A Low-Level Structure-based Approach for Detecting Source Code Plagiarism

Article *in* IAENG International Journal of Computer Science · December 2017

CITATIONS
11

READS
260

1 author:

Oscar Karnalim
University of Newcastle
50 PUBLICATIONS **190** CITATIONS

SEE PROFILE

A Low-Level Structure-based Approach for Detecting Source Code Plagiarism

Oscar Karnalim

Abstract—**According to the fact that source code plagiarism is an emerging issue in Computer Science programming courses, several source code plagiarism detection approaches are developed. One of them is Karnalim's approach, an approach which detects plagiarism based on low-level tokens. This paper proposes an expansion of such approach by incorporating three contributions which are: flow-based token weighting; argument removal heuristic; and invoked method removal. Flow-based token weighting aims to reduce the number of false-positive results; argument removal heuristic aims to generate more-accurate linearized method content; and invoked method removal aims to fasten processing time. According to our evaluation, three findings can be deducted about proposed approach. Firstly, advantages provided by our proposed approach are prominent in both controlled and empirical environment. Secondly, our proposed approach outperforms Karnalim's and state-of-the-art approach in terms of time efficiency. Finally, our approach is moderately effective to handle plagiarism attacks in practical environment.**

***Index Terms*—plagiarism detection, source code, low-level language, java, bytecode**

I. INTRODUCTION

PLAGIARISM is an act for reusing other people's work without acknowledging them as its original author(s) beforehand [1, 2, 3]. In undergraduate Computer Science (CS) major, it emerges as a serious issue since most assignments are conducted electronically. A student coursework can be easily copied and pasted as a new one in a no time. Moreover, since plagiarists are not only limited into weak students [4], detecting this illegal behavior will require a lot of effort. Plagiarized source code might contain complex plagiarism attacks and deciding its originality usually takes a considerable amount of time. Regarding to these issues, an automatic plagiarism detection approach is highly desirable to extenuate lecturer effort for detecting such plagiarism manually.

Source code plagiarism is a specific plagiarism issue which comes into source code domain. When compared to plagiarism on other domains, we would argue that this issue is the most prominent one on CS majors based on twofold. On the one hand, source code is the most frequent representation that is used to complete CS assignments [5]. Such finding is natural since programming is one of the core topic in CS and most programming courses usually ask student to submit source codes to complete weekly assignment. On the other hand, source code is a potential to-be-plagiarized representation. Such finding is deducted from the fact that most source code assignments are graded using an auto-grader [6] instead of human evaluator and tricking such system is easy as long as the students know how it works.

To handle such prominent issue, this paper proposes a source code plagiarism detection approach that relies on low-level representation, which is Bytecode, an executable code for Java programming language. In fact, such form has been frequently used in various research tasks such as software watermarking [7], software retrieval system [8], software keyphrase extraction [9], and virtual machine optimization [10]. Our approach is extended from Karnalim's approach [11] by incorporating threefold: flow-based token weighting, argument removal heuristic, and invoked methods removal. These additional features are expected to generate higher effectiveness and efficiency for low-level approach.

II. RELATED WORKS

When detecting source code plagiarism, most approaches are classified into two categories which are attribute-based and structure-based approach [12]. Attribute-based approach detects plagiarism by comparing key properties from given source codes whereas structure-based approach detects plagiarism by comparing source code ordinal structure. It is important to note that such classification is not agreed by all researchers. Some of them claim that text-based approach should be added beside both approaches [13, 14, 4]. According to their perspective, such approach should be explicitly defined since it does not take source code features into consideration. It only treats source code as a raw text during its process. Yet, we would argue that, in terms of methodology, such approach can still be considered as either attribute-based or structure-based approach. Therefore, in this section, all implementations of text-based approach will be mapped to the first two approaches: attribute-based and structure-based approach.

Attribute-based approach (ABA) extracts key properties from source codes and compares them to each other for detecting plagiarism. Two or more source codes are considered as plagiarized to each other *iff* these source codes yield similar key properties. Earlier work of this approach is conducted by Ottenstein using software science metrics [15]. However, since key properties in his work are not sufficient to represent source code characteristics, additional properties, such as the number of variables, methods, loops, conditional statements, method invocations, and programmer style

Manuscript received April 05, 2017; revised October 04, 2017. This work was supported by Maranatha Christian University, Indonesia.
Oscar Karnalim is with Faculty of Information Technology, Maranatha Christian University, Indonesia (email: oscar.karnalim@it.maranatha.edu; ORCID: http://orcid.org/0000-0003-4930-6249).

behavior, are introduced on further works about ABA [16, 17, 12, 18, 19, 20].

According to the fact that most plagiarized codes do not share exactly-similar characteristics toward their original code, several ABAs incorporate approximate characteristic-matching instead of the exact one. In such manner, detected plagiarism is not only limited to verbatim copy but also partially-similar copy. Generally speaking, such approximate matching is adapted from threefold: Information-Retrieval (IR), Machine Learning (ML), and domain-specific measurement. Firstly, IR-based matching defines similarity based on standard IR similarity algorithms. Two works which utilize such matching are Ramirez-de-la-Cruz et al's work [21], which incorporates Cosine Similarity, and Cosma & Joy's work [22], which incorporates Latent Semantic Analysis. Secondly, ML-based matching defines similarity based on classification and clustering algorithm. Two works which utilize such matching are Bandara & Wijayarathna's work [19], which combines three classification algorithms for deducting similarity, and Jadalla & Elnagar's work [23], which defines similarity based on similar cluster. Finally, domain-specific matching defines similarity based on domain-specific similarity measurement. Two works which utilize such matching mechanism are Merlo's work [24], which incorporates spectral similarity, and Smeureanu & Iancu's work [25], which incorporates graph similarity.

Structure-based approach (SBA) detects plagiarism based on ordinal structure similarity. Two or more source codes are considered as plagiarized to each other *iff* these source codes share similar structure. Typically, this approach is more accurate than ABA even though it takes longer processing time. In general, structure similarity in such approach is determined based on two phases. First of all, source codes are converted into intermediate representation such as source code token [26, 1, 27, 16, 28, 29], compiler-based representation [30, 31, 32], or low-level token [33, 34, 35, 11, 4]. Afterwards, tokens from two source codes will be treated as two sequences and then compared using string matching algorithm such as Rabin-Karp Greedy String Tiling (RK-GST) [36], Winnowing Algorithm [37], and Local Alignment [38].

Source code token refers to lexical token extracted from source code using programming-language-specific lexer and parser. Such representation has been implemented in numerous works [26, 1, 16, 28, 29], including publicly available plagiarism detection tools [27, 37, 36], since it can be generated easily in a no time. However, despite its popularity, we would argue that such representation is weak against high level plagiarism attacks such as modifying control flow and encapsulating instructions as methods.

Compiler-based representation refers to an intermediate form which is tightly-related with compiler processes. To the best of our knowledge, there are three works which explicitly use such representation. These works are Chilowics et al's work [31], which incorporates syntaxtree, Ellis & Anderson's work [32], which incorporates inorder-linearized parse tree, and Chilowics et al's work [30], which incorporates call graph and information metrics.. Even though compiler-based representation is more effective than source code token for detecting similarity, such representation usually takes numerous processes to be generated, especially when given programming language grammar is rather complex.

Low-level token refers to the content of executable file that is resulted from compiling source code. We would argue that such representation is more effective and efficient when compared to other representations since low-level representation typically contains only semantic-preserving instructions and most syntactic sugars on that form are automatically translated into its original form [11, 4]. Generally speaking, related works that incorporate low-level tokens can be classified into twofold: works that are focused on Java programming language and works that are focused on .NET programming languages. On the one hand, works that are focused on Java programming language was initiated by Ji et al [35]. They extract low-level tokens, which is Java bytecode in their case, from source code executables and compare them directly using string similarity algorithm. Their work, at some extent, is extended by Karnalim [11] with several new contributions such as recursive-handled method linearization, instruction generalization, and instruction interpretation. On the other hand, works that are focused on .NET programming languages was initiated by Juričić [33]. He detects source code plagiarism by converting source codes into Common Intermediate Language (CIL) tokens and determining their similarity using Levenstein distance. Juričić et al [34] and Rabbani & Karnalim [4] then extend his work by replacing its similarity algorithm and modifying minor features.

Instead of utilizing only either ABA or SBA for detecting source code plagiarism, three works combine both approaches in order to get more accurate result. Firstly, Menai & Al-hassoun [39] displays source code similarity from both ABA and SBA at once as its result. Such displayed results are expected to help users for determining plagiarized codes according to given assignment. If given assignment allows a little modification, users can rely on ABA's result. Otherwise, they can rely on SBA's result. As we know, ABA is less sensitive than SBA for detecting plagiarism. Secondly, Engels et al [40] incorporates MOSS similarity result, which is a result from SBA, as one of its learning feature to classify whether two source codes are plagiarized to each other. The classification itself is determined based on 12 attributes and could be roughly classified into ABA. Finally, Ohno & Murao [41] combines programming style and structural similarity to determine plagiarism. A source code is considered as a plagiarized code *iff* its programming style is significantly different with programming style found on previously submitted assignments and its structure is similar to other source code. In their case, programming style is determined using ABA whereas structural similarity is determined using SBA.

At some points, both ABA and SBA are not combined to get more accurate result. Yet, it aims for more efficient processing time by sacrificing its accuracy. Such approach is typically implemented on large-scale source code repositories where pure SBA is not applicable since it will take a lot of processing time. The work of Burrows et al [14] is an example which falls into this category. It incorporates ABA and SBA as two layers for determining plagiarism. ABA is conducted to select initial plagiarism candidates whereas SBA is conducted to revalidate candidates given by ABA. In such manner, processing time required for detecting plagiarism may be lower than pure SBA since not all source codes are

compared using SBA. However, it may also yield lower accuracy than pure SBA since not all potential candidates could be detected through ABA as its first layer. Mozgovoy et al [42], at some extent, shares similar idea with Burrows et al except that they incorporate different ABA and SBA.

According to the fact that SBA outperforms ABA in most plagiarism cases [27, 16] and low-level tokens, at some extent, only represent source code semantic, this paper extends low-level SBA proposed by Karnalim [11] for detecting source code plagiarism. It is preferred to other low-level SBAs since his work has considered many aspects for detecting plagiarism (e.g. recursive-handled method linearization). It is important to note that our work does not focus on the combination of ABA and SBA directly since we believe that a well-developed SBA may also indirectly enhance the effectiveness of such combined form. Such combination will be observed further on other works.

Karnalim's work [11] is extended by incorporating following contributions: 1) Flow-based token weighting is introduced to generate more-sensitive result; 2) Argument removal heuristic is introduced to generate more-precise method linearization; and 3) Invoked method removal is introduced to fasten processing time. These contributions are expected to enhance the effectiveness and efficiency of current state-of-the-art of low-level approach, which is Karnalim's approach [11]. In terms of evaluation, our work will be evaluated based on four aspects which are the effectiveness toward controlled environment, the effectiveness toward empirical environment, time efficiency, and qualitative perspective. We would argue that these aspects could comprehensively exploit the characteristics of our proposed approach.

III. METHODOLOGY

A. Proposed System Flowchart

Our proposed low-level approach detects source code plagiarism by following system flowchart given on Fig. 1. Such flowchart is generalized from Rabbani & Karnalim's work [4] to make it applicable to other low-level SBAs. In our flowchart, detecting source code plagiarism is split into twofold, which are compilation and comparison phase.

In compilation phase, each source code will be compiled to its respective executable form. Even though this mechanism seems to slow down execution time, Rabbani & Karnalim [4] shows that it is still more efficient when compared to state-of-the-art SBA that uses source code token representation. Low-level form tends to have fewer tokens than its source code. Thus, it will obviously generate fewer processes and shorter time to detect plagiarism. In fact, this compilation phase could be skipped if submitted assignments are formed as IDE-generated projects. Most IDEs, such as Netbeans [43] or Visual Studio [44], generate executable files and store it on project directory each time source codes are compiled. Thus, with an assumption that most students tend to compile their code to recheck its correctness, IDE-generated executable files can be considered as a replacement of compilation phase result.

In comparison phase, all source codes are compared to each other in either low-level or source code token format. On the one hand, low-level token format, which will be generated by low-level token extraction, will be used when both compared source codes successfully generate low-level codes. On the other hand, source code token format, which will be generated by source code token extraction, will be used when at least one of the compared codes are uncompilable. In such manner, our flowchart can still detect plagiarism on uncompilable source codes, even though its similarity result may not be as sensitive as the low-level one. After measured, all pairs which similarity exceeds plagiarism threshold will be returned as our flowchart output.

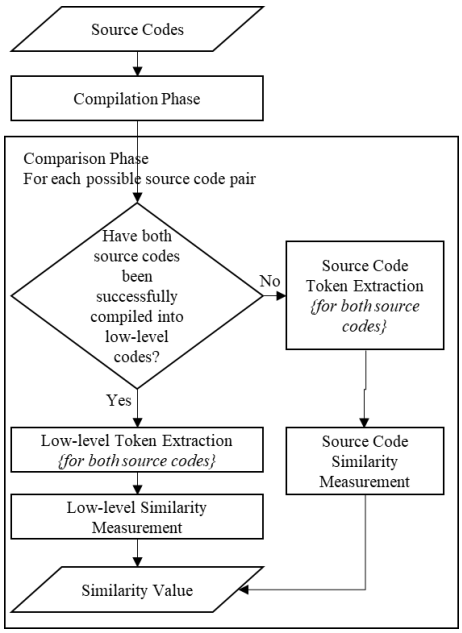

Fig. 1. Proposed Flowchart for Detecting Source Code Plagiarism

In our implementation, we apply several modifications on given flowchart as follows: 1) Our work is focused on Java programming language with Bytecode as its low-level form since we expand Karnalim's work [11]; 2) Similarity measurement is implemented based on minimum matching similarity [45], using RK-GST algorithm with 2 as its Minimum Matching Length (MML); 3) Our work does not define plagiarism threshold as a static constant since such threshold might be varied per case, regarding to assignment difficulty and possible applied plagiarism attacks. We will leave the decision of the best plagiarism threshold to the user; 4) Source code token extraction is conducted using ANTLR [46] and grammar listed on ANTLR GitHub repositories [47]; and 5) Source code similarity measurement naively considers each source code as a long token sequence.

B. Low-level Token Extraction

For each executable file from source code, its low-level tokens are extracted through five sequential phases (which detail can be seen on Fig. 2): 1) Raw low-level tokens are extracted from executable file and grouped per method through method content extraction; 2) All tokens are weighted based on their execution probability and number of containing loop through flow-based token weighting; 3) Tokens which invoke recursive method will be removed through recursive-method invocation elimination; 4) Tokens which invoke non-recursive method will be replaced with their respective invoked-method content through method linearization; and 5) The contents of methods that have been invoked on other

methods are removed from low-level tokens through invoked method removal.

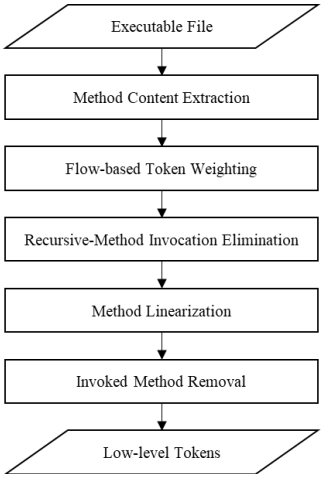

Fig. 2. Low-level Token Extraction Phases

It is important to note that our low-level token extraction is extended from Karnalim's work [11] by modifying method linearization and incorporating two new phases, which are flow-based token weighting and invoked method removal. These new or modified phases are expected to enhance the effectiveness and efficiency of our proposed source code plagiarism detection.

C. Method Content Extraction

This phase is responsible to extract low-level tokens from executable file, which is a set of class files in our case. Low level tokens are extracted using Javassist [48, 49, 50] where tokens for each extracted method will be reinterpreted and generalized based on Karnalim's work [11]. It is important to note that both reinterpretation and generalization are conducted to reduce the impact of over-technical implementation on low-level code. According to Karnalim's work [11], such over-technical implementation might reduce the accuracy of low-level approach since some source code fractions can be converted into more than one low-level representation regarding to its technical circumstances.

D. Flow-based Token Weighting

This phase is responsible to weigh each token based on Control Flow Graph (CFG). In general, each token would be assigned with two weight constants which are execution probability and the number of containing loop. Execution probability refers to the possibility of given token to be executed based on method content's CFG. It is represented as floating value between 0 to 1 inclusively where 0 refers to "never be executed" and 1 refers to "always be executed ". On the contrary, the number of containing loops represents how many loops are responsible for executing given token. It is represented as a non-negative number which is assigned as 0 by default. Such default value means that no loop contains given token.

A brief example about how flow-based token weighting works can be seen on Table I. On such example, a pseudo-code of linear search algorithm is weighted based on execution probability and the number of containing loops. On the one hand, execution probability for most lines are assigned as 1, which means that such line will always be executed regardless of its input. Line 9, 11, and 13 are the only lines which probability is not 1. They are assigned with 0.5 since they only have 50% execution chance. They are only executed if given condition from their respective previous lines (line 8, 10, and 12) has been fulfilled. On the other hand, the number of containing loops for most lines are assigned as 0 since they are not placed under a loop. Line 3, 4, 7, 8, and 9 are exceptional since they are assigned with 1 as its number of containing loop. Such constant means that these lines are placed under a loop.

TABLE I
FLOW-BASED TOKEN WEIGHTING ON LINEAR SEARCH ALGORITHM

Line	Algorithm	Execution Probability	The Number of Containing Loops
1	`n = input()`	1	0
2	`lst = new Array()`	1	0
3	`for i to n do`	1	1
4	`  lst[i] = input()`	1	1
5	`s = input()`	1	0
6	`idx = -1`	1	0
7	`for i to n do`	1	1
8	`  if lst[i] = s do`	1	1
9	`    idx = i`	0.5	1
10	`if idx = -1 do`	1	0
11	`  print("not found")`	0.5	0
12	`else`	1	0
13	`  print("found")`	0.5	0

To assign weight constants for each token, our work extends Karnalim & Mandala's weighting mechanism that generates CFG from regular sequence, *goto*, switch-case, and exception flow [8]. Execution probability for each token is assigned based on pseudo-execution toward resulted graph whereas the number of containing loops is assigned based on the number of detected nested loop. However, in our work, we simplify nested loop detection mechanism since our work needs no information about loop type. The simplified implementation of such mechanism works as follows:

a) Generate all possible Strongly-Connected Components (SCC) from given CFG using Tarjan's algorithm [51].
b) For each SCC, increment the number of containing loop on each of its token.
c) Remove the first token for each SCC and do similar procedures (from a to c) recursively only on remaining SCC members,
d) Repeat until all SCC are processed.

Both weight constants will be used to enhance the sensitivity of our proposed similarity measurement. When comparing two tokens, our approach will not only compare mnemonic but also weight constants. Two tokens are considered as different tokens even though they share similar mnemonic if they share different weight constants. In such manner, resulted similarity would be more accurate, resulting fewer false positive results.

For a broader view about improved sensitiveness through flow-based weighting, we can see algorithm given on Table II that contain three *print-hello* instructions. Logically, these

instructions should not be considered as similar to each other since they are not always executed once each time the algorithm is invoked. *print-hello* on line 2 will be always executed once; *print-hello* on line 4 may be executed more than once; and *print-hello* on line 6 only has 50% probability to be executed. Using flow-weighting mechanism, those *print-hello* instructions can be distinguished to each other, even though they share similar mnemonic, since they share different weight constants.

TABLE II
A CASE STUDY TO SHOW THE IMPACT OF FLOW-BASED WEIGHTING

Line	Algorithm
1	`s = input()`
2	`print("Hello")`
3	`for i to s do`
4	`  print("Hello")`
5	`if s > 0 do`
6	`  print("Hello")`

Even though such weighting mechanism seems weak against dummy-flow plagiarism attacks when perceived from source code perspective, we would argue that such weighting mechanism offers more benefits when implemented on low-level forms. Compilation phase, which is used to convert source code to low-level form, usually removes unnecessary flows and optimizes them directly. Therefore, most dummy-flow plagiarism attacks will be removed and have no impact on low-level form. They only put some effects if applied flows rely on user input or method parameter.

E. Recursive-Method Invocation Elimination

This phase is responsible to remove all recursive-method invocation tokens from method content. Such tokens are eliminated in this phase since these methods will yield endless linearization processes at method linearization, the 4th phase of low-level token extraction. This phase adapts Karnalim & Mandala's approach [8] which works in threefold. Firstly, all method invocations are converted to directed graph where A → B states that at least one token from method A invokes method B. Secondly, Strongly-Connected Components (SCC) from given graph are detected using Tarjan's algorithm [51]. Finally, methods that are included on recursive SCC are marked as recursive methods and all method invocation tokens which invoke these methods are removed from method contents.

F. Method Linearization

This phase is responsible to linearize method contents by replacing all method invocations with their respective invoked-method content. It is extended from Karnalim's work [11] by incorporating argument removal heuristic, a heuristic that is able to approximately remove tokens for preparing method invocation's argument. According to Karnalim's work [11], such tokens are the main reason why his work is weak against inlining and outlining method. Each time his approach linearizes a method by replacing method invocation token with its respective invoked-method content, tokens for preparing such method's argument are still remained, causing numerous mismatched tokens.

In Bytecode, preparing arguments for a method invocation is usually implemented by pushing values to runtime stack wherein the number of pushed values is typically similar with the number of method's parameters. Therefore, our heuristic is implemented by simply removing N tokens before such invocation where N represents the number of invoked method's parameters. It is important to note that object caller reference, which is implicitly embedded as an additional parameter for non-static method invocation, is also considered as a method parameter in our work. Consequently, when a non-static method is invoked, N will be assigned as the number of explicit method parameter + 1.

In fact, our heuristic is not always accurate, especially for handling arguments that involve additional operations such as arithmetic operation, method invocation, or object creation. These arguments might generate more than one token per argument since their value is resulted from other process and that process might be represented as numerous tokens, resulting inaccurate argument detection for our heuristic. However, since detecting such tokens accurately will take a considerable amount of processing time, we exclude it from our consideration and just simply focus on the regular ones.

Several examples of tokens for preparing arguments in Bytecodes can be seen in Table III where each example is assigned with a unique ID that starts with *C*. These examples are generated to provide a brief explanation about why our proposed heuristic will work accurately on most cases. Firstly, for handling C01 and C02, our heuristic will accurately

TABLE III
THE EXAMPLES OF TOKENS FOR PREPARING ARGUMENTS

ID	Method Invocation	Generated Bytecode Tokens for Preparing Arguments
C01	`foo();` {static method that is invoked on its own class}	
C02	`Class.foo();` {static method that is invoked on other class}	
C03	`obj.foo();` {object method}	`ref_load`
C04	`foo(5);` {static method with an argument}	`numeric_const`
C05	`foo(5,3);` {static method with two arguments}	`numeric_const, numeric_const`
C06	`foo(a,b);` {static method with primitive arguments}	`primitive_load, primitive_load`
C07	`foo(a,b);` {static method with reference arguments}	`ref_load, ref_load`
C08	`foo(a,b-2);` {static method with arithmetic operation as its argument}	`primitive_load, primitive_load numeric_const, substraction`
C09	`foo(foo2(x));` {static method with another static method invocation as its argument}	`primitive_load, invoke_method_foo2, primitive_load`
C10	`foo(new Object());` {static method with object initialization as its argument}	`new, dup, invoke_constructor`

remove no token since both cases invoke static method without parameter. Secondly, for handling C03, our heuristic will accurately remove one token since invoked method is an object method (i.e. non-static method). Thirdly, for handling C04-C07, our heuristic will accurately remove one, two, two, and two tokens respectively according to the number of explicit method parameters. Finally, for handling C08-C10, our heuristic will generate inaccurate number of removed arguments. It will remove two, two, and one tokens respectively, even though each method generates more tokens for preparing arguments due to their additional operation. C08 generates 4 tokens since it requires to subtract *b* with 2; C09 generates 3 tokens since it requires to invoke *foo2*; and C10 generates 3 tokens since it requires to create a new object. We would argue that such inaccuracy is natural since these three cases incorporate additional operation while preparing their argument.

Even though our heuristic is only able to accurately remove tokens on seven of ten cases described in Table III, we would argue that our heuristic is still considerably effective in practice. We believe that the last three cases are seldom used for obfuscating source code plagiarism since implementing these attacks requires high programming skill, which is not owned by most plagiarists.

G. Invoked Method Removal

This phase is responsible to remove the contents of invoked methods from low-level tokens. Such mechanism aims to speed up processing time on similarity measurement by reducing the number of compared token. It is applied by marking all methods that have been invoked on method linearization, the 4th phase of low-level token extraction, and remove contents related to these methods from low-level tokens. It is important to note that such mechanism will not affect the completeness of extracted tokens since the contents of invoked methods are implicitly defined on invoker method as a result of method linearization.

H. Low-level Similarity Measurement

Low-level similarity measurement defines similarity degree by summing local similarities resulted from paired method contents. We do not rely only on main method for measuring similarity since not all programming assignments are featured with main method (e.g. programming assignment to complete methods on abstract data type). In general, our similarity measurement works in threefold: 1) Low-level tokens are grouped per containing method before comparison; 2) Methods from both codes will be paired to each other based on their method signature similarity; and 3) Similarity value is resulted by comparing the content for each method pair and merging the result.

Firstly, low-level tokens for each source code are grouped per their respective containing method before comparison. Our work does not compare the whole tokens directly to speed up processing time. Similarity algorithm in this work, which is RK-GST, takes $(max(N,M))^3$ processes where *N* and *M* are the number of tokens in compared source codes. Thus, splitting tokens into smaller sub-sequences based on containing method will reduce processing time significantly since $\sum_{i=1}^{p}(max(Ni,Mi))^3 < (max(N,M))^3$ where p is the number of methods; $\sum_{i=1}^{p} Ni = N$; and $\sum_{i=1}^{p} Mi = M$.

Secondly, methods from both codes will be paired to each other based on their method signature similarity. This mechanism is applied to speed up processing time by assuming that most programming assignments force students to follow a particular structure such as mandatory class and method names. In general, our proposed pairing mechanism works in twofold: 1) All possible pairs are ranked in descending order based on their Inverse Levensthein Distance (ILD). ILD is calculated as in (1) where *dist(A,B)* refers to Levensthein distance between both strings, *A* is the first method signature, and *B* is the second method signature. In such manner, the number of differences between both signatures will be inversely proportional to ILD. A method pair which members share similar signature will be assigned with a high score; and 2) Method pairs which member has been occurred on higher rank are removed to avoid redundant member on selected pairs.

$$ILD(A,B) = \frac{1}{dist(A,B)+1} \quad (1)$$

Finally, after all method pairs with the most similar signature have been selected, similarity degree is defined based on (2) where *A* and *B* refer to compared codes; *length(A)* and *length(B)* refer to the number of tokens in *A* and *B* respectively; *Pairs* refers to selected method pairs; and *s(Pa,Pb)* refers to the number of matched tokens between *Pa* and *Pb* that is resulted from RK-GST algorithm. It is important to note that, in our work, two tokens are only considered as similar to each other if these tokens share similar mnemonic and weighting constant (i.e. constants that have been generated from flow-based token weighting, the 2nd phase of low-level token extraction).

$$sim(A,B) = \frac{\sum_{P \in Pairs} s(Pa,Pb)}{\sum_{P \in Pairs} min(length(Pa), length(Pb))} \quad (2)$$

In fact, our similarity measurement is adapted from Karnalim's similarity measurement [11], an extended version of minimum matching similarity [45] that is resistant against dummy-based plagiarism attacks (e.g. incorporating dummy statements, methods, or classes). However, we apply two modifications on such measurement which are: 1) Method signature similarity in Karnalim's work [11] is replaced with ILD. Even though both mechanisms yield similar behavior, we believe that ILD is more intuitive since ILD score is proportional to signature similarity. The higher its score, the more similar both signatures will be. It is different with Karnalim's signature similarity where its score is inversely proportional to signature similarity. The lower its score, the more similar both signatures will be; and 2) A stricter rule is applied for token similarity. Two tokens are considered as similar to each other *iff* their mnemonic and flow-based weighting constants are similar.

IV. Controlled Effectiveness Evaluation

This section aims to revalidate the advantages of our proposed approach in a controlled environment. In general,

there are 11 advantages that will be evaluated. Eight of them are adopted advantages of Karnalim's approach [11], a predecessor of our approach, while the other three are our new advantages, which are flow-based token weighting, argument removal heuristic, and invoked method removal. For each advantage, artificial test case(s) which exploit such advantage will be carefully designed and used to check whether such advantage is prominent or not.

A. Evaluating the Advantages of Karnalim's Approach

Based on Karnalim's work [11], we have extracted 8 advantages of his approach which details including its controlled evaluation cases can be seen on Table IV. For convenience, each advantage will be assigned with a unique ID that starts with KAR and each evaluation case will be referred as *EKAR* + advantage ID + case number. All evaluation cases are generated based on Karnalim's original source codes that were used to generate plagiarism attacks [11]. These codes are extracted from Liang's book [52] and cover 7 programming materials which are Output, Input, Branching, Loop, Method, Array, and Matrix. They will be referred as ORIG1 to ORIG7 respectively.

In order to evaluate adopted advantages of KArnalim's Approach (KAA), the effectiveness of such approach will be compared to Standard Lexical Token approach (SLT), a state-of-the-art approach for detecting source code plagiarism. SLT works by converting both source codes into lexical token sequences, removing the comments, and comparing them to each other using a particular similarity algorithm. However, to simplify result justification in this evaluation, our SLT will use similar similarity algorithm with KAA. It will use minimum matching similarity [45], using RK-GST algorithm with 2 as its MML.

In terms of measuring effectiveness, according to the fact that low-level representation (in our case, Bytecodes) yields fewer tokens than the source code itself [11, 4], comparing both scenarios based on normalized similarity may be unfair. One mismatched token on low-level approach (i.e. KAA) may lower its normalized similarity significantly due to its short token length. Therefore, a new similarity measurement which does not rely heavily on token length is proposed for comparison purpose. It is called Inverse number of Mismatched Token (IMT) and resulted from (3). In general, IMT works by negating the number of Mismatched Token (MT) from both sequences (*A* and *B*). It will yield a non-positive integer as its output which is ranged from $-\infty$ to 0. The higher IMT value generated from comparing two sequences, the more similar these sequences are. However, it is important to note that zero IMT (the highest possible IMT value) does not mean that both sequences are exactly similar. It only means that all tokens from shorter sequence are found on the longer one, as it is known that MT for each case is generated by subtracting the minimum length of both sequences with the number of matched tokens.

$$IMT(A,B) = -1 * MT(A,B) \quad (3)$$

Our work does not incorporate division-based inverse mechanism (e.g. inverse mechanism as in our ILD) since we intend to generate IMT distribution that has similar pattern as in MT. In division-based mechanism, the distance between points is not uniform and gets smaller when the given values are extremely large. For example, suppose there are 3 values which are 1, 2, and 3. If inversed in division-based manner, delta value between the 1st and 2nd inversed value (1/1 - 1/2 = 1 - 0.5 = 0.5) will be not uniform with delta value between the 2nd and 3rd inversed value (1/2 - 1/3 = 0.5 - 0.33 = 0.17), even though the distance of their original delta value is uniform. Such inversing mechanism is quite different with our inverse mechanism, which only negates the MT, since negation operation will generate similar distance between points as in MT regardless of point's original position.

IMT result for each evaluation case from Table IV can be seen in Fig. 3. Vertical axis represents IMT value for each case whereas horizontal axis represents the evaluation cases. In general, KAA tends to generate higher or equal IMT than SLT

TABLE IV
KARNALIM'S ADVANTAGES WITH THEIR CONTROLLED EVALUATION CASES

ID	Advantage	Evaluation Cases
KAR1	Proposed approach is not affected by whitespace modification	By considering ORIG7 as original code, plagiarized code is generated by removing all whitespaces. This case will be referred as EKAR11.
KAR2	Proposed approach is not affected by comment modification	By considering ORIG7 as original code, plagiarized codes are generated by adding inline comment for each statement. The number of inline comment will be increased per evaluation case from one to four per statement. These cases will be referred as EKAR21 to EKAR24 respectively
KAR3	Proposed approach is not affected by delimiter modification	By considering ORIG7 as original code, plagiarized codes are generated by adding extra semicolon for each statement. The number of semicolon will be increased per evaluation case from one to four per statement. These cases will be referred as EKAR31 to EKAR34 respectively.
KAR4	Proposed approach, at some extent, is not affected by identifier renaming	For each Karnalim's original source code except ORIG1, plagiarized codes are generated by renaming all variable and method names. These cases will be referred as EKAR41 to EKAR46 respectively.
KAR5	Proposed approach is not affected by semantically-similar syntactic sugar replacement	By considering ORIG4 as original code, plagiarized codes are generated by replacing *while* syntax with either *for* or *do while* syntax. These cases will be referred as EKAR51 and EKAR52 respectively.
KAR6	Proposed approach handles inlining and outlining method	Original code is ORIG3 with all local variables are replaced with the global ones. Plagiarized codes are generated by encapsulating statements into nested methods, starting from a simple method to 3-level nested methods. These cases will be referred as EKAR61 to EKAR63 respectively.
KAR7	Proposed approach ignores dummy methods	Original code is ORIG3 with all local variables are replaced with the global ones. Plagiarized codes are generated by adding dummy method(s). The number of dummy method(s) will be increased per evaluation case from one to three methods. These cases will be referred as EKAR71 to EKAR73 respectively.
KAR8	Proposed approach ignores dummy global variables	By considering ORIG3 as original source code, plagiarized codes are generated by adding global object variable(s). The number of global object variable(s) will be increased per evaluation case from one to three variables. These cases will be referred as EKAR81 to EKAR83 respectively.

in all cases. Therefore, it can be roughly stated that there is no contradicting view about Karnalim's advantages mentioned in Table IV. Karnalim's approach is, in general, more effective than state-of-the-art approach (SLT) when evaluated in controlled environment.

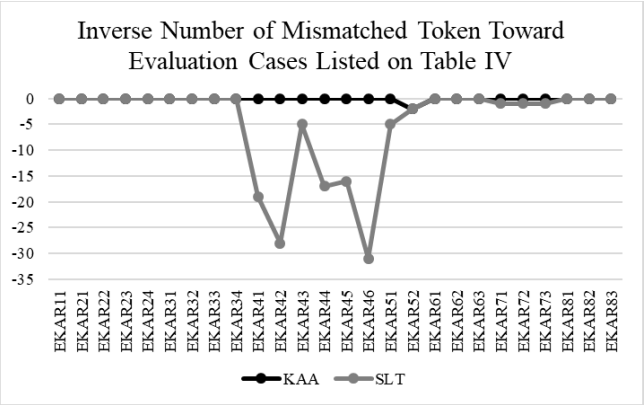

Fig. 3. Inverse Number of Mismatched Token Toward Evaluation Cases Listed on Table IV

From cases about the first three advantages (EKAR11 to EKAR34), both KAA and SLT yield zero IMT on all cases. They are resistible to whitespace, comment, and delimiter modification, regardless of how many modifications are involved. On the one hand, KAA is resistible to such modifications since it deducts similarity based on low-level form (i.e. Bytecodes). It has excluded whitespace, comment, and delimiter during compilation phase. On the other hand, SLT is resistible to such modifications due to its excluding mechanism and minimum matching similarity. Whitespace and comment are handled by excluding them at conversion phase whereas extra delimiters are handled by incorporating minimum matching similarity that ignores extra tokens. Even though both scenarios yield similar result, we would argue that KAA is more effective than SLT for handling such modification since KAA's excluding mechanism is implemented automatically in programming language compiler. It need no additional effort for reimplementation.

From cases about identifier renaming (EKAR41 to EKAR46), KAA is resistible to such attacks due to Bytecode's local variable renaming and method signature similarity. Bytecode's local variable renaming will rename all local variables with technical names according to their sequential order when the source code is compiled into Bytecodes. Such mechanism enables KAA to handle identifier renaming on local variables since both original and renamed variable name will be replaced with similar technical name as long as they firstly occur in similar ordinal position. Method signature similarity, on the contrary, will consider two approximately-similar method signatures as similar to each other as long as such pair yields the highest ILD among related pairs. Such mechanism, at some extent, could handle small modification on method name, especially in our case. An evidence for this case is KAA result on EKAR44, a case which plagiarized code involves method name modification. It still generates zero IMT since the method name is only modified by adding a character as its additional postfix.

As seen in Fig. 3, SLT seems fluctuated in EKAR41 to EKAR46 since SLT considers each renamed identifier as a mismatched token and the number of renamed identifier in each case varies. SLT fails to consider renamed identifier as a match with its original form since it only distinguishes identifier based on its mnemonic. Such issue, however, is handled in KAA with local variable renaming and method signature similarity. Thus, according to such finding, it can be roughly stated that KAA is more effective than SLT for handling identifier renaming.

From cases about semantically-similar syntactic sugar replacement (EKAR51 and EKAR52), KAA outperforms SLT in general since it yields higher IMT on one case while generating similar IMT on the other one. On the one hand, in EKAR51, KAA generates zero IMT since both *while* and *for* syntax generate similar bytecode sequence due to their similar semantic. Such high similarity, however, cannot be achieved by SLT since SLT determines token similarity based on source code form and both *while* and *for* syntax share different form. On the other hand, in EKAR52, KAA cannot generate zero IMT since *while* and *do-while* syntax do not share identical semantic. By definition, both syntaxes actually generate different control flow. *while* syntax checks the condition before performing the action while *do-while* syntax do the action once before checking the condition for future iteration. Despite such unidentical generated control flow, KAA is still as effective as SLT for handling such replacement.

From cases about inlining and outlining method (EKAR61 to EKAR63), both KAA and SLT yield zero IMT in all cases. In other words, method encapsulation is handled well by both scenarios, regardless of how many nested method encapsulations are involved. SLT is able to generate high IMT on such cases since it ignores pattern context at comparison phase. It could detect similar pattern from any location of given source codes. In our cases (EKAR61 to EKAR63), SLT matches tokens from main method to tokens from encapsulating method, resulting no mismatched tokens (i.e. zero IMT). Such zero IMT is also achieved by KAA even though it only determines similarity locally per method. When observed further, KAA can generate such result thanks to its method linearization. For each method body, method linearization will replace each method invocation with its respective method content. Consequently, even though several instructions are encapsulated as a method and located separately outside the compared method, KAA is still able to consider encapsulated tokens as a part of compared method body. We would argue that such phenomenon makes KAA becomes more beneficial than SLT for handling method encapsulation since it guarantees that the matched sequences are from similar context.

From cases about dummy methods (EKAR71 to EKAR73), KAA generates zero IMT for all cases since it only determines similarity from paired methods and ignores all dummy methods that has no matching pair in the original code. SLT, on the contrary, should also yield similar result thanks to minimum matching similarity. However, it generates -1 IMT in all cases due to RK-GST limitation. A matched subsequence with length 1 will be considered as a mismatch since its length is shorter than MML (which is 2 in our case). Therefore, we would argue that KAA is more beneficial than SLT for handling dummy methods since it outperforms SLT on such cases.

From cases about dummy global variables (EKAR81 to EKAR83), both KAA and SLT yield zero IMT in all cases.

Thus, it can be stated that dummy global variables have no impact on both scenarios. On the one hand, KAA is resistible to such attack since it only considers method content for comparison. It automatically ignores all global variables. On the other hand, SLT is resistible to such attack due to minimum matching similarity. Using such matching similarity, dummy global variables will be considered as extra tokens that will be excluded before comparison. Despite similar IMT results, we would argue that KAA is more beneficial than SLT for handling dummy global variables since it excludes such tokens before comparison phase, resulting shorter token sequences for comparison.

By and large, it can be roughly stated that declared advantages of KAA are prominent when compared to SLT. However, it is important to note that not all prominences are explicitly shown as higher IMT than SLT. Some of them are shown implicitly by providing more beneficial characteristics such as contextual similarity or faster processing time.

B. *Evaluating the Advantages of Flow-based Token Weighting*

Flow-based token weighting aims to enhance the sensitivity of Karnalim's approach by considering control flow weight while comparing token. Two tokens are only considered as similar to each other *iff* its mnemonic and control flow weight constants are exactly similar. To evaluate such sensitivity, we incorporate two scenarios based on the existence of such weighting mechanism. The first one, which is referred as Weighted Low-Level approach (WLL), refers to our proposed approach in this paper whereas the second one, which is referred as Unweighted Low-Level approach (ULL), refers to the proposed approach without flow-based token weighting. We could state that flow-based token weighting enhances the sensitivity of low-level approach *iff* WLL outperforms ULL in terms of the number of Mismatched Token (MT) toward our controlled evaluation cases, which only modification is about changing token scope.

In general, controlled evaluation cases are generated based on 7 original source codes that were used to generate plagiarism attacks in Karnalim's work [11]. For each original source code, plagiarized codes will be generated in twofold. On the one hand, some of them will be generated by moving each variable declaration gradually to larger scope where each movement is considered as a new plagiarized code. It will start with the first variable declared in the source code and continue to other variable right after the first variable reaches the global scope. Evaluation cases generated in this manner are expected to evaluate the sensitivity of flow-based token weighting toward gradual slight token scope modification. These cases will be referred as EFLA cases. On the other hand, some of them will be generated by encapsulating the content of each method with a 5-times-iteration traversal, that is represented as a *for* syntax. Each original source code will generate one plagiarized code through such mechanism. Evaluation cases generated in this manner are expected to evaluate the sensitivity of flow-based token weighting toward numerous token scope modifications at once. These cases will be referred as EFLB cases.

The detail of generated evaluation cases per original source code can be seen in Table V. In general, there are 29 evaluation cases where 22 of them are generated based on moving variable declaration (EFLA cases) and the rest of them are generated based on 5-times-iteration traversal encapsulation (EFLB cases). It is important to note that EFLA cases are generated unevenly per original source code since the number of declared variables in each original source code varies.

MT result for EFLA cases from Table V can be seen in Fig. 4. Vertical axis represents MT value for each case whereas horizontal axis represents EFLA cases. In most cases, both WLL and ULL generate zero MT, even though WLL is expected to generate higher MT than ULL. Such phenomenon is natural since most EFLA modifications are about moving in-method outermost variable declaration to global variable (i.e. class attribute). Therefore, since WLL assumes method scope as the most outside influenced layer and global variables are automatically included on such layer, WLL is insensitive to such modifications, resulting similar MT as in ULL. Nevertheless, WLL still outperforms ULL in 6 cases, which are EFLA52, EFLA53, EFLA73, EFLA74, EFLA75, and EFLA76, since it can detect the movement of a variable as a mismatched token. In these cases, a variable declaration is moved from loop body to its larger scope. Thus, since WLL will generate different weight constants for token in and out of a loop, WLL will consider moved variable declaration as different token when compared to its original form.

According to the fact that plagiarized code for EFLA cases are generated by moving variable declarations gradually per scope, each case should generate higher MT than its predecessor as long as both cases are originated from similar source code. However, as seen in Fig. 4, some of them still generate similar MT with its predecessor. Each approach has its own cause for such phenomenon. On the one hand, in WLL, some cases generate similar MT with its predecessor since the number of scope-modified token on both cases are similar, considering the fact that moving variable declaration does not always change the number of scope-modified token,

TABLE V
ORIGINAL SOURCE CODES WITH THEIR GENERATED EVALUATION CASES

Original Source Code	EFLA cases	EFLB Cases
ORIG2	0 case since there is no variable declaration involved on such task.	1 case which will be referred as EFLB1.
ORIG2	4 cases which will be referred as EFLA21 to EFLA24 respectively.	1 case which will be referred as EFLB2.
ORIG3	5 cases which will be referred as EFLA31 to EFLA35 respectively.	1 case which will be referred as EFLB3.
ORIG4	1 case which will be referred as EFLA41.	1 case which will be referred as EFLB4.
ORIG5	3 cases which will be referred as EFLA51 to EFLA53 respectively.	1 case which will be referred as EFLB5.
ORIG6	3 cases which will be referred as EFLA61 to EFLA63 respectively.	1 case which will be referred as EFLB6.
ORIG7	6 cases which will be referred as EFLA71 to EFLA76 respectively.	1 case which will be referred as EFLB7.

especially when such variable has been moved before. EFLA53, with EFLA52 as its predecessor, is an example of such phenomenon. On the other hand, in ULL, some cases generate similar MT with its predecessor since variable declaration involved in these cases generates more than one token in low-level form and such tokens can be accurately detected through RK-GST algorithm. EFLA22, with EFLA21 as its predecessor, is an example of such phenomenon.

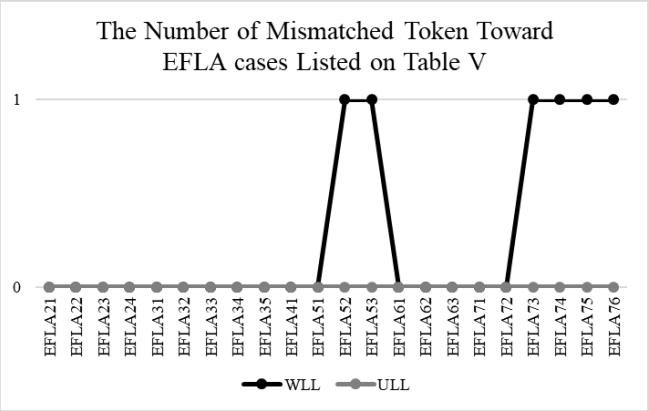

Fig. 4. The Number of Mismatched Token Toward EFLA cases Listed on Table V

MT result for EFLB cases from Table V can be seen in Fig. 5. Vertical axis represents MT value for each case whereas horizontal axis represents EFLB cases. ULL generates zero MT on all cases since it ignores scope modification. It considers token similarity only based on token mnemonic. Therefore, since EFLB plagiarized codes are generated by changing token's scope through control flow weight, ULL cannot differentiate plagiarized code from its original code. WLL, on the contrary, generates fluctuated MT results since the number of scope-modified tokens per case is varied and all scope-modified tokens will be considered as mismatches to their original form due to different flow-based weight constants. In our case, all tokens inside 5-times-iteration traversal in plagiarized code are considered as mismatches to its paired token in original code since such traversal changes the number of containing loop of given tokens. It is important to note that incorporated 5-times-traversal on plagiarized code is not considered as mismatched tokens on both WLL and ULL due to minimum matching similarity that automatically excludes extra tokens from comparison.

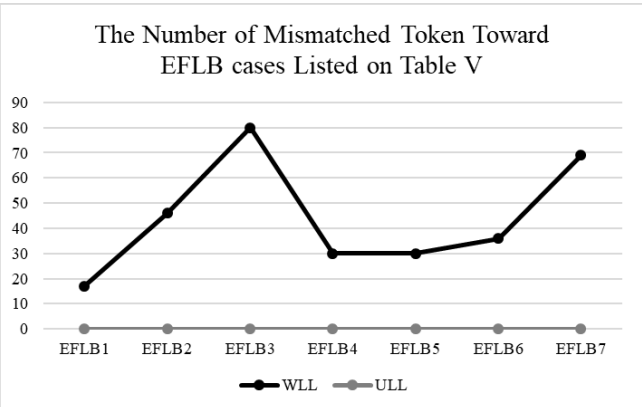

Fig. 5. The Number of Mismatched Token Toward EFLB cases Listed on Table V

To sum up, it can be roughly stated that flow-based token weighting could enhance the sensitiveness of low-level approach since WLL, an approach with such weighting, can detect token scope modification in most cases while ULL, an approach without such weighting, cannot detect given modification in all cases. We would argue that such sensitiveness is important to reduce the number of false positive plagiarism result, as it is known that tokens in different context should not be considered as similar to each other, even though they share similar mnemonics.

C. Evaluating the Advantages of Argument Removal Heuristic

Argument removal heuristic aims to enhance the effectiveness of method linearization by removing remained argument-preparation tokens for each method invocation. To evaluate such mechanism, controlled evaluation cases which plagiarism attack is about encapsulating statement(s) as method(s) with various parameter(s) are generated. We expect our heuristic to effectively exclude most remained argument-preparation tokens on these cases.

In general, our controlled evaluation cases consist of 14 cases which are categorized into twofold: cases to simulate the increasing number of argument and cases to simulate argument preparation on various method invocations. On the one hand, cases to simulate the increasing number of argument consist of 4 cases, namely EARG01 to EARG04 respectively. These cases are generated from ORIG3, a Karnalim's original source code which covers branching material, where its respective plagiarism attack is generated by encapsulating partial statements as a static method with zero to three primitive-type parameters respectively. On the other hand, cases to simulate argument preparation on various method invocations consist of 10 cases, namely EARG05 to EARG14 respectively. These cases are generated from *Hello World* program where each case is designed to simulate method invocations defined in Table III (C1-C10) respectively.

For each case, we will check how many argument-preparation tokens are removed as a result of incorporating argument removal heuristic. Two scenarios will be taken into consideration which are a scenario that involves argument removal heuristic and a scenario that does not involve it. Both scenarios are adapted from our proposed approach which involves no flow-weighting mechanism, namely Unweighted Low-Level approach (ULL). In fact, the result of given heuristic is not affected by flow-weighting mechanism. Yet, we choose to ignore such weighting mechanism so that the reader could easily understand the heuristic's benefit without being confused with other aspects.

The number of removed argument-preparation token for each evaluation case can be seen in Fig. 6. Vertical axis represents the number of removed argument-preparation token whereas horizontal axis represents the evaluated cases. In general, the behavior of argument removal heuristic matches perfectly with our expectation. It accurately removes most argument-preparation tokens, resulting fewer tokens involved in comparison phase.

The results of EARG01 to EARG04 show that increasing number of argument can be handled with our heuristic. It accurately removes argument-preparation tokens as many as the number of argument. It removes no token at EARG01,

which involves no argument, and three tokens at EARG04, which involves three arguments.

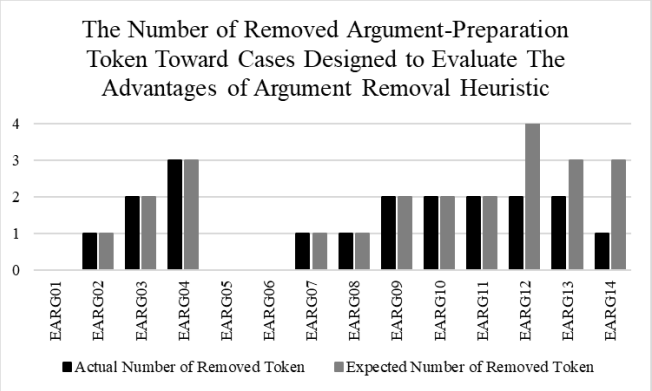

Fig. 6. The Number of Removed Argument-Preparation Token Toward Cases Designed to Evaluate The Advantages of Argument Removal Heuristic

The results of EARG05 to EARG14 show that our heuristic works as expected toward method invocations defined in Table III. It can accurately remove argument-preparation tokens from 7 cases (EARG05 to EARG11) while removing only some of them on the remaining 3 cases (EARG12 to EARG14). On the one hand, our heuristic can accurately remove such tokens on 7 cases since these cases involve no additional operation while preparing the arguments. These cases only generate one argument-preparation token for each involved argument. On the other hand, our heuristic can only remove some argument-preparation tokens on remaining 3 cases since these cases involve additional operation while preparing the arguments. EARG12 put a reduction operation on the 2nd argument; EARG13 put a method invocation as its argument; and EARG14 put an object creation as its argument. The existence of such additional operations violates our heuristic assumption which claims that the number of tokens for preparing argument will be in-sync with the number of method parameter. Therefore, it is natural that our heuristic cannot work properly on such cases.

By and large, according to our controlled evaluation, two findings can be deducted. On the one hand, argument removal heuristic can remove argument-preparation tokens correctly as long as there is no additional operation involved while preparing the arguments. These tokens can be removed regardless of its form, either a direct constant, a primitive variable, or even a reference variable. On the other hand, incorporating additional operations in arguments might lead our heuristic to remove only some of the argument-preparation tokens. As it is known that most additional operations will generate more than one token for each argument while our heuristic only removes one token per argument. According to these findings, we would argue that our heuristic could enhance the effectiveness of method linearization since most argument-preparation tokens are removed through given mechanism.

D. Evaluating the Advantages of Invoked Method Removal

Invoked method removal aims to speed up processing time on similarity measurement by removing the content of all invoked methods from low-level tokens. To evaluate such mechanism, controlled evaluation cases which plagiarism attack is about encapsulating main-method statement(s) as method(s) are generated. For each case, its plagiarized code is generated by encapsulating statement(s) on ORIG3 as method(s) where one statement will be encapsulated as one method. The number of encapsulated statement per case is varied from 1 to 10 where each number will be assigned exclusively to one case and each case will be referred as *EIMR* + the number of encapsulated statement. From these evaluation cases, we expect invoked method removal to accurately exclude the content of additional methods found on plagiarized source code from low level tokens, considering the fact that these methods have been invoked on the main method.

The number of removed invoked methods toward our controlled evaluation cases can be seen on Fig. 7. It is clear that invoked method removal works as expected since it accurately removes all additional methods from plagiarized code. Both actual and expected number of removed method in all cases are similar. We believe that such high accuracy could cut up processing time on similarity measurement since not all method content will be compared through RK-GST algorithm. Some of them will be excluded prior to comparison due to our invoked method removal mechanism.

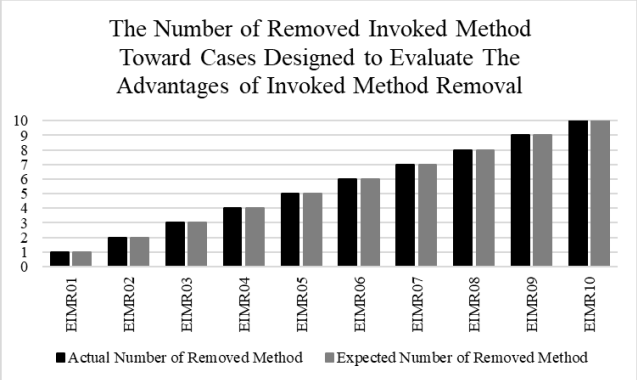

Fig. 7. The Number of Removed Invoked Method Toward Cases Designed to Evaluate The Advantages of Invoked Method Removal

V. EMPIRICAL EFFECTIVENESS EVALUATION

This section aims to revalidate the advantages of our proposed approach in an empirical environment. In general, there are 11 advantages that will be evaluated in this section. The first eight advantages are adopted from Karnalim's approach [11] whereas the others are our new contributions. Each advantage will be evaluated based on empirical dataset which plagiarism attack favors such advantage.

The detail of each advantage with its favoring plagiarism attack and original source code can be seen on Table VI. Each advantage will be featured with a unique ID that is prefixed with *ADV* and its original source code will be taken from Karnalim's original source codes [11]. For each advantage, its original source code will be plagiarized by 11 lecturer assistants by incorporating its favoring plagiarism attack. While plagiarizing the code, lecturer assistants should not change source code semantic. However, they could use other attacks if needed, as long as such attacks are used to support predefined favoring plagiarism attack. It is also worth to note that since the 6th and the 11th advantage refer to exactly-similar plagiarism attack, both of them will be merged as one

TABLE VI
THE ADVANTAGES OF PROPOSED APPROACH WITH THEIR FAVORING PLAGIARISM ATTACK AND ORIGINAL SOURCE CODE

ID	Advantage	Plagiarism Attack	Original Source Code
ADV01	Proposed approach is not affected by whitespace modification	Modify source code indentation	ORIG7
ADV02	Proposed approach is not affected by comment modification	Modify source code comments	ORIG7
ADV03	Proposed approach is not affected by delimiter modification	Modify source code delimiters	ORIG7
ADV04	Proposed approach, at some extent, is not affected by identifier renaming	Modify source code identifier names	ORIG7
ADV05	Proposed approach is not affected by semantically-similar syntactic sugar replacement	Replace syntactic sugar with other semantically-similar form	ORIG4
ADV06	Proposed approach handles inlining and outlining method	Encapsulate source code fragments	Original code in EKAR63
ADV07	Proposed approach ignores dummy methods	Add dummy methods	Original code in EKAR73
ADV08	Proposed approach ignores dummy global variables	Add dummy global variables	ORIG3
ADV09	Proposed approach is sensitive to control flow change	Modify instruction scope	ORIG7
ADV10	Proposed approach, at some extent, enhances the effectiveness of method linearization by removing argument-preparation tokens	Encapsulate instructions as a method with numerous parameters	Original code in EARG14
ADV11	Proposed approach enhances the efficiency of similarity measurement by removing the content of all invoked methods before comparison	Encapsulate source code fragments	Original code in EKAR63

plagiarism attack, resulting only 10 plagiarism attacks that will be used by each lecturer assistant.

In total, there are 110 evaluation cases that will be used in this evaluation. Such cases are classified to 10 plagiarism attacks wherein each attack is conducted by 11 lecturer assistants. For easy reference at the rest of this paper, each case will be assigned with an ID that is formed from the concatenation of *E*, advantage ID, and lecturer assistant ID. For instance, EADV0102 means that such case is generated for ADV01 advantage by lecturer assistant with 02 as his/her ID.

Before used as our dataset, plagiarism cases collected from lecturer assistants are observed manually to ensure whether such cases follow given instructions or not. As a result, 8 cases are removed since their main attack is not in-sync with our request. These cases are EADV0302, EADV0502, EADV0801, EADV0802, EADV0807, EADV0809, EADV0811, and EADV0904.

A. *Evaluating the 1st and 8th Advantage: Adopted Advantages from Karnalim's Approach*

In order to evaluate the first eight advantages, which are adopted from Karnalim's approach [11], two scenarios are proposed which are WLL and SLT. WLL refers to our proposed approach while SLT refers to state-of-the-art approach that was used in our controlled evaluation to evaluate the impact of Karnalim's approach (KAA). For each advantage, its existence is proved to be prominent *iff* WLL generates higher Inverse number of Mismatched Token (IMT) than SLT. If both approaches generate similar IMT, such advantage is only considered to be prominent *iff* WLL has at least one implicit benefit when compared to SLT.

Evaluation analysis provided in this section will be presented per advantage where each advantage usually covers up to 11 plagiarism cases. It will be started from the 1st advantage, which is about whitespace modification, to the 8th advantage, which is about dummy global variable. Beside displaying the result of each advantage toward given cases, we will also provide a brief description regarding to the characteristic of implemented attacks and discuss their impact toward our advantages. Such information is expected to give a clear view toward how our advantages work to the reader.

From cases about whitespace modification (EADV0101 to EADV0111), we found that involved plagiarism attacks can be roughly classified into threefold which are: 1) Stripping all removable whitespaces; 2) Replacing each whitespace with multiple whitespaces or vice versa; and 3) Adding or removing whitespaces unevenly. According to our manual observation toward given cases, the latter attack is the most frequent one to occur. We believe that such high occurrence is natural since such attack is the fastest one to be conducted when compared to other whitespace-based attacks.

Both WLL and SLT generates zero IMT in all cases about whitespace modification. Such finding is supported by the fact that whitespace is automatically excluded by both approaches, resulting no effect to generated IMT. However, we would argue that the 1st advantage, which claims that proposed approach is not affected by whitespace modification, is prominent despite similar IMT result for both approaches since WLL's whitespace removal mechanism is conducted automatically at compilation phase, resulting no additional effort for reimplementation.

From cases about comment modification (EADV0201 to EADV0211), we found that most plagiarism attacks are about changing natural-language terms used in given comment, adding new comments, or removing old comments. Only a few of them are about changing comment format from one-lined to multiple-lined form or vice versa. We believe that such finding is natural since changing comment format requires technical knowledge about how to write source code comment. It is far more difficult to be conducted when compared to other comment-based attacks.

Both WLL and SLT still generates zero IMT in all cases about comment modification since, similar with whitespace, comment is also excluded automatically before comparison for both approaches. Nevertheless, despite similar IMT result for both approaches, we would argue that the 2nd advantage, which claims that proposed approach is not affected by comment modification, is prominent since WLL applies such

removal mechanism automatically at compilation phase, resulting no additional effort for reimplementation.

From cases about delimiter modification (EADV0301 to EADV0311), most plagiarism attacks are focused on inserting semicolon, bracket, and/or curly bracket in various position. From our perspective, inserting semicolon is the most obvious attack since semicolon has no other function except that for separating statement and putting it not at the end of source code statement will be seen as an obvious attempt to plagiarize. It is quite different with inserting bracket or curly bracket where the plagiarists could argue that such delimiters are used by them to categorize the statements in clearer manner.

IMT result for both WLL and SLT toward cases about delimiter modification can be seen in Fig. 8. WLL is not affected by such modification since all delimiters will be excluded at compilation phase. It generates zero IMT in all cases. SLT, on the contrary, is considerably affected by such modification since some plagiarized codes split original tokens into several single tokens by inserting additional delimiter between them. The split tokens will not be detected as a match by SLT's RK-GST algorithm since their respective length (which is 1) is lower than RK-GST minimum matching length (which is 2). The most extreme result generated by such mechanism is found on EADV0305, a case which generates the lowest IMT. In such case, delimiters are embedded on all possible positions to split original source code tokens, resulting 6 single original tokens that are detected as mismatches by SLT's RK-GST algorithm. In short, it can be roughly stated that the 3rd advantage, which claims that proposed approach is not affected by delimiter modification, is prominent since WLL generates higher IMT than SLT on some cases.

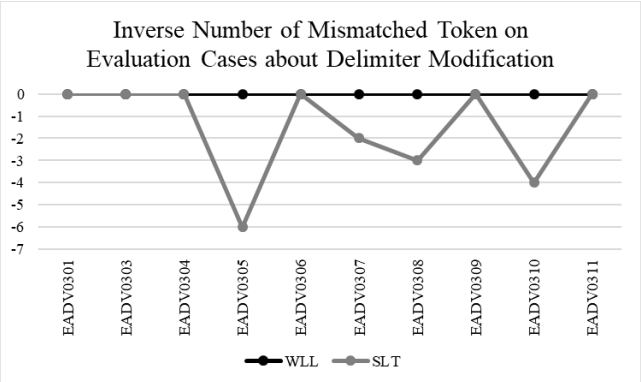

Fig. 8. The Inverse Number of Mismatched Token on Evaluation Cases about Delimiter Modification

From cases about identifier renaming (EADV0401 to EADV0411), most plagiarism attacks are focused on renaming method and variable identifier. Some plagiarists rename it with a slight modification, such as adding a supplementary prefix character for each identifier, while the others put a tremendous modification, such as replacing existing identifier name with an extremely long name. However, regardless of its form, all identifier renaming will only affect IMT based on the number of renamed identifier, as it is known that both approaches do not consider the number of character-based modification on their comparison metric.

IMT result for both WLL and SLT toward cases about identifier renaming can be seen in Fig. 9. WLL generates zero IMT in all cases since it is not affected by identifier renaming involved on given cases. All renaming mechanism only targets local variable and method, which are handled quite well by WLL through Bytecode's local variable renaming and method signature similarity. Bytecode's local variable handles local variable renaming while method signature similarity handles method renaming. Both of them work in similar fashion as in KAA. SLT, on the contrary, generates fluctuated IMT since it only determines identifier similarity in a naïve manner. Two identifiers are considered as similar to each other *iff* both identifiers share similar mnemonic. Consequently, each renamed identifier will be considered as different token when compared to its original token, resulting numerous mismatched tokens. SLT generates the worst result on EADV0403, EADV0404, EADV0407, and EADV0409 since, in these cases, all renamable identifiers are renamed, resulting -31 IMT when measured using SLT. In short, it can be roughly stated that the 4th advantage, which claims that proposed approach, at some extent, is not affected by identifier renaming, is prominent since WLL generates higher IMT in these cases.

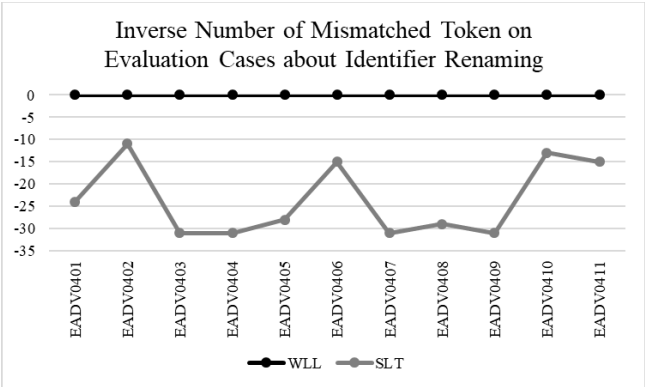

Fig. 9. The Inverse Number of Mismatched Token on Evaluation Cases about Identifier Renaming

From cases about syntactic sugar replacement (EADV0501 to EADV0511), converting a *while* loop to either *for* or *do-while* loop is occurred on all cases. Such finding is natural since it is the most obvious replacement that can be conducted on given code. In some cases, such conversion is often featured with either changing increment form (e.g. *a++;* to *a=a+1*) or variable renaming to obfuscate plagiarized code further. It is important to note that, despite the fact that variable renaming is not a syntactic-sugar-replacement attack, the plagiarists argue that such attack is needed to smooth up the relevancy between involved variable name with newly-introduced loop on their plagiarized code.

IMT result for both WLL and SLT toward cases about syntactic sugar replacement can be seen in Fig. 10. WLL generates zero IMT in all cases since it, at some extent, converts syntactic sugars to their initial form and excludes some mismatched tokens through minimum matching similarity. Both mechanism might remove most possible mismatches from comparison. SLT, on the contrary, generates fluctuated IMT toward these cases since it cannot handle syntactic-sugar replacement and variable renaming, as it is known that SLT compares token only based on its mnemonic

without considering other aspects such as token semantic or variable occurrence order. Among given cases, SLT generates the worst result on EADV0501 since such case combines loop conversion, iterator increment change, and variable renaming at once as its attack. Such combination generates numerous mismatched tokens, resulting -12 IMT when measured using SLT. In short, it can be roughly stated that the 5th advantage, which claims that proposed approach is not affected by semantically-similar syntactic sugar replacement, is prominent since WLL generates higher IMT than SLT in some cases.

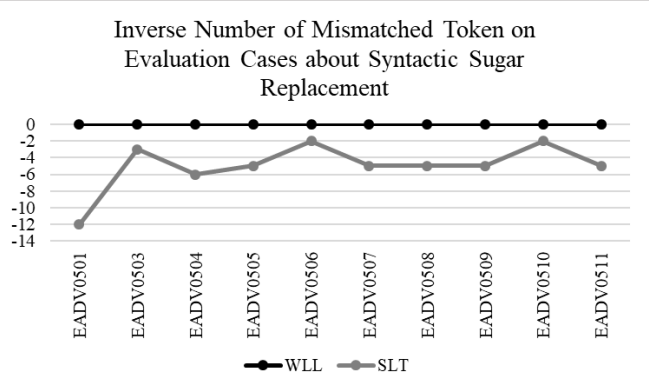

Fig. 10. The Inverse Number of Mismatched Token on Evaluation Cases about Syntactic Sugar Replacement

From cases about encapsulating source code fragments (EADV0601 to EADV0611), all plagiarism attacks are about encapsulating statements as methods where the only variation is about the encapsulation scope. It starts with the largest scope, the whole main-method statements, to the narrowest one, a single statement.

IMT result for both WLL and SLT toward cases about encapsulating source code fragments can be seen in Fig. 11. WLL generates higher IMT than SLT on half cases thanks to method linearization and argument removal heuristic. When combined, both mechanisms, at some extent, could replace each method invocation with its respective method content without leaving argument-preparation tokens. As a result, it could match method invocation with its method content, resolving issues caused by plagiarism attacks about inlining and outlining method. Nevertheless, on the other half cases, WLL generates lower or similar IMT to SLT. When observed further, such phenomenon is caused by twofold: 1) WLL's argument removal heuristic does not always yield accurate result, especially for handling argument that uses additional operation. Therefore, it could generate lower IMT for WLL and, sometimes, such IMT underperforms SLT's IMT; and 2) SLT could detect similar pattern from any location of given source codes since it ignores token context during comparison. Tokens from main method will be matched with tokens from encapsulating method even though both of them has no direct relation. Consequently, it could generate higher IMT for SLT, and, sometimes, such IMT outperforms WLL's IMT. In short, it can be roughly stated that the 6th advantage, which claims that proposed approach handles inlining and outlining method, is prominent since WLL generates higher IMT than SLT in half cases.

From cases about dummy methods (EADV0701 to EADV0711), we found that dummy methods can be roughly classified into twofold: relevant and irrelevant dummy method. Relevant dummy method refers to uninvoked method that has similar context with main program. For instance, uninvoked method about power of two that is found on calculator program. Irrelevant dummy method, on the contrary, refers to uninvoked method that has different context with main program. For instance, uninvoked method about calculating box volume that is found on calculator program. We would argue that the first category is preferred to be conducted on real plagiarism task since it will be less obvious to be accused as a plagiarism case. The plagiarists could argue that they misread the problem instruction and assumes that such methods are needed.

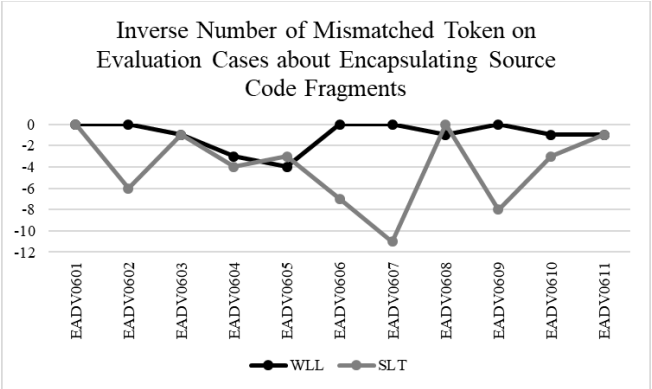

Fig. 11. The Inverse Number of Mismatched Token on Evaluation Cases about Encapsulating Source Code Fragments

IMT result for both WLL and SLT toward cases about dummy methods can be seen in Fig. 12. WLL generates higher IMT than SLT in most cases since it ignores dummy methods from comparison. WLL determines similarity based on comparing methods with the most similar signature. Therefore, since dummy methods are only found on plagiarized code, they will have no pair on original code, resulting no impact toward WLL's IMT. SLT, on the contrary, might be slightly affected by dummy methods even though it applies RK-GST algorithm which could detect similar sub-sequence regardless of its position. When observed further, SLT could generate -1 IMT on these cases since, on plagiarized code, most dummy methods have the copy of main method's statements as their content and main method is not placed as the first method. Both modification might confuse RK-GST algorithm when applied together since such algorithm compares tokens from upper-left to bottom-right sequentially regardless of their context. Some tokens from main method in original code might be considered as a match with tokens from dummy method in plagiarized code, resulting mismatches for tokens from main method in plagiarized code. In short, it can be roughly stated that the 7th advantage, which claims that proposed approach ignores dummy methods, is prominent since WLL generates higher IMT than SLT in most cases and WLL is not affected by method order change.

From cases about dummy global variables (EADV0801 to EADV0811), we found that dummy global variables can be roughly classified into twofold: primitive and non-primitive dummy global variable. Primitive dummy global variable refers to unused global variable with built-in type (e.g. *int* and *float*) whereas non-primitive global variable refers to unused

global variable with reference type (e.g. *String* and *Scanner*). However, regardless of its form, both kinds of variable will only affect IMT based on the number of incorporated variable, as it is known that both approaches do not consider variable type in their comparison metric.

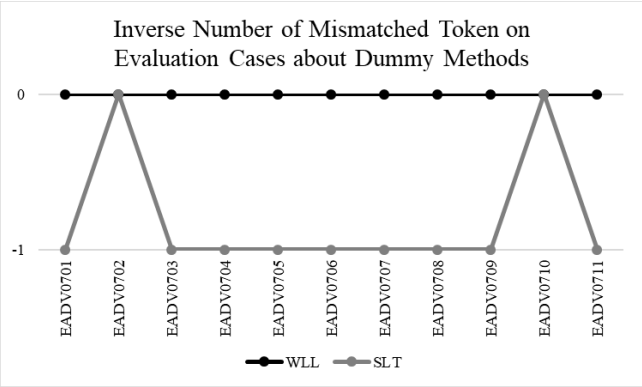

Fig. 12. The Inverse Number of Mismatched Token on Evaluation Cases about Dummy Methods

Both WLL and SLT still generates zero IMT in all cases about dummy global variables since they exclude such variables when determining similarity result. WLL excludes such variables before comparison since it only compares method content whereas SLT excludes such variables during comparison thanks to minimum matching similarity. Despite similar IMT result for both approaches, we would argue that the 8th advantage, which claims that proposed approach ignores dummy global variables, is prominent since WLL excludes such variables before comparison, resulting fewer tokens to be compared than SLT.

To sum up, according to our empirical dataset, it can be stated that the advantages of Karnalim's approach [11] are prominent in our approach since WLL outperforms SLT either explicitly or implicitly on given cases. On the one hand, it outperforms SLT explicitly by generating higher IMT in five advantages, which are the 3rd to 7th advantages. On the other hand, it outperforms SLT implicitly by providing indirect impact, such as automatic mechanism or fewer compared tokens, on remaining three advantages, which are the 1st, 2nd, and 8th advantage.

B. *Evaluating the 9th Advantage: The Sensitiveness of Control Flow Change*

In order to evaluate the 9th advantage, which is about enhancing the sensitiveness of proposed approach toward control flow change, two scenarios are proposed which are WLL and ULL. WLL refers to our proposed approach while ULL refers to WLL without flow-based token weighting. The 9th advantage is proved to be prominent *iff* WLL generates similar IMT with ULL. Such rule is applied based on the fact that original and plagiarized code for each case share similar semantic and flow-based token weighting should not be affected by plagiarism attacks that do not change program semantic.

From cases about control flow change (EADV0901 to EADV0911), we found that all instruction-scope-based attacks are about relocating variable declaration from local (i.e. method variable) to global scope. According to our informal survey, the lecturer assistants argue that such modification is the only instruction-scope-based attack that, at some extent, does not change program semantic.

IMT result for both WLL and ULL toward cases about instruction scope modification can be seen in Fig. 13. Both WLL and ULL generate similar IMT result on all cases. This finding is natural since incorporated attacks are only about moving local to global variables and, according to WLL assumption, such mechanism will not change token weight.

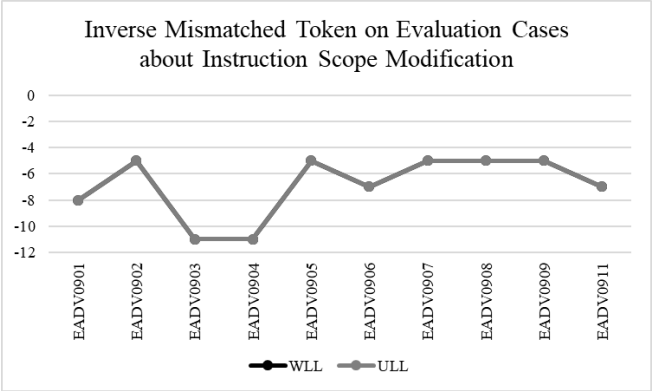

Fig. 13. The Inverse Number of Mismatched Token on Evaluation Cases about Instruction Scope Modification

It is important to note that both WLL and ULL do not generate zero IMT on cases about instruction scope modification since, on these cases, the structure of existing methods is also modified as a part of plagiarism attack. The lecturer assistants argue that such modification is necessary to make the plagiarized code less obvious. They state that global variables, which are generated as a result of instruction scope modification, should be accessed by accessing it directly and such variables should not be passed as method parameter nor return value.

To sum up, since WLL generates similar IMT with ULL, it can be roughly stated that the 9th advantage, which claims that proposed approach is sensitive to control flow change, is prominent.

C. *Evaluating the 10th Advantage: Enhancing the Effectiveness of Method Linearization by Removing Argument-Preparation Tokens*

In order to evaluate the 10th advantage, which is about enhancing the effectiveness of method linearization by removing argument-preparation tokens, two scenarios are proposed which are WLL and WLL-ARG. WLL refers to our proposed approach while WLL-ARG refers to WLL without argument removal heuristic. The 10th advantage is proved to be prominent *iff* total compared tokens on WLL is fewer than total compared tokens on WLL-ARG. Such rule is applied with an assumption that the existence of argument removal heuristic is able to remove argument-preparation tokens, resulting fewer compared tokens on cases about encapsulating instructions as a method with numerous parameters.

From cases about encapsulating instructions as a method with numerous parameters (EADV1001 to EADV1011), incorporated parameters in given plagiarism attacks can be roughly classified into twofold: instruction-related and dummy parameter. On the one hand, instruction-related parameter refers to a parameter that is used as a part of operation in encapsulated instructions. Such parameter usually modifies several instructions so that they can be easily

integrated to given parameter. On the other hand, dummy parameter refers to a parameter that is completely unused in encapsulated instructions. It is only used to obfuscate given plagiarism attack and has no effect on program semantic. However, regardless of its type, both parameter types will be handled in similar fashion with our argument removal heuristic. All tokens for preparing arguments resulted from such parameters will be removed as long as no additional operation is involved.

The number of compared tokens for both WLL and WLL-ARG toward cases about encapsulating instructions as a method with numerous parameters can be seen in Fig. 14. On all cases, WLL generates fewer compared tokens than WLL-ARG. Such finding is natural since each case have at least one argument-preparation token and most of such token will be removed by argument removal heuristic. As seen in Fig. 14, delta value of compared tokens between both scenarios varies per case. It starts with 1 as its lowest delta value to 14 as its highest delta value. On the one hand, the lowest delta value, which is 1, is generated from EADV1001, EADV1002, EADV1005, and EADV1011. Argument removal heuristic only excludes one token on such cases since these cases only use either one instruction-related or dummy parameter on its encapsulating method. On the other, the highest delta value, which is 14, is generated from EADV1008. Argument removal heuristic excludes 14 tokens on given case since it uses 14 instruction-related parameters on its encapsulating method. Such parameters are used to store the subsequences of desired output, which will be concatenated on encapsulating method to generate desired output.

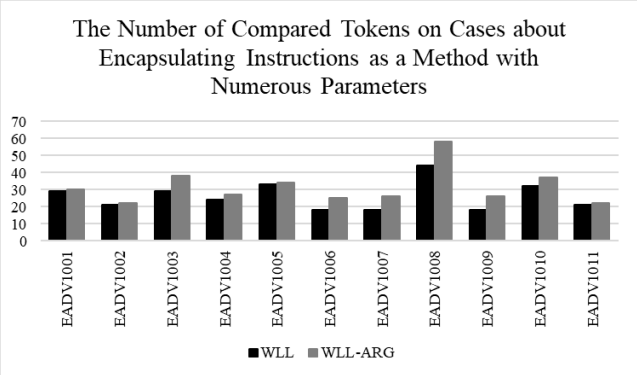

Fig. 14. The Number of Compared Tokens on Cases about Encapsulating Instructions as a Method with Numerous Parameters

To sum up, according to our result, it can be roughly stated that the 10th advantage, which claims that proposed approach, at some extent, enhances the effectiveness of method linearization by removing argument-preparation tokens, is prominent since the number of compared tokens on WLL is fewer than the number of compared tokens on WLL-ARG in all cases.

D. *Evaluating the 11th Advantage: Enhancing the Efficiency of Similarity Measurement by Removing the Content of All Invoked Methods Before Comparison*

In order to evaluate the 11th advantage, which is about enhancing the efficiency of similarity measurement by removing the content of all invoked methods before comparison, two scenarios are proposed which are WLL and WLL-IMR. WLL refers to our proposed approach while WLL-IMR refers to WLL without invoked method removal. The 11th advantage is proved to be prominent *iff* total compared methods on WLL is fewer than total invoked methods on WLL-IMR. Such rule is applied with an assumption that the existence of invoked method removal is able to remove invoked methods form comparison, resulting fewer compared methods on cases about encapsulating source code fragments.

From cases about encapsulating source code fragments (EADV0601 to EADV0611), the number of invoked methods varies per case. It is ranged from 2 to 8 methods with 4.454 methods per case in average. Our invoked method removal is expected to effectively remove all invoked methods listed on both original and plagiarized source code regardless of its number.

The number of compared methods for both WLL and WLL-IMR toward cases about encapsulating source code fragments can be seen in Fig. 15. WLL involves four compared methods per case regardless of its number of initial methods while WLL-IMR involves numerous compared methods regarding to its number of initial methods. On the one hand, WLL involves only four compared methods regardless of its number of initial methods since, on these cases, most methods have been invoked on main method, resulting only four non-invoked methods for comparison. These methods are originated from both original and plagiarized source code where each code contributes two method: main method and implicit constructor. It is important to note that the latter method is automatically generated by Java compiler. Thus, it will always be found on each similarity measurement. However, since its length is considerably short, we would argue that the existence of such method is not significant and we could ignore it for our current work. On the other hand, WLL-IMR involves numerous compared methods regarding to its number of initial methods since it considers all initial methods when determining similarity degree, resulting more processes to be conducted.

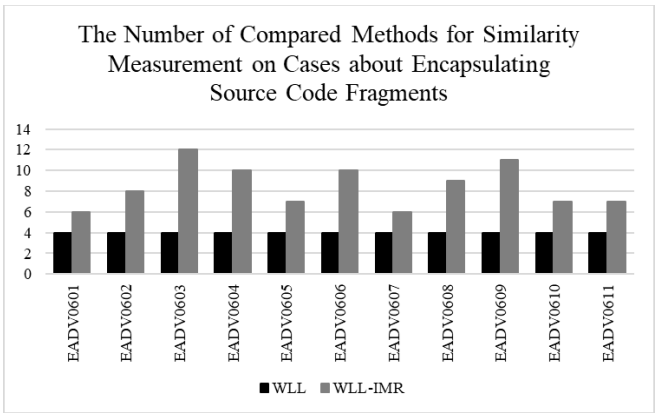

Fig. 15. The Number of Compared Methods on Cases about Encapsulating Source Code Fragments

To sum up, according to our result, it can be roughly stated that the 11th advantage, which claims that proposed approach enhances the efficiency of similarity measurement by removing the content of all invoked methods before comparison, is prominent since the number of compared methods on WLL is fewer than the number of compared methods on WLL-IMR in all cases.

VI. TIME EFFICIENCY EVALUATION

This section aims to measure time efficiency of our proposed approach (WLL) when compared to Karnalim's approach (KAA) and state-of-the-art approach (SLT). Such efficiency will be measured by comparing processing time between involved approaches toward raw dataset that has been used by Karnalim [11] to enlist popular plagiarism attacks on Introductory Programming. It consists of 378 plagiarism pairs which were collected from 9 lecturer assistants and mapped to 6 categories based on Faidhi & Robinson's plagiarism levels [18]. However, since processing source codes on such dataset is considerably fast due to their short number of token; and empirical processing time is inaccurate to capture short execution time, we use Approximate Estimated Time (AET), which is inspired from Rabbani & Karnalim's work [4], as our time efficiency metric. Such metric determines time efficiency based on the number of involved processes instead of empirical processing time, resulting accurate result even for capturing short execution time.

AET for WLL, KAA, and SLT can be seen on (4), (5), and (6) respectively. Firstly, AET for WLL is defined as the total AET for three phases: compilation, extraction, and comparison phase. First, compilation phase takes N_0+M_0 processes where N_0 and M_0 represent the number of source code tokens from compiled source codes. Second, extraction phase takes $(N^2 + 9N) + (M^2 + 9M)$ where N and M represent the number of low-level tokens from extracted executable files. Each executable file takes $(T^2 + 9T)$ for each T tokens where its details can be seen in Table VII. Each phase is featured with its respective assumption which indirectly determines its generated AET based on the worst case. Last, comparison phase, which relies on RK-GST, takes $max(N,M)^3$ processes. Such AET is generated based on RK-GST worst case time complexity. Secondly, AET for KAA is defined in similar manner as in WLL except that it excludes flow-based token weighting, argument removal heuristic, and invoked method removal from extraction phase. Finally, AET for SLT is defined in similar manner as in WLL except that it excludes all low-level extraction processes and replaces N & M with N_0 & M_0 at comparison phase. Such modifications are applied based on the fact that SLT detects similarity based on source code tokens instead of the low-level ones.

$$AET_{WLL}(M,N) = (max(N,M))^3+N^2+M^2+9N+9M+N_0+M_0 \quad (4)$$

$$AET_{ULL}(M,N) = (max(N,M))^3+N^2+M^2+4N+4M+N_0+M_0 \quad (5)$$

$$AET_{SST}(M,N) = (max(N_0,M_0))^3+N_0+M_0 \quad (6)$$

Averaged AET result toward plagiarism-level-focused evaluation dataset can be seen in Fig. 16. Vertical axis represents averaged AET value for each approach per plagiarism level whereas horizontal axis represents Faidhi & Robinson's plagiarism levels [18]. In general, it can be stated that WLL is moderately efficient in terms of its processing time since it generates extremely fewer processes than SLT and slightly more processes than KAA for each plagiarism level. On the one hand, it generates extremely fewer processes than SLT since low-level tokens are far more concise than source code tokens. For most program statements, the number of low-level tokens required to represent such statement is usually fewer than the number of source code tokens required to represent similar statement. Such phenomenon reduces the number of compared tokens on WLL, resulting fewer number of processes when compared to SLT, even though AET for WLL is more complex than AET for SLT in terms of time complexity. On the other hand, it generates slightly more processes than KAA since both approaches accept similar number of tokens per case and AET for WLL is slightly more complex than AET for KAA in terms of time complexity. It takes $5N + 5M$ processes more where N and M are the length of compared token sequences.

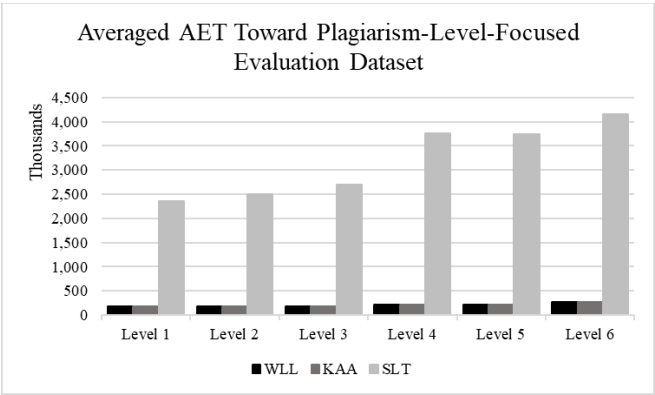

Fig. 16. Averaged Approximate Estimated Time per Plagiarism Level

To sum up, there are two findings that can be deducted which are: 1) Our approach is extremely more efficient than state-of-the-art approach due to the compactness of token representation in low-level form; and 2) Our approach is slightly less efficient than Karnalim's approach due to the existence of flow-based token weighting, argument

TABLE VII
AET FOR EACH EXTRACTION PHASE IN WLL

Extraction Phase	AET	Assumption
Method Content Extraction	T	-
Flow-based Token Weighting	3T	*2T* processes for detecting loops using Tarjan's algorithm by assuming each graph has *T* nodes and *T* edges. *T* processes for assigning flow-based token weight.
Recursive-Method Invocation Elimination	3T	*2T* processes for detecting recursive methods using Tarjan's algorithm by assuming each graph has *T* nodes and *T* edges. *T* processes for eliminating recursive methods by assuming the worst case: each method contains one token and all of them are recursive methods.
Method Linearization	$T^2 + T$	T^2 processes for linearizing methods by assuming the worst case: each method consists of one token and all of them are required to be linearized. *T* processes for argument removal heuristic by assuming the worst case: each existing token on method body should be removed.
Invoked Method Removal	T	Assuming the worst case of invoked method removal: all methods are invoked where each method consists of one token.

removal heuristic, and invoked method removal.

VII. QUALITATIVE EVALUATION

This evaluation aims to qualitatively measure the effectiveness of our advantages in practice. We want to revalidate whether our advantages are useful in practical environment or not. In order to do that, we conduct a survey toward 11 lecturer assistants from Programming courses about our advantages. We assume that the feedbacks of these assistants could represent plagiarism phenomena in practical environment since they should have had numerous experience in terms of detecting source code plagiarism.

A. Survey Questions

Since we assume that each respondent is accustomed to detect plagiarism attack due to their experience, we do not ask directly whether the advantages of our approach are effective or not. Instead, we convert each advantage to a plagiarism attack that favors given advantage and ask them the characteristics of that attack in practical environment. We believe that such conversion could provide more objective result since the respondents could understand the impact of such advantages based on example. Plagiarism attack that will be used to represent each advantage is taken from attack mapping that has been used to generate advantage-focused empirical evaluation dataset. It can be seen on Table VI. In addition to providing plagiarism attacks that favor given advantages, we also provide sample cases which show the impact of given attacks. The sample case for each attack per advantage can be seen on Table VIII. Advantage ID is referenced based on ID convention generated on Table VI whereas each sample case is referenced from advantage-focused controlled evaluation dataset. Since ADV6 and ADV11 share similar attack, both advantages will be represented as one plagiarism attack at once, resulting only 10 plagiarism attacks displayed to the respondents.

TABLE VIII
THE ADVANTAGES OF PROPOSED APPROACH WITH THEIR PLAGIARISM ATTACK AND SAMPLE CASE

Advantage ID	Plagiarism Attack	Sample Case
ADV1	Modify source code indentation	EKAR11
ADV2	Modify source code comments	EKAR24
ADV3	Modify source code delimiters	EKAR34
ADV4	Modify source code identifier names	EKAR46
ADV5	Replace syntactic sugar with other semantically-similar form	EKAR52
ADV6	Encapsulate source code fragments	EKAR63
ADV7	Add dummy methods	EKAR73
ADV8	Add dummy global variables	EKAR83
ADV9	Modify instruction scope	EFLA76
ADV10	Encapsulate instructions as a method with numerous parameters	EARG14
ADV11	Encapsulate source code fragments	EKAR63

For each attack, the respondents are required to measure its semantic similarity, obfuscation, and occurrence degree according to their experience as programming assistants. For clarity, these degrees will be converted to three research questions which are:

a) R1: How similar the semantics between original and plagiarized code if such attack is performed?
b) R2: How obfuscated the plagiarized code if such attack is performed?
c) R3: How frequent is the occurrence of such attack among student's works?

First of all, R1 aims to collect assistant's perspective toward semantic similarity. It aims to check whether similarity in our approach matches human-defined semantic similarity or not. Secondly, R2 aims to measure the impact of plagiarism attacks toward human evaluators. It aims to evaluate whether such attacks could be easily recognized by humans or not. Finally, R3 aims to approximately measure the occurrence of given plagiarism attacks. It aims to check how frequent the given attacks are occurred in student's works. These three research questions will be answered in 5-points scale where each point represents an approximate proportion. 1 refers to less than 20%; 2 refers to greater than 20% and less or equal to 40%; 3 refers to greater than 40% and less or equal to 60%; 4 refers to greater than 60% and less or equal to 80%; and 5 refers to greater than 80% and less or equal to 100%. For each given answer, the respondents are required to provide their own rationale, which will be used for our further analysis.

Each advantage will be considered as a prominent advantage in practical environment if its generated attack gets maximum score on all research questions by all respondents. High score on those aspects means that such attack does not change program semantic, generates perfect obfuscation for human evaluator, and occurs frequently on student's work. In short, such attack is required to be detected by a plagiarism detection system. Therefore, since such attack favors our advantage, it can be stated that our advantage is prominent on practical environment.

B. Respondents

Our respondents consist of 11 undergraduate students who were assigned as lecturer assistants in Programming courses. The statistics of such respondents can be seen in Table IX. There are two findings which can be deducted from given statistic. On the one hand, among these respondents, there are two well-experienced assistants who have assisted numerous Programming courses. Consequently, standard deviation in the first two rows on given table are considerably high. On the other hand, in terms of Grade Average Point (GPA), all respondents have GPA higher than 3.5. Each of them is considered either as a first-class or second-class honored student. Thus, it can be stated that only smart lecturer assistants with high academic merit are selected for this evaluation. We expect such selection could provide more objective result since they could perform in-depth analysis while answering the questionnaire.

C. Responses for R1: How similar the semantics between original and plagiarized code if such attack is performed?

The average semantic similarity degree for each attack per advantage can be seen on Fig. 17. Vertical axis represents average degree for each attack whereas horizontal axis represents plagiarism attacks that are represented with its respective advantage. It is interesting to see that none of the cases yield average value higher or equal to 3. In other words, all cases are roughly considered to have semantic similarity lower than 40%. When observed further, most respondents

were confused with the differences between syntactic and semantic similarity. They tended to consider every modification as a semantic modification. Thus, they provided low score for all cases. We would argue that such finding is natural for our respondents since they had no experience in compiler techniques. Among these respondents, only two of them have some experiences on compiler techniques. They provided high score in most cases. However, the impact of such scores is insignificant when compared to majority scores since each case still yields low average score.

TABLE IX
RESPONDENT STATISTICS

Variable	Min	Max	Average	Standard Deviation
The number of course session which has been assisted	1	30	6.818	8.155
The distinct number of Programming course session which has been assisted	1	5	2.909	1.239
Grade Point Average (GPA)	3.6	3.96	3.826	0.131

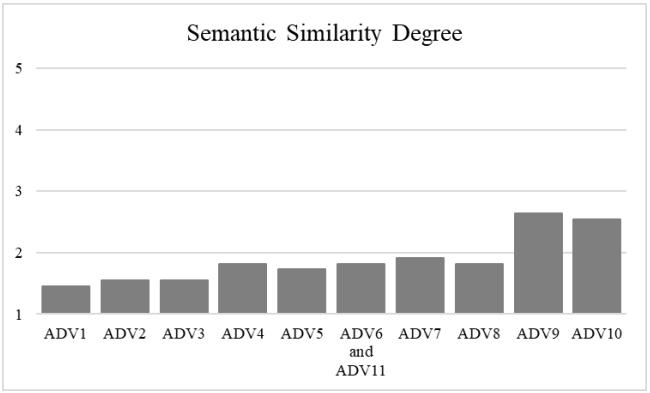

Fig. 17. Average Semantic Similarity Degree

One of the extreme examples regarding to respondent confusion about syntactic and semantic similarity is ADV1 case. It yields the lowest score when compared to other cases even though it involves no semantic modification. It only changes source code indentation which will be discarded at compilation phase. According to the respondent's rationales, they admitted that they had considered such modification as a semantic change. ADV9 case, which gains the highest score, is also affected by such confusion. It gets considerably high score since its modification is insignificant on syntactic level.

In order to get accurate perspectives about semantic similarity of given attacks, we reconduct such survey on similar respondents. The procedure of such survey is quite similar with our initial survey except that the definition of semantic similarity is given comprehensively to respondents before they fill up the survey. The average semantic similarity degree for each attack per advantage can be seen on Fig. 18. Vertical axis represents average degree for each attack whereas horizontal axis represents plagiarism attacks that are represented with its respective advantage. According to given results, ADV1, which was scored with the lowest similarity value on the initial survey, is scored with the highest similarity value (4.09 of 5) on our second survey. Such significant change is natural since, in post-survey, the respondents have understood the definition of semantic similarity. They will only provide low similarity score on source code pairs which modification affects the program flow, such as ADV10. ADV10 gains the lowest score in post-survey since the respondents argued that additional parameters resulted from given case affect the program flow quite significantly when compared to other cases.

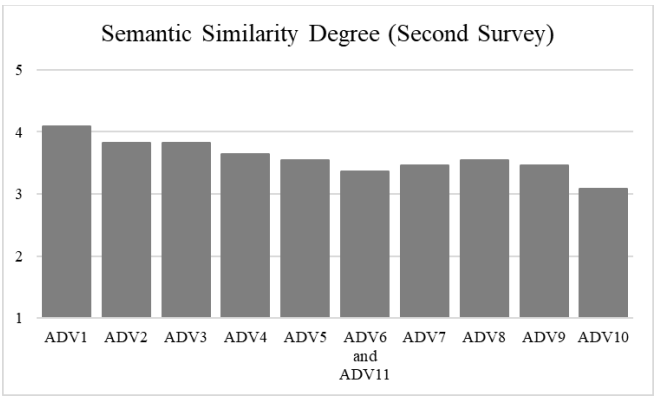

Fig. 18. Average Semantic Similarity Degree Resulted from The Second Survey

D. *Responses for R2: How obfuscated the plagiarized code if such attack is performed?*

The average obfuscation degree for each attack per advantage can be seen on Fig. 19. Vertical axis represents average degree for each attack whereas horizontal axis represents plagiarism attacks that are represented with its respective advantage. None of involved cases yield average result higher or equal to 3. In other words, all cases have obfuscation degree lower than 40%. Therefore, it can be stated that, according to our respondents, plagiarism attacks involved in these cases are not obfuscated enough. They could still be detected through manual observation by the respondents. However, it is important to note that such result is generated based on our respondents who are limited to students with high academic merit. We would argue that resulted obfuscation degrees might be higher if ordinary lecturer assistants were involved.

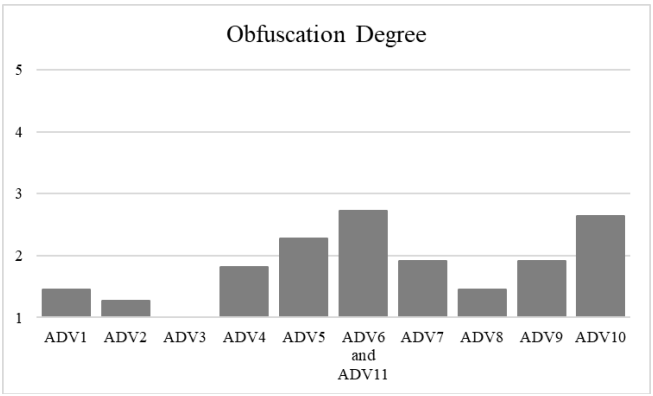

Fig. 19. Average Obfuscation Degree

Among these cases, ADV3 case yields the lowest score, which is 1, since all respondents think that modifying delimiter is an obvious attack. It only slightly intensifies the obfuscation degree while its existence could make the human examiner becomes more suspicious about given code. No programmers intentionally put extra delimiters while solving the problem. Normally, they will focus on source code semantic instead. ADV6 & ADV11 case, on the contrary, yields the highest score since, according to our respondents, encapsulating source code fragments is the most advanced

attack in our evaluated cases. It, at some extent, could change source code layout significantly and can only be done by smart students due to its complexity.

E. *Responses for R3: How frequent is the occurrence of such attack among student's works?*

In fact, this evaluation would become more comprehensive if its findings were deducted empirically from student's programming works, that could be extracted from various programming classes in previous semesters. However, since we do not record such data in our university, we alter our evaluation mechanism to approximately measure it through lecturer assistant's memory and experience about such occurrences.

The average occurrence degree for each attack per advantage can be seen on Fig. 20. Vertical axis represents average degree for each attack whereas horizontal axis represents plagiarism attacks that are represented with its respective advantage. In general, most cases are quite seldom occurred since their score is lower than 3, which means that their occurrences are less than 40%. Among these cases, ADV4 case is the only case which does not follow such trend. It generates 4.091 average score, which means that its occurrence is higher than 80%. According to respondent's rationales, they stated that identifier renaming, which is found on ADV4 case, is popular among students since it is easy to be conducted and less obvious to be suspected as a plagiarism act. As we know, identifier names are purely determined based on human natural language knowledge. Thus, it is harder to claim such modification as a plagiarism attack. The students could easily avoid the accusation by claiming that they thought these names by themselves while creating the code. Even though ADV1, ADV2, and ADV3 case are easier to be conducted, these cases still generate lower obfuscation degree than ADV4 since they are too obvious. They could make the human examiner becomes more suspicious about given code.

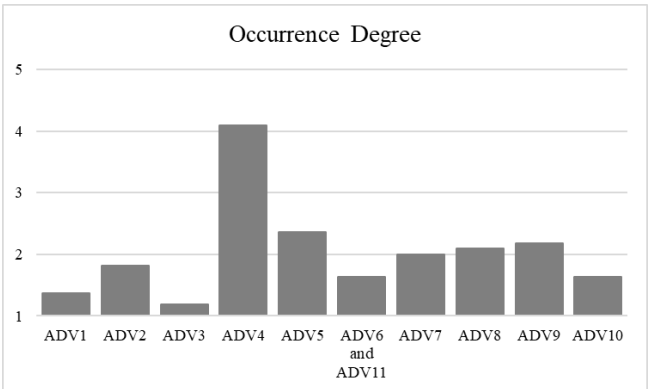

Fig. 20. Average Occurrence Degree

F. *Generalized Result*

According to respondent's feedbacks, several findings can be deducted which are:

a) Plagiarism attacks regarding to our advantages generate low similarity degree when the respondents consider both syntactic and semantic aspect. They will assume every source code modification as a differentiating factor. Such finding is deducted from initial survey result about R1 where all cases are scored lower than 3 in our scale.

b) Plagiarism attacks regarding to our advantages generate high semantic similarity degree since, according to our respondents, these attacks do not significantly change program behavior. Such finding is deducted from second-survey result about R1 where all cases are scored higher than 3 in our scale.

c) Plagiarism attacks regarding to our advantages can be detected by our respondents easily since, according to our respondents, such attacks have low obfuscation degree. Such finding is deducted from the survey result about R2 where all cases are scored lower than 3 in our scale. We would argue that such low results are supported by the fact that our respondents are smart students who have high academic merit. The resulted obfuscation degrees might be higher if ordinary lecturer assistants were involved.

d) Most plagiarism attacks regarding to our advantages are seldom occurred since they are either too complex to be conducted or too obvious to be suspected as a plagiarism attack. Identifier renaming, which is handled by our fourth advantage (ADV4), is the only case which does not follow such trend. According to our respondent's experience, more than 80% plagiarized code pairs contain such attack. These findings are deducted from the survey result about R3 where most cases are scored lower than 3 and only ADV4 case is scored higher than 3.

When perceived based on our approach's advantages, it can be stated that the advantages of our approach target plagiarism attacks that do not significantly change program semantic, generate moderate obfuscation, and generate moderate occurrences on student's works. Thus, we would argue that our approach is moderately effective to handle plagiarism attacks in practical environment.

VIII. CONCLUSION AND FUTURE WORK

In this paper, a low-level structure-based approach for detecting source code plagiarism is proposed. It is extended from Karnalim's work [11] by incorporating flow-based token weighting, argument removal heuristic, and invoked method removal. Flow-based token weighting is intended to reduce the number of false-positive results; argument removal heuristic is intended to generate more-accurate linearized method content; and invoked method removal is intended to fasten processing time. According to our evaluation schemes, three findings can be deducted regarding to our proposed approach. Firstly, the advantages provided by our proposed approach are prominent in both controlled and empirical environment. Secondly, in general, our proposed approach outperforms Karnalim's and state-of-the-art approach in terms of time efficiency. Finally, our approach is moderately effective to handle plagiarism attacks in practical environment.

For further research, our work will be expanded to handle source code plagiarism in object-oriented environment. As we know, most programming tasks nowadays are written in object-oriented fashion and handling object-oriented code is an inevitable task. One of such expansion has been published on [53]. It proposes a naïve approach to linearize abstract method. In addition of such future work, we also intend to

combine attribute-based approach with our approach. Such combination is expected to generate more-accurate plagiarism detection.

REFERENCES

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

[2] S. Hannabuss, "Contested texts: issues of plagiarism," *Library Management*, vol. 22, no. 6, pp. 311-318, 2001.

[3] H. Maurer, F. Kappe and B. Zaka, "Plagiarism - A Survey," *Journal of Universal Computer Sciences*, vol. 12, no. 8, pp. 1050-1084, 2006.

[4] F. S. Rabbani and O. Karnalim, "Detecting Source Code Plagiarism on .NET Programming Languages using Low-level Representation and Adaptive Local Alignment," *Journal of Information and Organizational Sciences*, vol. 41, no. 1, pp. 105-123, 2017.

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

[6] K. Danutama and I. Liem, "Scalable Autograder and LMS Integration," *Procedia Technology*, vol. 11, pp. 388-395, 2013.

[7] J. Hamilton and S. Danicic, "An Evaluation of The Resilience of Static Java Bytecode Watermarks Against Distortive Attacks," *IAENG International Journal of Computer Science*, vol. 38, no. 1, pp. 1-15, 2011.

[8] O. Karnalim and R. Mandala, "Java Archives Search Engine Using Byte Code as Information Source," in *International Conference on Data and Software Engineering (ICODSE)*, Bandung, 2014.

[9] O. Karnalim, "Software Keyphrase Extraction with Domain-Specific Features," in *The 10th International Conference on Advanced Computing and Applications (ACOMP)*, Can Tho, 2016.

[10] S. Alekseev, A. Karoly, D. T. Nguyen and S. Reschke, "Static and Dynamic JVM Operand Stack Visualization And Verification," *IAENG International Journal of Computer Science*, vol. 41, no. 1, pp. 62-71, 2014.

[11] 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.

[12] Z. A. Al-Khanjari, J. A. Fiadhi, R. A. Al-Hinai and N. S. Kutti, "PlagDetect: a Java programming plagiarism detection tool," in *ACM Inroads*, New York, ACM, 2010, pp. 66-71.

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

[14] 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, pp. 8-15, 2007.

[15] K. Ottenstein, "An Algorithmic Approach to the Detection and Prevention of Plagiarism," in *SIGCSE Bulletin*, New York, ACM, 1977, pp. 30-41.

[16] Z. Djuric and D. Gasevic, "A Source Code Similarity System for Plagiarism Detection," *The Computer Journal*, vol. 56, no. 1, pp. 70-86, 2012.

[17] S. Grier, "A Tool that Detects Plagiarism in Pascal Programs," in *The 12th ACM SIGCSE Technical Symposium*, New York, 1981.

[18] 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.

[19] 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, pp. 337-343, 2011.

[20] 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.

[21] 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.

[22] G. Cosma and M. Joy, "Evaluating the performance of LSA for source-code plagiarism detection," *Informatica*, vol. 36, no. 4, pp. 409-424, 2012.

[23] 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, pp. 121-135, 2008.

[24] E. Merlo, "Detection of Plagiarism in University Projects Using Metrics-based Spectral Similarity," in *Dagstuhl Seminar 06301 - Duplication, Redundancy, and Similarity in Software*, 2007.

[25] I. Smeureanu and B. Iancu, "Source Code Plagiarism Detection Method Using Protege Built Ontologies," *Informatica Economica*, vol. 17, no. 3, pp. 75-86, 2013.

[26] 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.

[27] 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, pp. 1016-1038, 2002.

[28] 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.

[29] N. Upreti and R. Kumar, "'CodeAliker' - Plagiarism Detection on the Cloud," *Advanced Computing: An International Journal*, vol. 3, no. 4, pp. 21-26, 2012.

[30] M. Chilowicz, É. Duris and G. Roussel, "Finding Similarities in Source Code Through Factorization," *Electronic Notes in Theoretical Computer Science*, vol. 238, no. 5, pp. 47-62, 2008.

[31] 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.

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

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

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

[35] 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.

[36] M. J. Wise, "YAP3: Improved detection of similarities in computer programs and other texts," *ACM SIGCSE Bulletin*, vol. 28, no. 1, pp. 130-134, 1996.

[37] 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.

[38] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195-197, 1981.

[39] M. E. B. Menai and N. S. Al-Hassoun, "Similarity Detection in Java Programming Assignments," in *The 5th International Conference on Computer Science & Education*, Hefei, 2010.

[40] 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.

[41] 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, pp. 4729-4739, 2011.

[42] M. Mozgovoy, S. Karakovskiy and V. Klyuev, "Fast and Reliable Plagiarism Detection System," in *The 37th ASEE/IEEE Frontiers in Education Conference*, Milwaukee, 2007.

[43] "Welcome to Netbeans," Oracle, [Online]. Available: https://netbeans.org/. [Accessed 23 2 2017].

[44] "Visual Studio | Developer Tools and Services | Microsoft IDE," Microsoft, [Online]. Available: https://www.visualstudio.com/. [Accessed 23 2 2017].

[45] M. J. Wise, "Detection of similarities in student programs: YAP'ing may be preferable to plague'ing," in *Proceedings of the twenty-third SIGCSE technical symposium on Computer science education*, New York, 1992.

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

[47] "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].

[48] "Javassist by jboss-javassist," jboss-javassist, 1999. [Online]. Available: http://jboss-javassist.github.io/javassist/. [Accessed 23 2 2017].

[49] S. Chiba, "Load-time Structural Reflection in Java," in *ECOOP 2000 - Object-Oriented Programming*, Berlin, 2000.

[50] S. Chiba and M. Nishizawa, "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators," in *The 2nd International Conference on Generative Programming and Component Engineering (GPCE '03)*, Berlin, 2003.

[51] R. Tarjan, "Depth-first Search and Linear Graph Algorithm," *SIAM Journal of Computing*, vol. 1, no. 2, pp. 146-160, 1972.

[52] Y. D. Liang, Introduction to Java Programming Comprehensive Version Ninth Edition, Prentice Hall, 2013.

[53] O. Karnalim, "An Abstract Method Linearization for Detecting Source Code Plagiarism in Object-Oriented Environment," in *The 8th International Conference on Software Engineering and Service Science (ICSESS)*, Beijing, 2017.