may become over-sensitive since one mismatch or gap may reduce its similarity percentage significantly. However, when perceived in the number of penalties, CILS yields extremely smaller number of penalties than STDS since STDS considers code delimiter token as its token member. These tokens may obfuscate similar pattern and reduce the effectiveness of Adaptive Local Alignment. In term of efficiency, CILS is still more efficient than STDS on multiple plagiarism attacks. It even reduces more token size since CIL representation is more compact than the source code itself. In general, we can conclude that CILS is better than STDS in terms of effectiveness and efficiency.

In next research, our proposed method will be expanded to handle cross-language source code plagiarism for all .NET programming languages. It is expected to detect plagiarism even though given source codes are written on different .NET programming languages. Moreover, we will also propose a plagiarism detection system which enable our approach to be implemented in programming course.

References

[1] G. Cosma and M. Joy, "Towards a Definition of Source-Code Plagiarism," *IEEE Transactions on Education,* vol. 51, no. 2, pp. 195 - 200, 2008.

[2] D. Sraka and B. Kaucic, "Source Code Plagiarism," in *The 31st International Conference on Information Technology Interfaces*, 2009.

[3] O. Karnalim, "Detecting Source Code Plagiarism on Introductory Programming Course Assignments Using a Bytecode Approach," in *The 10th International Conference on Information & Communication Technology and Systems (ICTS)*, Surabaya, 2016.

[4] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," School of Computing, Queen's University, Canada, 2007.

[5] S. Burrows, S. M. M. Tahaghoghi and J. Zobel, "Efficient and effective plagiarism detection for large code repositories," *Software-Practice & Experience,* vol. 37, no. 2, 2007.

[6] Z. Duric and D. Gasevic, "A Source Code Similarity System for Plagiarism Detection," *The Computer Journal,* vol. 55, 2012.

[7] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology,* vol. 147, 1981.

[8] L. Prechelt, G. Malpohl and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *Journal of Universal Computer Science,* vol. 8, no. 11, 2002.

[9] C. Kustanto and I. Liem, "Automatic Source Code Plagiarism Detection," in *SNPD '09. 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, Daegu, 2009.

[10] V. Juricic, T. Juric and M. Tkalec, "Performance evaluation of plagiarism detection method based on the intermediate language," *INFuture2011: "Information Sciences and e-Society",* 2011.

[11] S. Hannabuss, "Contested texts: issues of plagiarism," *Library Management,* vol. 22, 2001.

[12] H. Maurer, F. Kappe and B. Zaka, "Plagiarism - A Survey," *Journal of Universal Computer Sciences,* vol. 12, no. 8, 2006.

[13] N. Heintze, "Scalable document fingerprinting," in *USENIX Workshop on Electronic Commerce*, Oakland, 1996.

[14] R. Brixtel, M. Fontaine, B. Lesner and C. Bazin, "Language-independent clone detection applied to plagiarism detection," in *10th IEEE Working Conference on Source Code Analysis and Manipulation*, Timisoara, 2010.

[15] T. Hoad and J. Zobel, "Methods for identifying versioned and plagiarised documents," *Journal of The American Society for Information Science and Technology,* vol. 54, no. 3, 2002.

[16] G. Cosma and M. Joy, "Evaluating the performance of LSA for source-code plagiarism detection," *Informatica,* vol. 36, pp. 409-424, 2012.

[17] B. Croft, D. Metzler and T. Strohman, Search Engine : Information Retrieval in Practice, Boston: Pearson Education .Inc, 2010.

[18] J. Donaldson, A. Lancaster and P. Sposat, "A Plagiarism Detection System," in *The 12th SIGCSE Technical Symposium on Computer Science Education*, New York, 1981.

[19] M. H. Halstead, "Natural laws controlling algorithm strcuture?," *ACM SIGPLAN Notices,* vol. 7, no. 2, 1972.

[20] M. H. Halstead, "Elements of software science (Operating and programming systems series)," Elsevier Science, New York, 1977.

[21] U. Bandara and G. Wijayarathna, "A machine learning based tool for source code plagiarism detection," *International Journal of Machine Learning and Computing,* vol. 1, no. 4, 2011.

[22] R. C. Lange and S. Mancoridis, "Using code metric histograms and genetic algorithms to perform author identification for software forensics," in *The 9th annual conference on Genetic and evolutionary computation*, New York, 2007.

[23] A. Ohno and H. Murao, "A two-step in-class source code plagiarism detection method utilizing improved CM algorithm and SIM," *International Journal of Innovative Computing, Information, and Control,* vol. 7, no. 8, 2011.

[24] S. Engels, V. Lakshmanan and M. Craig, "Plagiarism detection using feature-based neural networks," in *The 38th SIGCSE technical symposium on Computer science education*, New York, 2007.

[25] A. Jadalla and A. Elnagar, "PDE4Java: Plagiarism Detection Engine for Java source code: a clustering approach," *International Journal of Business Intelligence and Data Mining,* vol. 3, no. 2, 2008.

[26] A. Ramirez-de-la-Cruz, G. Ramirez-de-la-Rosa, C. Sanchez-Sanchez, H. Jimenez-Salazar, C. Rodriguez-Lucatero and W. A. Luna-Ramirez, "High level features for detecting source code plagiarism across programming languages," in *Cross-Language Detection of SOurce COde Re-use Conference*, 2015.

[27] S. Schleimer, D. S. Wilkerson and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," in *The ACM SIGMOD International Conference on Management of Data*, San Diego, 2003.

[28] D. Gitchell and N. Tran, "Sim: a utility for detecting similarity in computer programs," in *The 13th Technical Symposium on Computer Science Education*, New Orleans, 1999.

[29] P. Clough, "Plagiarism in natural and programming languages: an overview of current tools and technologies," Department of Computer Science, University of Sheffeld, 2000.

[30] M. J. Wise, "YAP3: Improved detection of similarities in computer programs and other texts," *ACM SIGCSE Bulletin,* vol. 28, no. 1, 1996.

[31] A. Ahtiainen, S. Surakka and M. Rahikainen, "Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises," in *The 6th Baltic Sea Conference on Computing Education Research*, Uppsala, 2006.

[32] M. Mozgovoy, K. Frederiksson, D. R. White, M. S. Joy and E. Sutinen, "Fast plagiarism detection system," *Lecture Notes in Computer Science,* vol. 3772, 2005.

[33] J. Hage, P. Rademaker and N. Van Vugt, "Plagiarism detection for Java: a tool comparison," in *The 11th Computer Science Education Research Conference*, Heerlen, 2011.

[34] J.-S. Lim, J.-H. Ji, H.-G. Cho and G. Woo, "Plagiarism detection among source codes using adaptive local alignment of keywords," in *The 5th International Conference on Ubiquitous Information Management and Communication*, Seoul, 2011.

[35] D. Shah, H. Jethani and S. H. Joshi, "(CLSCR) Cross Language Source Code Reuse Detection Using Intermediate Language.," in *FIRE Workshop*, 2015.

[36] J.-H. Ji, G. Woo and H.-G. Cho, "A Plagiarism Detection Technique for Java Program Using Bytecode Analysis," in *ICCIT '08. Third International Conference on Convergence and Hybrid Information Technology*, Busan, 2008.

[37] D. Pawelczak, "Online detection of source-code plagiarism in undergraduate programming courses," in *The International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, Athens, 2013.

[38] M. Chilowicz, É. Duris and G. Roussel, "Finding Similarities in Source Code Through Factorization," in *8th Workshop on Language Descriptions, Tools and Applications*, 2008.

[39] M. G. Ellis and C. W. Anderson, "Plagiarism Detection in Computer Code," 2005.

[40] M. Chilowicz, E. Duris and G. Roussel, "Syntax tree fingerprinting for source code similarity detection," in *IEEE 17th International Conference on Program Comprehension*, Vancouver, 2009.

[41] V. Juričić, "Detecting source code similarity using low-level languages," in *33rd International Conference on Information Technology Interfaces*, Dubrovnik, 2011.

[42] T. Parr, "ANTLR," 2014. [Online]. Available: http://www.antlr.org/. [Accessed 07 12 2015].

[43] "GitHub - antlr/grammars-v4: Grammars written for ANTLR v4; expectation that the grammars are free of actions.," [Online]. Available: https://github.com/antlr/grammars-v4. [Accessed 8 12 2016].

[44] "Ildasm.exe (IL Disassembler)," [Online]. Available: https://msdn.microsoft.com/en-us/library/f7dy01k1(v=vs.110).aspx. [Accessed 8 12 2016].

[45] "Partition III - CIL - Microsoft," [Online]. Available: download.microsoft.com/download/7/3/3/733ad403.../ms%20partition%20iii.pdf. [Accessed 8 12 2016].

[46] J. A. W. Faidhi and S. K. Robinson, "An Empirical approach for detecting program similarity and plagiarism within a university programming environment," *Computer & Education,* vol. 11, no. 1, pp. 11-19, 1987.

[47] S. Halim and F. Halim, Competitive Programming 3, lulu, 2013.