ITS
Institut
Teknologi
Sepuluh Nopember

ICTS
2016

PROCEEDINGS OF

2016 International Conference on
Information & Communication
Technology and Systems (ICTS)

October 12th, 2016

Department of Informatics, Faculty of Information Technology
Institut Teknologi Sepuluh Nopember (ITS)

TABLE OF CONTENTS

T2: Human-Computer Interaction, Multimedia Application

T3: Pattern Recognition, Intelligent Systems, Knowledge Data Discovery, Bioinformatics/Biomedical Apps, Content-Based Multimedia Retrieval, Remote Sensing, Image Processing, Big Data, Information Retrieval, Robotic, Modelling Simulation, Applied Computing

T4: Software Engineering, Formal Methods, E-Learning, Enterprise Information System, Risk Management, Geographic Information System

Detecting Source Code Plagiarism on Introductory Programming Course Assignments Using a Bytecode Approach

Oscar Karnalim
Faculty of Information Technology
Maranatha Christian University
Bandung, Indonesia
oscar.karnalim@gmail.com

***Abstract*—Even though there are various source code plagiarism detection approaches, most of them only concern with low-level plagiarism attack with an assumption that plagiarism is only conducted by students who are not proficient in programming. However, plagiarism is often conducted not only due to student incapability, but also because of bad time management. Thus, high-level plagiarism attack should be detected and evaluated. This paper proposes source code plagiarism detection approach which can detect most introductory-programming-course plagiarism attacks at any level by utilizing low-level instructions instead of source code tokens. Several mechanisms are also introduced to improve its effectiveness such as instruction generalization, instruction reinterpretation, method-based comparison, and method linearization. Since low-level instruction is a language-dependent feature, Java is selected as target programming language with bytecode as its low-level instruction. Based on evaluation, it can be concluded that our approach is more effective to detect most plagiarism attack types than raw source code approach on introductory programming course. This evaluation is based on plagiarism attack types that are collected through controlled experiment.**

Keywords—source code plagiarism; source code similarity; low-level language; bytecode; plagiarism attack types

I. INTRODUCTION

Source code plagiarism is a major issue which emerges in Programming course [1]. Using modern technology, Students can easily obtain their colleague's coursework, modify it, and then submit it as their own. In order to give more objective result to students, plagiarism should be detected and the plagiarist should be punished. However, detecting plagiarism by hand is quite time-consuming. Programming assignments are usually given every week and each of them consists of at least a dozen of source codes [2]. To overcome this problem, several automatic methods for detecting source code plagiarism are developed.

When detecting plagiarism in source codes, language-specific features are often needed, especially to detect advanced plagiarism attack (e.g. method inlining and syntactic sugar modifications). Most plagiarism detections utilize lexer (and parser) for each programming language in order to exploit language-specific features such as tokens, code patterns, and syntactic sugar translations [2, 3]. Yet, this approach requires more maintenance effort since lexer and parser provided should be up to date with their respective programming language. Moreover, as most programming languages evolve to make programmer more comfortable, more syntactic sugars may be added in future release which yield frequent change in lexer and parser.

On the other hand, several other approaches rely on low-level instruction which is generated as a result of compiling the source code [4, 5]. This method is more beneficial than source code approaches since its structure is less frequently changed. Low-level structure is intended for machine, so that syntactic sugar is not required and all source-code syntactic sugars are automatically translated. Additionally, low-level instruction only considers semantic-preserving instructions (without source-code template and delimiter) which may yield more precise similarity. Based on these reason, this paper introduces a pairwise low-level-based source code plagiarism detection which is focused on Java introductory programming course. Our approach extends Ji *et al* proposed method [4] with several distinctive features. Beside proposed plagiarism detection approach, this paper also enlists several possible plagiarism attack types in Java introductory programming which then will be classified based on Faidhi and Robinson's plagiarism classification [6].

II. RELATED WORKS

Most source code plagiarism detections are intended to be developed as language-independent. This kind of approach generalizes all programming languages and treat them as raw text with an assumption that the most-distinctive plagiarism features are not language-dependent. One of this approach is proposed by Brixtel et al which detect plagiarism by utilizing multiple level plagiarism recognition from character level to corpus level [7]. However, most-distinctive plagiarism features are often found in programming paradigm and syntactic sugars which may not be detected through language-independent approach.

978-1-5090-1381-4/16/$31.00 ©2016 IEEE

Due to that reason, language-specific features are still utilized for detecting source code plagiarism in various degrees. Some of them try as much as possible to keep their approaches as language-independent by enabling developer to incorporate new programming language through several preprocessing phases. These kind of approaches are commonly used in major plagiarism detection methods such as attribute-counting [6], token matching [2, 3], classification [8, 9, 10], latent semantic analysis [11], and clustering [12] method. In spite of its benefit which enable developer to incorporate new programming language, preprocessing phases required by these approaches may not be easy to implement, especially when target language has different and unique patterns. Additionally, language-specific features utilized in these approaches are also limited due to its language-independent behavior.

In order to utilize more language-specific features, several approaches exploit more sophisticated representation such as abstract syntax tree [13, 14, 15, 16], control flow graph [17], and program dependence graph [18]. These approaches are tightly-coupled with their target programming language which means that it requires considerable effort when applied to other programming language. However, these approaches tend to yield more precise result than other approaches due to its language-specific manner.

On the other hand, several different and unique approaches are also developed by utilizing low-level instruction instead of source code [19, 20]. Anjali *et al* exploits execution trace of Java bytecodes to detect plagiarism with an assumption that similarity can be measured through dynamic behavior [19]. This approach is not practical since both bytecodes should be executed which is not efficient in terms of time, especially on NP-complexity bytecodes. Moreover, in programming course assignment, most programs always have similar dynamic behavior due to assignment restriction. Cuomo *et al* exploits Java bytecodes for detecting plagiarism by applying formal method to determine bytecode similarity [20]. Yet, applying formal method may yield time-efficiency drawback which is not suitable for detecting a dozens of plagiarism cases.

Since string matching algorithm is quite effective and efficient for detecting source code plagiarism, several researches treat low-level instructions as token sequences and compare them using string matching algorithm [5, 6]. Juričić utilizes Levensthein distance on instruction mnemonics in order to measure similarity of two Common Intermediate Language (CIL) sequences. Nevertheless, this approach may yield faulty result since mnemonic name and functionality are not always related and similar functionality does not always entail similar mnemonic name. Ji *et al* propose bytecode-based Java plagiarism detection by generalizing bytecode instruction and compare its sequences using adaptive local alignment. They also implement main-method linearization in order to handle method-based plagiarism attack.

In this paper, we extend Ji *et al* proposed method [5] in order to detect source code plagiarism in introductory programming course. We have several distinctive features which are expected to yield more precise result. First, instructions are generalized not only by functionality but also data type re-categorization. Second, special instructions are reinterpreted as regular instructions (e.g. *switch-case* and *jsr* are converted to *goto*). Third, string literals and *goto* information are involved in our plagiarism detection. Forth, method-based comparison proposed in this paper is not only relying on main method but also all methods since not all programming assignments rely on main method. Finally, our method linearization handles recursive method.

III. PLAGIARISM ATTACKS

Since plagiarism detection can be developed more precisely if most possible plagiarism attacks are known and listed, we propose controlled experiment for collecting plagiarism attacks. In controlled experiment, we enlist possible plagiarism attacks based on 378 plagiarized source codes generated by respondents. To collect more varied plagiarism attacks, respondents are limited to students who are proficient at programming from various classes. Additionally, most of them are lecturer assistants who are experienced in detecting student coursework plagiarism. The statistic of our respondents is shown in Table I. Though they are small in number, they may yield more varied plagiarism attacks than numerous average students due to their experiences. Each respondent is asked to plagiarize seven source codes from Liang's textbook [21] which cover introductory programming course materials. Then, each source code should be plagiarized based on six plagiarism levels defined by Faidhi and Robinson [6]. Since there are nine respondents who plagiarize seven source codes to six plagiarized source codes each, this experiment yields 378 plagiarized source codes (9 respondents * 7 source codes * 6 plagiarism levels).

TABLE I. STUDENT RESPONDENT STATISTICS

Class	Number of Respondents	Status
2011	1	Former lecturer assistant
2012	2	Former lecturer assistants
2013	4	Current lecturer assistants
2015	2	-

Beside plagiarism attack types that are extracted from controlled experiment, several augmented attack types are also added which is resulted from our experience as Java programming lecturer. Both types are listed in Table II where augmented types are marked with blue color and all attack types can be occurred in reverse order. L2.5 is a new level defined besides Faidhi and Robinson's plagiarism level and located between L2 and L3. This level is focused in modifying minor supplementary features (e.g. package system in Java). In our perspective, this level is easier than L3 and more difficult than L2.

TABLE II. PLAGIARISM ATTACK TYPES

ID	Level	Attack Type
0001	L0	Verbatim copy
1001	L1	Modify comments and whitespaces
1002		Modify source code delimiter except arithmetic bracket
1003		Modify the usage of bracket in arithmetic operation
2001	L2	Modify identifier name
2501	L2.5	Change the package name and structure
2502		Change package import
2503		Avoid explicit class import by using full class name on each respective class declaration
3001	L3	Declare all variables at the beginning of source code
3002		Assign variable declarations with its default values
3003		Merge two or more variable declarations
3004		Merge variable declaration and assignment
3005		Change local to global variable
3006		Reuse declared variables for other processes
3007		Incorporate dummy variables
3008		Rearrange method declaration
3009		Change access modifier in attributes and methods
4001	L4	Encapsulate the content of main method as particular method and call it on main method as a replacement of its content
4002		Encapsulate particular task as a void method with the use of global variables (no parameter involved)
4003		Encapsulate particular task as a void method without the use of global variables (parameters are involved)
4004		Replace particular task with a non-void method
4005		Incorporate dummy methods
5001	L5	Utilize API-based instruction instead of regular instruction
5002		Break down API-based instruction to several more-specific API-based instructions
5003		Exchange API-based instruction with other API-based instruction that yield similar functionality for particular circumstance
5004		Incorporate useless parameters on API method calls
5005		Replace constant value with variable
5006		Replace constant with arithmetic operation which yield similar result
5007		Change operand order in complex arithmetic operation
5008		Merge several arithmetic operations without the use of temporary variables
5009		Locate increment/decrement instruction as index or method parameters directly instead of treating them as single instruction
5010		Replace increment/decrement instruction with their respective binary operator form
5011		Replace combined assignment with their respective binary operator form
5012		Replace primitive data type with other primitive data type that yield similar functionality for particular circumstance
5013		Replace reference data type with other reference data type that yield similar functionality for particular circumstance
5014		Incorporate useless casting
5015	L5	Replace if-else branching with switch-case
5016		Change loop type
5017		Change array/collection iteration from regular traversal to for-each traversal
6001	L6	Replace a number of repetitive instructions with loop
6002		Change loop boundary
6003		Reverse loop direction from ascending to descending
6004		Convert loop to void recursive method
6005		Convert loop to non-void recursive method
6006		Rearrange branching statements based on its condition validation sequence
6007		Replace logical expression with other expression that yield similar meaning
6008		Incorporate logical expression that can be replaced with boolean constant
6009		Incorporate dummy instructions except logical expression
6010		Incorporate useless assignment in parameter or return of a non-boolean method
6011		Rearrange loosely-coupled instructions

IV. METHODOLOGY

Our source-code-comparison approach consists of four steps: 1) select pairwise method candidates; 2) generalize and reinterpret method instructions; 3) linearize method content; 4) measure code similarity.

A. Select Pairwise Method Candidates

This step takes two project directories as its input and compile all source codes to class files. Project directories is utilized as input since most programming courses rely on sophisticated IDE (e.g. Netbeans and Eclipse). However, since most IDEs store compilation results (classes) on their project directories, compiling is only conducted when a project has no class file. Furthermore, raw token comparison is also applied instead of bytecode comparison if either one or both of the compared projects is uncompilable.

After compilation, all methods are extracted from each class in both projects and compared. Comparison is performed by comparing methods instead of the whole classes due to RKGST complexity, $O(n^3)$. Comparing two classes with 5 instructions in 2 methods (with 2 and 3 method instructions respectively) in method perspective is faster than the whole class ($2^3 + 3^2 < 5^3$). Although our approach relies on method comparison, comparing all method in pairs may be inefficient due to its $O(mn)$ complexity. However, with an assumption that most programming assignments have identifier name patterns defined by their lecturer, the number of pairwise method comparison can be reduced by only comparing methods which have similar identifier name. Utilizing this mechanism may reduce comparison complexity to $min(O(m),O(n))$. Furthermore, an equitable balance of pair distribution is also involved so that each method is only compared once and each pair selected has the most similar identifier name among all possible pairs. This pair distribution mechanism does not only reduce time complexity but also handles

plagiarism attack which is based on dummy methods and attributes.

The algorithm for selecting pairwise method candidates takes two method arrays as its input and return selected method pairs as its result. First, all method are compared in pair and sorted based on pair similarity in ascending order. Pair similarity is determined with Levensthein distance between both method identifier names wherein method identifier name refers to the concatenation of full class name, method name, and method descriptor. Thus, after all pairs are sorted, each pair which member(s) is occurred in more-similar pair is removed.

B. Generalize and Reinterpret Method Instructions

Each method content in selected method pairs is extracted with Javassist [22] and most instructions are either generalized or reinterpreted. Generalization is conducted since most bytecode instructions are over-specific. Several instructions may yield similar purpose and only differ in technical implementation (e.g. *goto* and *goto_w*). Generalization rules can be seen in Fig. 1 which is grouped based on their similar functionality with respect to data type re-categorization. Data type re-categorization split data types into two general categories (primitive and reference type) instead of real data types since data type change is often utilized in plagiarism attack. Furthermore, other opcodes which value is not listed in Fig. 1 are not translated and taken as raw instructions. On the other hand, reinterpretation converts special instructions to regular instructions so that plagiarism attack can be handled more precisely. *jsr*, *jsr_w*,and *ret* are reinterpreted as *goto* sequence which mechanism is adopted from [23]. *tableswitch* and *lookupswitch* are also reinterpreted as *goto* sequence but its mechanism is adopted from [24]. *putstatic* and *putfield* are converted to *primitive_store/reference_store* whereas *getstatic* and *getfield* are converted to *primitive_load/reference_load*. Both conversions are based on its data types.

Generalized Mnemonic	Opcode Value
numeric_const	2-17
primitive_load	21-24, 26-41, 46-49, 51-53
ref_load	25, 42-45, 50
pop	87, 88
addition	96-99
substraction	100-103
multiplication	104-107
division	108-111
remainder	112-115
conversion	133-147
invoke_method	182-185
goto	167, 172-177, 200

Generalized Mnemonic	Opcode Value
load_constant_pool	18-20
primitive_store	54-57, 59-74, 79-82, 84-86
ref_store	58, 83, 75-78
dup	89-91
dup2	92-94
shl	120, 121
shr	122, 123
ushr	124, 125
and	126, 127
or	128, 129
xor	130, 131
conditional_goto	153-166, 198, 199

Fig. 1. Mnemonic Generalization Rules

Furthermore, since *load_constant_pool*, *goto*, and *conditional_goto* are frequently occured in bytecodes, they are concatenated with their respective parameters in order to distinguish them with each other. *load_constant_pool* is concatenated with its string literal due to the fact that string plays important role in program. Moreover, to ignore slight modification in string literal, concatenated string is converted to lowercase with its non-alphanumeric characters removed. On the other hand, *goto* and *conditional_goto* are concatenated with their jump distance so that two instructions with different jump target are not considered similar.

C. Linearize Method Content

After all methods in selected pairs are translated to instruction sequences, each method is linearized to handle L4 plagiarism attack. Linearization is adopted from [24] which implement dynamic programming mechanism. However, our approach is quite different with [24] since we remove recursive method calls on recursive methods instead of limiting recursive method linearization.

D. Measure Code Similarity

Similarity in this paper extends YAP [25] equation with respect to number of method instructions, as in (1). *sim(A,B)* measures similarity of two projects which result is ranged from 0 to 1 inclusively (0 represents not match at all whereas 1 represents exact match). *Pairs* represents selected method pairs resulted from the first step wherein each pair (*p*) consists of one *A* method member (*Pa)* and one *B* method member (*Pb*). *s(Pa,Pb)* is measured using rabin-karp greedy string algorithm (*RKGST*) [26] with 2 as its minimum match length (*MML*) whereas *length(X)* represents the number of instructions in *X*. Utilizing this equation may also handle dummy instructions since the similarity of each method pair is based on method with fewer instructions instead of both methods. Thus, dummy instruction attached to plagiarized project may not affect similarity result. For example, even though project *B* (1 method with 28 instructions) is plagiarized from project *A* (1 method with 23 instructions) by incorporating 5 dummy instructions, its similarity still yield 1 (23/23) since it is divided by the number of instructions in *A* (23).

$$sim(A,B) = \frac{\sum_{P \in Pairs} s(Pa,Pb)}{\sum_{P \in Pairs} min(length(Pa),length(Pb))} \quad (1)$$

V. EVALUATION

Evaluation is conducted by comparing our approach (*B*) with 2 source-code-based approaches which are raw token comparison (*RT*) and method-based raw token comparison (*MRT*). *RT* and *MRT* only differ in token extraction where *RT* takes all tokens avalaible and *MRT* only takes token sequence from method contents. In both approaches, source codes are translated to token sequences where each token is represented by its respective token type instead of its value. Translation is conducted using ANTLR [27] whereas token type is defined based on Java 8 grammar provided by [28]. Both source-code-based approaches adopt similar pairwise candidate selection and similarity measurement that is utilized in *B*. Yet, *MRT* utilizes method-source-code token sequences instead of bytecode instructions and *RT* extends *MRT* by utilizing class pairs instead of method pairs.

In terms of token size, all approaches yield different token size which detail can be seen in Fig. 2. Horizontal axis represents 100 projects from evaluation dataset (50 cases * 2 for each case) whereas vertical axis represents token size of

each project. *RT* yields the largest token size since all tokens are included whereas *MRT* yield less token size than *RT* since *MRT* does not incorporate template tokens (e.g. class header). *B* yield the least token size since *B* only consider semantic-preserving tokens by removing both source-code template and delimiter tokens. As a result of only considering semantic-preserving tokens, *B* tends to yield more-precise result due to its semantic-orientation and faster comparison time due to its limited tokens.

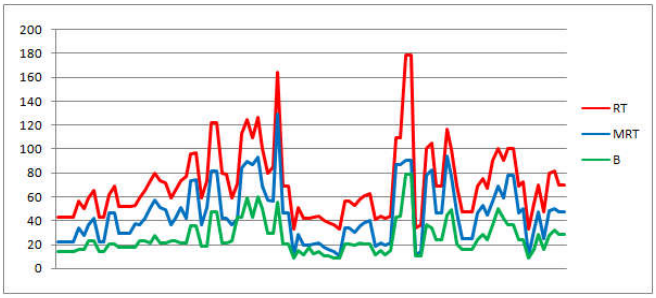

Fig. 2. Token Size Comparison

However, since token size varies for each approach, effectiveness cannot be evaluated by comparing similarity result which is represented as percentage. Therefore, we propose evaluation mechanism that is based on how many mismatched token toward the number of compared tokens. A plagiarism detection approach is considered as resistible to a particular plagiarism attack iff the number of mismatched token between target source code and plagiarized source code is lower or equal with *MML* defined in *RKGST*. This scheme is based on two assumptions which are: 1) each plagiarism case in evaluation data only contains one kind of attack; 2) token sequences with size smaller than *MML* are not detected as plagiarism in *RKGST*.

The number of mismatched tokens for each evaluation case can be seen in Fig. 3 where horizontal axis represents 50 cases from evaluation dataset and vertical axis represents the number of mismatched tokens. As seen in Fig. 3, *RT* and *MRT* results are incosistent due to non-semantic-preserving tokens. Even though *MRT* has more precise result due the absence of template tokens, *MRT* is still affected by source code delimiter tokens. Furthermore, since *MRT* is not featured with method linearization, *MRT* yields worse result than *RT* when detecting L4 plagiarism attack. Uncompared methods in *MRT* is automatically compared in *RT* since method is the member of class file by default. On the other hand, since *B* yields the lowest number of mismatched token among other approaches in most cases, it can be concluded that *B* is the most resistible approach to handle plagiarism attacks. L0-L3 are handled at tokenizing phase in compiler translation whereas L4 is handled by method linearization. Thus, most L5 attacks which are based on syntactic sugar are automatically translated at bytecode level so that it can be detected easily. For high-level plagiarism attacks (half of L5 attack and L6 attack), bytecodes is supposed to be ineffective due to the fact that bytecodes is a technical detailed representation of source code. Small semantic-related change in source code may yield great impact on bytecode structure. However, this impediment can be handled through bytecode generalization and reinterpretation so that only significant information are extracted and all similar instructions are treated as one instruction.

However, several outlier cases arise where *B* has 3 or more mismatched tokens (case 5003, 5015, 6001, 6003, and 6005). Case 5003 replaces *system.out.print* with *system.out.printf* for printing two strings. Even though it only differs in method name at source code level, it affects greatly at bytecode level since parameter type of both method are different. *system.out.print* requires one regular argument whereas *system.out.printf* requires variable length argument which is converted as array in bytecode representation. Case 5003 is an example where bytecode level are less beneficial than source-code level. In case 5015, *B* yields high number of mismatched tokens since switch-case reinterpretation is not exactly similar to if-else statement due to their different behavior and capability. Not all if-else branching can be converted to switch-case and vice versa. However, switch-case reinterpretation yields more similar structure than comparing it as raw instruction (*B* result is still higher than *RT* and *MRT*). On the other hand, case 6001, 6003, and 6005 are quite difficult to detect due to their significant logic change. These cases yield big difference in both source-code and bytecode level.

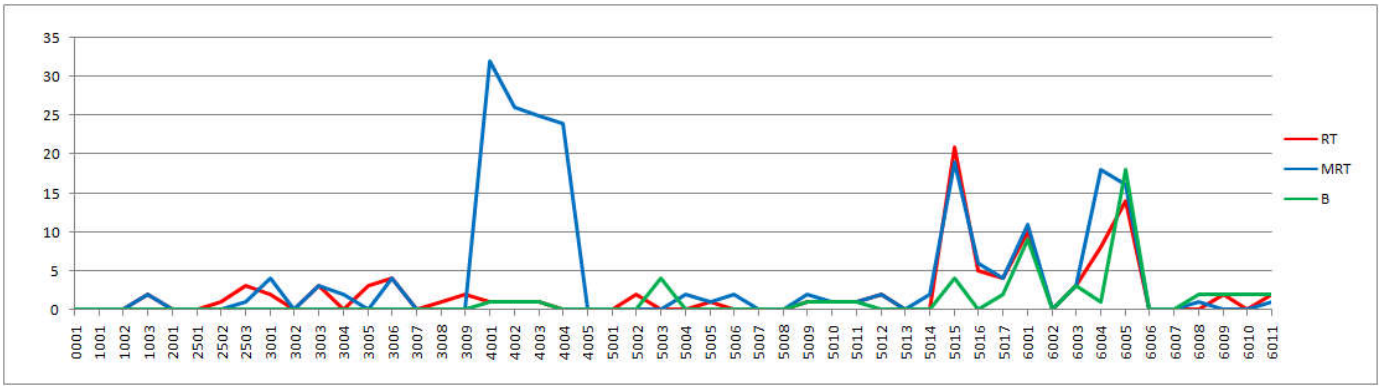

Fig. 3. Mismatched Token Comparison

VI. CONCLUSION

Based on evaluation, our proposed bytecode-based approach is more resistible than raw-token-source-code approach to handle plagiarism attacks (see Fig. 3). Several outlier cases may occur but our approach still yield more compromising result than raw-token-source-code approach. Our proposed approach consists of 4 steps where each step has a particular plagiarism detection mechanism: Pairwise method candidate selection handles dummy methods and attributes by selecting only similar method pairs based on their identifier; Instruction generalization and reinterpretation handles data types, syntactic sugars, and bytecode specificity; Method linearization handles method-based plagiarism attack with the concern of recursive methods; Code similarity handles dummy instructions. Furthermore, since bytecode is the result of compiling source code, most low-level plagiarism attacks (L0-L3) are unavailing since whitespaces and comments are automatically removed and all identifiers are renamed.

VII. FUTURE WORK

In next research, our proposed method will be expanded to handle plagiarism attacks in object-oriented programming. Several advanced programming features are added into our consideration such as exception, inheritance, interface, and polymorphism. Moreover, we will also propose enviroment setting which enable our approach to be implemented in programming course.

REFERENCES

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

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

[3] L. Prechelt, G. Malpohl and M. Philippsen, "Finding plagiarism among a set of programs with JPLag," *Journal of Universal Computer Science,* vol. 8, no. 11, pp. 1016-1038, 2002.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] D. Poongodi and A. G. Tholkkappia, "Multi-Agent based Sequence Algorithm for Detecting Plagiarism and Clones in Java Source Code using Abstract Syntax Tree," *International Journal of Computer Applications,* vol. 90, 2014.

[16] T. Sager, A. Bernstein, M. Pinzger and C. Kiefer, "Detecting similar Java classes using tree algorithms," in *The 2006 international workshop on Mining software repositories*, New York, 2006.

[17] D.-K. Chae, J. Ha, S.-W. Kim, B. Kang and E. G. Im, "Software plagiarism detection: a graph-based approach," in *The 22nd ACM international conference on Information & Knowledge Management*, New York, 2013.

[18] J. Krinke, "Identifying similar code with program dependence graphs," in *Eighth Working Conference on Reverse Engineering*, Stuttgart, 2001.

[19] V. Anjali, T. R. Swapna and B. Jayaraman, "Plagiarism Detection for Java Programs without Source Codes," in *The International Conference on Information and Communication Technologies*, Kochi, 2014.

[20] A. Cuomo, A. Santone and U. Vilano, "A novel approach based on formal methods for clone detection," in *The 6th International Workshop on Software Clones*, 2012.

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

[22] S. Chiba, "Load-Time Structural Reflection in Java," in *14th European Conference Sophia Antipolis and Cannes*, France, 2000.

[23] H. Park, S. Choi, H.-i. Lim and T. Han, "Detecting Code Theft via a Static Instruction Trace Birthmark for Java Methods," in *International Conference on Industrial Informatics*, Daejeon, 2008.

[24] 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.

[25] 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.

[26] M. J. Wise, "Running rabin-karp matching and greedy string tiling," Basser Departement of Computer Science, Sydney University, 1993.

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

[28] "grammars-v4/java8 at master · antlr/grammars-v4 · GitHub," 2016. [Online]. Available: https://github.com/antlr/grammars-v4/tree/master/java8. [Accessed 07 12 2015].

ICTS 2016 Conference Schedule

Wednesday, October 12th, 2016

Time	Agenda
08.00 - 09.00	Registration
09.00 - 09.05	Conference Statement by Conference Chair (Ridho Rahman Hariadi, M.Sc)
09.05 - 09.20	Opening Ceremony by Rector (Prof. Joni Hermana)
09.20 - 10.20	Keynote Speech 1: Prof. Shengdong Zhao
10.20 - 10.35	Coffee Break
10.35 - 11.35	Keynote Speech 2: Prof. Riyanarto Sarno
11.35 - 13.00	Lunch
13.00 - 15.00	Parallel Sessions
15.00 - 15.05	Conference Summary and Closing Ceremony