Python source code plagiarism attacks on introductory programming course assignments

Oscar Karnalim
oscar.karnalim@it.maranatha.edu

Faculty of Information Technology, Maranatha Christian University, Indonesia

Abstract. This paper empirically enlists Python plagiarism attacks that have been found on Introductory Programming course assignments for undergraduate students. According to our observation toward 400 plagiarism-suspected cases, there are 35 plagiarism attacks that have been conducted by students. It starts with comment & whitespace modification as the most frequent attack and ends with replacing regular instruction with API-based instruction as the least frequent one. In addition to such primary finding, we have also found two additional findings. First, when classified based on Faidhi & Robinson's taxonomy, the occurrence trend of such attacks is not proportional to increasing plagiarism level due to the nature of Python programming language, course syllabus, and student preferences. Second, incorporated plagiarism attacks are proportional to student experience, even though such relation is, sometimes, mitigated by student understanding and assignment restrictions.

Keywords: source code plagiarism, plagiarism attack, introductory programming, python

Introduction

Source code plagiarism is an act of generating a source code from another code with a slight modification (Parker & Hamblen, 1989). Despite the fact that such activity is a trivial task for Computer Science (CS) students (Sraka & Kaucic, 2009), detecting such illegal behavior takes a considerable amount of time. The lecturer should check each possible source code pairs and decide which pairs should be accused as plagiarism cases. Consequently, several plagiarism detection systems are developed to alleviate such work (Djuric & Gasevic, 2012; Lim et al., 2011; Rabbani & Karnalim, 2017). Using such system, the lecturer is not required to check each possible pair manually. He/she is only required to revalidate plagiarism suspected pairs generated by such automatic system.

Even though there are numerous plagiarism detection systems available, most of them were only evaluated in black-box manner, which is not sufficient to draw out the characteristic of proposed system comprehensively. To the best of our knowledge, there are only two works which evaluated their respective system without such black-box manner (Prechelt et al., 2002; Karnalim, 2016). Both works evaluated their proposed system by analyzing the impact of their system toward empirically-listed plagiarism attacks. They defined which attacks favor their system and which attacks do not. Prechelt et al. (2002) evaluated their JPlag with plagiarism attacks listed from 647 Java source code pairs whereas Karnalim (2016) evaluated his low-level plagiarism detection system with plagiarism attacks listed from 378 Java source codes.

Python is a popular programming language for learning Introductory Programming due to its simplicity (Guo, 2013). It handles most technical details implicitly, resulting simple yet powerful syntaxes for programmer. Due to such popularity, we intend to propose a Python-targeted plagiarism detection system for education environment. However, before developing such system, we would argue that it is important to understand the characteristic of Python plagiarism attacks beforehand. In this paper, we propose such initial study. We plan to empirically enlist Python plagiarism attacks on Introductory Programming course and analyze its trend from two aspects: Faidhi & Robinson's taxonomy (Faidhi & Robinson, 1987) and student experience. Such study is expected to provide a brief characteristic about Python plagiarism attacks in educational environment, especially in Introductory Programming course.

Literature review

Source code plagiarism is an emerging issue in Computer Science (CS) education, especially in Programming course (Cosma & Joy, 2008). In such course, there are, at least, two causes which encourage students to do plagiarism. First of all, most programming assignments are written electronically. Such circumstance might encourage students to do plagiarism since electronic files can be copied and modified easily in no time, without leaving a particular trace. On the other, most programming assignments are graded automatically by a judge program without human intervention. Such circumstance might encourage students to do plagiarism since plagiarism checker on such judge program is usually less accurate than human evaluator. The students could easily trick such system as long as they know how it works.

To handle such emerging issues, there are numerous source code plagiarism detection systems have been developed. In general, such systems can be roughly classified into two categories which are attribute- and structure-based approach (Al-Khanjari et al., 2010). Attribute-based approach detects plagiarism based on key properties from given source codes whereas structure-based approach detects plagiarism based on source code ordinal structure. However, it is important to note that both approaches are not exclusive to each other. In some works, both approaches are merged together (Engels et al., 2007; Ohno & Murao, 2011), resulting more comprehensive result.

Attribute-based approach uses key properties from source code, such as software metrics, to detect plagiarism (Djuric & Gasevic, 2012; Al-Khanjari et al., 2010; Bandara & Wijayarathna, 2011). Two source codes are considered as plagiarized to each other if, and only if, both codes share similar key properties. The similarity of given key properties itself could be measured using various mechanisms such as information retrieval (Ramirez-de-la-Cruz et al., 2015; Cosma & Joy, 2012), classification (Bandara & Wijayarathna, 2011), and clustering (Jadalla & Elnagar, 2008). However, regardless of its similarity measurement, the effectiveness of attribute-based approach is affected heavily by extracted key properties. When such properties do not sufficiently capture the characteristic of given source code, such approach will not generate accurate result.

Structure-based approach uses ordinal structure from source code, such as token sequence, to detect plagiarism. Such approach usually works in twofold. Firstly, all source codes are translated into their respective intermediate form such as lexical token sequence (Djuric & Gasevic, 2012; Kustanto & Liem, 2009; Lim et al., 2011), compiler-based representation (Chilowicz et al., 2009), or low-level codes (Juričić, 2011; Juricic et al., 2011; Ji et al., 2008; Karnalim, 2016; Rabbani & Karnalim, 2017). Secondly, generated intermediate representation would be compared in pairwise manner through string-based similarity algorithm such as Rabin-Karp Greedy String Tiling (RKGST) (Wise, 1996), Winnowing Algorithm (Schleimer et al., 2003), and Local Alignment (Smith & Waterman, 1981). According to several works (Prechelt et al., 2002; Djuric & Gasevic, 2012), this approach is more effective than attribute-based approach due to its sensitivity. The similarity of two source codes are not only defined by the number of similar properties (which is tokens in this case) but also the order of given properties.

Despite the fact that there are numerous works about plagiarism detection system, to the best of our knowledge, only a small number of them were evaluated based on empirically-listed plagiarism attacks. Most of them were only evaluated in black-box manner, taking a bunch of source codes as a dataset and evaluating the system only based on its general accuracy toward given dataset. Thus, it might be difficult to exploit specific characteristics of given system. The researchers could not declare which attacks are effective and ineffective toward their system. By incorporating empirically-listed plagiarism attacks, the researchers could define the strengths and weaknesses of their proposed system in more comprehensive manner. Moreover, they could also avoid biased result caused by imbalance distribution of given attacks since they could control such distribution explicitly.

In addition to providing a comprehensive metric for evaluation, empirically-listed plagiarism attacks could also aid the researchers to design more effective plagiarism detection system. They could adjust the proposed system for prioritizing popular attacks rather than the rare ones. Moreover, they could also ignore several rare attacks if detecting such attacks take a considerable amount of time.

To the best of our knowledge, previous research enlisted source code plagiarism attacks empirically in an explicit manner (Ahmadzadeh et al., 2011; Prechelt et al., 2002; Karnalim, 2016). Firstly, Ahmadzadeh et al. (2011) enlisted plagiarism attacks occurred on 20 Java source codes and used it to check the tendency of plagiarism attacks among novice students. Their work generated 11 attacks which are varied from indentation to object-oriented modification. Secondly, Prechelt et al. (2002) enlisted plagiarism attacks occurred on 647 Java source code pairs in the evaluation of JPlag. Such mechanism generated 21 attacks which are varied from whitespace to data structure modification. Finally, Karnalim (2016) enlisted plagiarism attacks occurred on 378 Java source codes in the evaluation of low-level plagiarism detection approach. His work generated 50 attacks which are varied from verbatim copy to loosely-coupled instruction rearrangement.

Unfortunately, among these works, there are no contributions which are focused on Python programming language, even though such language is quite popular nowadays for learning Introductory Programming (Guo, 2013). Thus, this paper proposes an empirical study of Python plagiarism attacks found on Introductory Programming course. In addition to such main goal, we also aim to find out the trend of such attacks from two perspectives: Faidhi & Robinson's taxonomy (Faidhi & Robinson, 1987) and student experience. The result of this study is expected to become either an evaluation baseline or a prior knowledge for developing Python-targeted plagiarism detection system.

Methodology

Generally, our methodology aims to enlist Python plagiarism attacks and find out the trend of such attacks toward Faidhi & Robinson's taxonomy (Faidhi & Robinson, 1987) and student experience. Such methodology consists of four phases that should be executed in sequential manner. These phases are raw data collection, plagiarism-suspected pair filtering, manual listing of plagiarism attacks, and trend analysis.

First of all, raw data collection aims to collect all student's source codes that will be used as our dataset. In our case, since we aim to enlist Python plagiarism attacks found on Introductory programming, our dataset is collected from an undergraduate class of Introductory Programming course which was held in the odd semester of 2016/2017 academic year. Such class was conducted in 16 weeks where each week consists of two sessions: Theory and Laboratory session.

The syllabus of both theory and laboratory sessions can be seen on Table 1. According to materials listed on given table, this course covers 2 built-in functions, which are *print* and *input,* and 10 syntaxes, which are *variable assignment, if-then-else, while-do, for-traversal, function declaration, function invocation, static array assignment, static array access, static matrix assignment,* and *static matrix access*. The last four syntaxes, which are about *array* and *matrix*, are taught by assuming that the size of both *array* and *matrix* is static instead of dynamic. *Array* and *matrix* are represented as the multiplication of *[None]* variable instead of a standard Python list. Such modification was applied to accustom the students with default *array* and *matrix* representation on Java and C#, the programming languages that they will learn on the 4th semester. However, it is important to note that we do not restrict our students to only use such functions and syntaxes. They could use other instructions which they have learned outside the class if necessary.

For each laboratory session except the 1st, 7th, and 14th session, laboratory assignment is represented as five sub-assignments that should be completed in 150 minutes. Two of them are related to technical knowledge (e.g. implementing a syntax learned from theory session directly) while the others are related to logical problem solving. For each sub-assignment, we took all student's source codes and treated them as a part of our dataset, resulting 1,428 Python source codes taken from 55 sub-assignments (11 laboratory assignments with 5 sub-assignments each).

Table 1. Course syllabus

Week	Course material of theory session	Course material of laboratory session
1	Introduction and Data Type	Adaptation to Programming Environment
2	Input and Output	Output
3	Branching	Input
4	The 1st Quiz	Branching
5	Traversal	Traversal
6	Nested Traversal	Nested Traversal
7	The 2nd Quiz	Laboratory Mid-Test
8	Mid-Test	-
9	Void Function	Void Function
10	Return Function	Return Function
11	Array	Array
12	The 3rd Quiz	Function and Array
13	Matrix	Matrix
14	Searching and Sorting	Searching and Sorting
15	The 4th Quiz	Laboratory Final Test
16	Final Test	-

Second, plagiarism-suspected pair filtering aims to filter possible source code pairs which one of its member is suspected to be plagiarized from another member. In our case, for each sub-assignment, 10 source code pairs will be randomly taken as a part of our plagiarism-suspected pairs. We prefer to select 10 pairs randomly rather than only taking Top-10 pairs with the highest similarity degree since some plagiarism attacks might not be found on top pairs due to its significant modification. However, to ensure that selected pairs still generate high similarity degree, we only took pairs that generate similarity degree higher than average threshold, a threshold that is resulted by averaging the highest and lowest similarity degree from all source code pairs on that sub-assignment. It is important to note that such threshold is determined locally per sub-assignment instead of being determined globally for all sub-assignments since the number of possible modifications per sub-assignment might be varied. For instance, the *hello world* assignment should enable less modification than other complex assignment such as *sorting*. By determining average similarity locally per sub-assignment, the threshold can be adjusted automatically toward the number of possible modification, resulting fewer false positives and/or negatives for each sub-assignment.

In term of determining similarity degree between two source codes, both source codes will be converted to token sequences and compared to each other using a string similarity algorithm. On the one hand, converting source code to token sequences is conducted using ANTLR (Parr, 2014) with Python 3 grammar provided by ANTLR repository (https://github.com/antlr/grammars-v4/tree/master/python3). It is important to note that, in this phase, source code comments are excluded from generated token sequences so that our measurement is guaranteed to only consider semantic-preserving tokens. On the other hand, comparing string sequences is conducted based on an adaptation of JPlag similarity measurement (Prechelt et al, 2002) which detail can be seen in (1). *sim(A,B)* refers to similarity degree between two compared token sequences namely *A* and *B*; *coverage(A,B)* refers to the total size of shared token subsequences that is generated based on Rabin-Karp Greedy-String-Tiling (RKGST) algorithm (Wise, 1993) with 2 as its minimum match length; and $|A|$ & $|B|$ refer to the length of token sequence *A* and *B* respectively. Such measurement assures that each modification will affect the similarity result.

$$sim\,(A,B) = 2 * coverage\,(A,B) \,/\, (|A|+|B|) \qquad (1)$$

We have acknowledged that not all source code pairs which share high similarity degree are generated from plagiarism acts. Several pairs might be generated due to coincidence, especially when given assignment provides only one logical fashion to solve it. However, since filtering true-positive plagiarism pairs on these pairs might be difficult based on the fact that no students would want to confess their illegal behavior, we have no option but to rely on similarity degree to filter plagiarism pairs. In order to avoid misleading terminology, we refer such pairs as plagiarism-suspected pairs instead of plagiarism pairs. In other words, we do not guarantee that all pairs used in this work are generated from plagiarism acts. Some of them might be generated based on coincidence.

In short, the second phase, which is plagiarism-suspected pair filtering, generates 550 source code pairs from our dataset. It is collected from 55 sub-assignments that cover 11 assignments where, for each sub-assignment, 10 source code pairs which similarity degree is higher than local average similarity degree are taken.

Third, manual listing of plagiarism attacks aims to enlist all plagiarism attacks found on filtered dataset. In our case, such listing is conducted by the first author who has 7 years' experience for detecting source code plagiarism on Introductory Programming course. However, to mitigate the number of observed pairs, our work only considers source code pairs which both members correctly solve given sub-assignments. Such mechanism is applied based on the fact that, according to our informal observation in our course, most students tended to share their code to others if, and only if, they had completed writing the code for solving a sub-assignment. They seldom shared half-completed code to their friends since they were required to complete it first to get higher score. As a result, our work excludes 150 pairs and only considers 400 pairs for manual observation.

In order to simplify our work, we will enlist plagiarism attacks based on the list defined by Karnalim's work (Karnalim, 2016). His work is preferred to other works based on following reasons: 1) Plagiarism attacks enlisted by his work are more comprehensive than other works. It can be seen from the number of generated plagiarism attacks where his work outperforms the others; 2) His work is focused only on attacks found on Introductory Programming course, which is quite similar with our goal; and 3) To the best of our knowledge, his work is the only work which explicitly mapped listed plagiarism attacks based on Faidhi & Robinson's taxonomy (Faidhi & Robinson, 1987). We do believe that the relation between such taxonomy and listed plagiarism attacks might be beneficial for further evaluation about source code plagiarism detection system.

Plagiarism attacks defined by Karnalim (2016) are then modified and adapted based on the manual observation of given dataset. While observing each source code pair, each plagiarism attack will be either mapped to existing attack list defined by Karnalim or considered as a new attack type. Such mechanism is conducted based on the fact that the nature of programming language targeted by Karnalim's work (i.e. Java) is quite different with ours (i.e. Python).

Finally, trend analysis aims to find out the trends of collected attacks based on two perspectives: Faidhi & Robinson's taxonomy and student knowledge. On the one hand, to find out the trend toward Faidhi & Robinson's taxonomy, such attacks are mapped to six levels of Faidhi & Robinson's taxonomy and, according to attack occurrences per level, the relation between attack occurrences and increasing plagiarism level is analyzed. On the other hand, to find out the trend toward student experience, the relation between attack variance per week and student experience is analyzed.

By and large, it can be roughly stated that our methodology will generate three findings which are empirically-listed plagiarism attacks, the trend of collected attacks toward Faidhi & Robinson's taxonomy, and the trend of collected attacks toward student experience. The first finding will be generated on the 3rd phase, which is the manual listing of plagiarism attacks, whereas the other two will be generated on the last phase, which is the trend analysis.

Results and discussion

Empirically-listed plagiarism attacks

According to our proposed methodology, 35 distinctive plagiarism attacks are extracted from 400 plagiarism-suspected pairs on the third phase, which is manual listing of plagiarism attacks. The detail and occurrences of these attacks can be seen on Table 2 where each attack is assigned with a unique ID that starts with *P*. It is important to note that all enlisted attacks work in two-way reversible fashion. For example, if an attack is focused on incorporating a dummy method, then removing dummy method is also considered as that attack. As seen in Table 2, there are five attacks which occur frequently on our dataset. These attacks are P01, P03, P17, P27, and P28.

Table 2. Empirically-listed plagiarism attacks

ID	Attack type	Number of occurrences (pairs)
P01	Modify comment and whitespace	400
P02	Modify source code delimiter	45
P03	Modify identifier name	322
P04	Assign different default value to a variable	40
P05	Merge two or more variable assignments	11
P06	Change the variable scope	28
P07	Reuse declared variables for other processes	5
P08	Incorporate dummy variables	27
P09	Rearrange function declaration	17
P10	Encapsulate the content of main function as a particular function and call it on main function as a replacement of its content	18
P11	Encapsulate a particular task as a void function with the use of global variables	14
P12	Encapsulate a particular task as a void function without the use of global variables	7
P13	Encapsulate a particular task as a non-void function without the use of global variables	17
P14	Utilize API-based instruction instead of regular instruction	1
P15	Break down API-based instruction to several more-specific API-based instructions	10
P16	Exchange API-based instruction with other API-based instruction that yield similar functionality for particular circumstance	12
P17	Incorporate useless arguments on API function call or syntax form	75
P18	Replace constant value with variable or vice versa	9
P19	Replace constant with operation which yields similar result	6
P20	Change operand order in arithmetic or boolean operation	18
P21	Merge several operations without the use of temporary variables	31
P22	Replace increment/decrement instruction with their respective binary operator form	4
P23	Replace combined assignment with their respective binary operator form	18
P24	Replace data type with other data type that yields similar functionality for particular circumstance	4
P25	Incorporate useless casting	2
P26	Change loop type	8
P27	Incorporate dummy instructions without changing the decision logic	66
P28	Rearrange loosely-coupled instructions on similar scope	59
P29	Replace a number of repetitive instructions with a loop	9
P30	Change loop boundary	34
P31	Reverse loop direction from ascending to descending	3
P32	Rearrange branching statements based on its condition validation sequence	16
P33	Replace logical expression with other expression that yields similar meaning	27
P34	Incorporate logical expression that can be replaced with boolean constant	7
P35	Change incorporated algorithm with another algorithm which shares similar goal	2

First, P01, which is focused on modifying comment and whitespace, generates the highest number of occurrences on our dataset. It occurs on all 400 cases where most of them are represented as modifying comment. Such finding is natural since modifying comment is the easiest attack to be conducted. It does not affect the program flow yet it changes given program layout significantly. Indentation modification (e.g. replacing each tabulation with 3 spaces), on the contrary, only occurs on a small number of cases. We would argue that such few number of occurrences is caused by Python's strict indentation mechanism. Python forces the programmers to follow typical indentation style while writing the code. Therefore, source code indentation on Python source code cannot be modified freely as in other programming languages such as Java. Even though it is still possible to do such thing, it will take a considerable effort since all indentation tokens on given code should be replaced.

Second, P03, which is focused on modifying identifier name, generates the 2nd highest number of occurrences on our dataset. It occurs on 322 of 400 cases where most of them are represented as variable renaming. Such finding is natural since variable can be found on almost all sub-assignments, resulting it as the most prominent identifier type for this attack. The modification itself is varied from changing character capitalization (e.g. *maxsize* to *Maxsize*) to replacing the whole identifier name with other different-yet-similar term (e.g. *node* to *vertex*).

Third, P17, which is focused on incorporating useless arguments on API function call or syntax form, generates the 3rd highest number of occurrences on our dataset. It occurs on 75 of 400 cases even though such attack seldom occurs on other programming languages such as Java and C++. When discovered further, such unusual phenomenon is caused by the fact that Python overrides most of its built-in functions and syntaxes for programmer convenience, resulting numerous alternative forms for each function or syntax. Therefore, since such alternatives only differ in term of the number of parameters, useless arguments could be used easily as a plagiarism attack. In our case, *print* and *input* are the most frequently targeted built-in functions for this attack whereas *for-traversal* is the most frequently targeted syntax. The students frequently changed how they use these functions and syntax since they had been taught various forms of such functions and syntax on theory session. For instance, most students tended to discard some parameters on *for-traversal* since they had been taught how *for-traversal* works when represented with complete and incomplete parameters.

Fourth, P27, which is focused on incorporating dummy instructions, generates the 4th highest number of occurrences on our dataset. It occurs on 66 of 400 cases where such attacks could be roughly classified into two categories: incorporating either non-mandatory return keyword at the end of function declaration or useless print function invocation on the program body. The high number of occurrences of both attacks is natural since, in our course, both instructions, at some extent, do not affect student grades. We let the students to use return keyword if they want to and let them to freely put numerous print function invocations as long as the output would be quite similar with our desired output.

Last, P28, which is focused on rearranging loosely-coupled instructions, generates the 5th highest number of occurrences on our dataset. It occurs on 59 of 400 cases where most of them are only about re-arranging the position of one-line instruction such as variable assignment. Only a few of them re-arrange the order of numerous instructions at once.

In addition to findings found on Top-5 attacks, we also enlist several minor findings which have been discovered during the observation. These findings are: 1) In Python, it is possible to implement P02 attack by replacing string literal delimiter from single-quotation to double-quotation marks and vice versa, both marks are valid for declaring Python strings; 2)

Python enables multiple assignments, which is also a possible implementation of P05 attack; 3) Python does not have *do-while* syntax. Thus, *do-while* behavior on Python will be represented with *while* syntax, which might be varied in terms of implementation. Such variation could be used as P29 attack; and 4) The variance of incorporated attacks, at some extent, rely on the course syllabus. For example, P35 attack, which is not listed by Karnalim (2016), occurs in our dataset since, at that time, the students were required to implement an arbitrary sorting process while they had been taught three kinds of sorting algorithm on theory session.

The trend of empirically-listed plagiarism attacks toward Faidhi & Robinson's taxonomy

According to Faidhi & Robinson (1987), plagiarism attacks can be classified into six levels where, for each increasing level, the difficulty for doing such attack increases. The detail of such levels can be seen on Table 3. These levels start with level 1, which is about comment, whitespace, and delimiter modification, as the easiest level, to level 6, which is about logic change, as the hardest one. Among these levels, level 6 is the only level which is not affected by Python syntactical features. Such finding is natural since level 6 is about logic change, an attack category which is not directly related to programming language syntax.

Since higher attack level, at some extent, represents higher difficulty on such taxonomy (Faidhi & Robinson, 1987), the attack difficulty should be inversely proportional to the number of students who can do such attack. In other words, plagiarism attack frequency at a particular level should be fewer than plagiarism attack frequency at its lower level. In this section, we want to revalidate whether such trend is also applied to Python plagiarism attacks from Introductory Programming course. However, before analyzing such trend, plagiarism attacks listed on Table 2 will be mapped to Faidhi & Robinson's taxonomy. The result of such mapping can be seen on Table 4. Among these levels, level 5 is assigned with the most number of attacks, which is 15 of 35 attacks, while level 2 is assigned with the fewest number of attack, which is 1 of 35 attacks.

Table 3. Faidhi & Robinson's taxonomy of plagiarism attacks

Level	Attack Category
1	Comment, whitespace, and delimiter modification
2	Identifier renaming + level 1 attacks
3	Component declaration relocation + level 2 attacks
4	Inlining and outlining function + level 3 attacks
5	Program statement replacement + level 4 attacks
6	Logic change + level 5 attacks

Table 4. Mapped plagiarism attacks according to Faidhi & Robinson's taxonomy

Level	Included Plagiarism Attacks
1	P01 and P02
2	P03
3	P04 – P09
4	P10 – P13
5	P14 – P28
6	P29 – P35

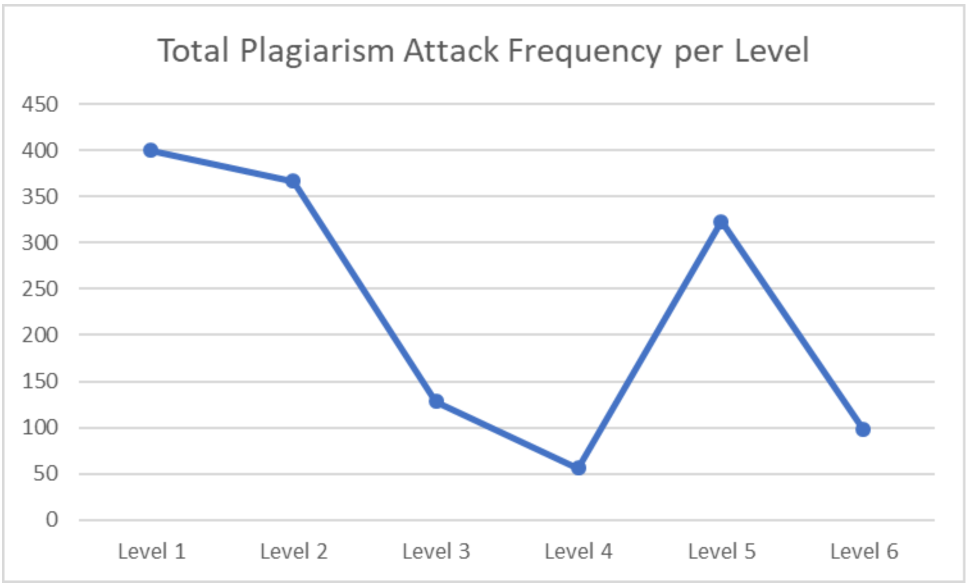

Figure 1. Total number of plagiarism attack occurrences per level

After all attacks have been mapped, total number of occurrences on each level is then summed and analyzed. However, since the number of occurrences in Table 2 do not provide sufficient information about how many attacks overlap to each other, we assume that all attacks on a particular level always overlap to each other. Therefore, for each level, its total number of occurrences are generated by selecting the highest number of occurrences from all attacks on given level. Total number of occurrences for each level can be seen on Figure 1. Horizontal axis represents Faidhi & Robinson's levels whereas vertical axis represents total number of occurrences. As seen in Figure 1, the number of plagiarism attack occurrences at a particular level is not always fewer than the number of plagiarism attack occurrences at its lower level. For instance, the number of level 5 attack occurrences is not fewer than the number of level 4 attack occurrences. When discovered further, there are several aspects which cause such phenomena.

Firstly, in Python, variable declaration cannot be written explicitly and it is always conducted implicitly when a variable is assigned with a value. Thus, it might mitigate the number of level 3 attack occurrences, as it is known that variable declaration relocation is a part of level 3 attacks. Secondly, since function material was introduced at the 9th session, level 4 attacks, which are about inlining and outlining method, might never occur at the beginning of the course, reducing the number of such attack occurrences. Thirdly, according to our informal in-class observation, most students tended to avoid using function since such mechanism made their code more complex. Consequently, such aspect might also reduce the number of level 4 attack occurrences. Finally, since Python overrides numerous built-in functions and syntaxes for programmer convenience, level 5 attacks, which are focused on program statement replacement, can be conducted easily, increasing the number of such attack occurrences.

To sum up, since the number of plagiarism attack occurrences at a particular level is not always fewer than the number of plagiarism attack occurrences at its lower level, it can be roughly stated that, in our case study, attack difficulty is not always proportional to increasing plagiarism level. We would argue that such finding is natural due to the nature of Python programming language, course syllabus, and student preferences.

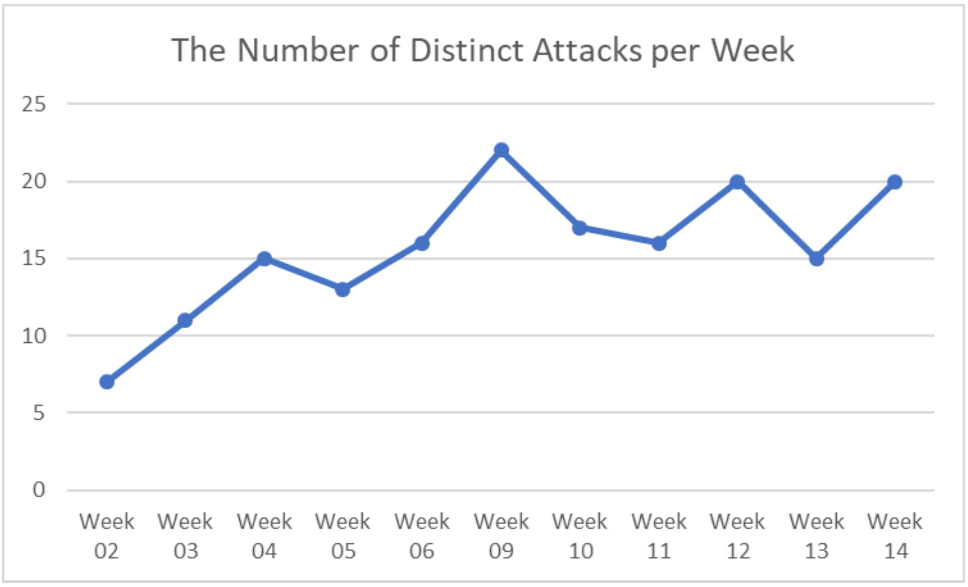

Figure 2. Number of distinct attacks per week

The trend of empirically-listed plagiarism attacks toward student experience

This section aims to revalidate whether the number of attack variance is proportional to the student experience or not. Based on the fact that student experience is enlarged as the number of lecture week increases, such relation can be measured based on the relation between the number of distinct attacks per week and the number of passed laboratory weeks. If the number of distinct attacks per week is increased as the number of passed laboratory weeks increases, then it can be stated that the number of attack variance is proportional to the student experience. The number of distinct attacks per laboratory week resulted from our dataset can be seen in Figure 2. Horizontal axis represents laboratory weeks whereas vertical axis represents the number of distinct attacks.

As seen in Figure 3, even though, in general, the number of distinct attacks is increased as the number of passed laboratory weeks increases, it is still reduced at certain points. To be specific, it is reduced at the 5th, 10th, 11th, and 13th week, which are about *traversal*, *return function*, *array*, and *matrix* material respectively. When discovered further through informal survey, such phenomenon occurs since, at that time, most students felt such materials were difficult to learn. They could not understand the materials comprehensively, resulting a discouragement to put various attacks on their code. Among given laboratory weeks, the 9th week, which is about *void function*, generates the most number of attack variance. Such finding is natural since, in such week, we encouraged the students to freely design the functions on their source codes. Thus, they would easily incorporate numerous obvious attacks without being worried to be accused as plagiarists.

By and large, when viewed in general, it is true that the number of attack variance is proportional to the student experience. However, since such trend is affected by student understanding and assignment restrictions, it is possible to see such trend fluctuates at certain points.

Threats to Validity

In general, there are two threats to validity which should be considered. On the one hand, it is important to note that our dataset does not include only true positive plagiarism cases due to limited human resource and time. However, we try to mitigate this threat by filtering plagiarism-suspected pairs through human-like approach for detecting plagiarism. We only took source code pairs which share high similarity degree in our dataset. On the other hand, due to limited observed cases, our result cannot be generalized to represent all possible Python plagiarism attacks for Introductory Programming course. However, we try to mitigate this threat by observing source codes collected for the whole semester and taking random source code pairs to avoid biased result. Yet, there is still a possibility that several attacks were not observed.

Conclusions

In this paper, we have enlisted Python plagiarism attacks on Introductory Programming course. Based on our 400 plagiarism-suspected pairs, there are 35 distinct plagiarism attacks, where Top-5 attacks are modifying comment and whitespace, modifying identifier name, incorporating useless arguments on API function call or syntax form, incorporating dummy instructions, and rearranging loosely-coupled instructions. Beside such finding, two additional findings can also be deducted which are: 1) Plagiarism attack difficulty is not always proportional to increasing plagiarism level due to the nature of Python programming language, course syllabus, and student preferences and 2) The number of distinct plagiarism attacks per week is proportional to the student experience even though it is still affected by student understanding and assignment restrictions.

For future work, these findings will be used as a baseline for developing a Python-targeted plagiarism detection system. We intend to propose a plagiarism detection system that is sensitive to popular Python plagiarism attacks in Introductory Programming environment. Moreover, we also intend to conduct similar empirical study on Data Structure course, which incorporates standard object-oriented techniques. Such finding then will be merged with this work to provide more-comprehensive study about Python plagiarism attacks in CS education.

References

Ahmadzadeh, M., Mahmoudabadi, E., & Khodadadi, F. (2011). Pattern of plagiarism in novice students' generated programs: An experimental approach. *Journal of Information Technology Education: Innovations in Practice, 10*, 195-205.

Al-Khanjari, Z.A., Fiadhi, J.A., Al-Hinai, R.A., & Kutti, N.S. (2010). PlagDetect: a Java programming plagiarism detection tool. *ACM Inroads*, 1(4), 66-71).

Bandara, U., & Wijayarathna, G. (2011). A machine learning based tool for source code plagiarism detection. *International Journal of Machine Learning and Computing, 1*(4), 337-343.

Chilowicz, M., Duris, E., & Roussel, G. (2009). Syntax tree fingerprinting for source code similarity detection. *Proceedings of IEEE 17th International Conference on Program Comprehension* (pp. 243-247). Vancouver: IEEE.

Cosma, G., & Joy, M. (2008). Towards a definition of source-code plagiarism. *IEEE Transactions on Education, 51*(2), 195-200.

Cosma, G., & Joy, M. (2012). Evaluating the performance of LSA for source-code plagiarism detection. *Informatica, 36*, 409-424.

Djuric, Z., & Gasevic, D. (2012). A source code similarity system for plagiarism detection. *The Computer Journal, 55*, 70-86.

Engels, S., Lakshmanan, V., & Craig, M. (2007). Plagiarism detection using feature-based neural networks. *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (pp. 34-38). New York: ACM.

Faidhi, J. A., & Robinson, S. K. (1987). An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education, 11*(1), 11-19.

Guo, P. J. (2013). Online python tutor: Embeddable Web-based program visualization for CS education. *Proceedings of the 44th SIGCSE Technical Symposium on Computer Science Education* (pp. 579-584). New York: ACM.

Jadalla, A., & Elnagar, A. (2008). PDE4Java: Plagiarism detection engine for Java source code: A clustering approach. *International Journal of Business Intelligence and Data Mining, 3*(2), 121-135.

Ji, J.-H., Woo, G., & Cho, H.-G. (2008). A plagiarism detection technique for Java program using bytecode analysis. *Proceedings of the third International Conference on Convergence and Hybrid Information Technology* (pp. 1092-1098). Busan: IEEE.

Juričić, V. (2011). Detecting source code similarity using low-level languages. *Proceedings of the 33rd International Conference on Information Technology Interfaces* (pp. 597-602). Dubrovnik: IEEE.

Juricic, V., Juric, T., & Tkalec, M. (2011). Performance evaluation of plagiarism detection method based on the intermediate language. In C. Billenness, A. Hemera, V. Mateljan, M. Banek Zorica, H. Stančić & S. Seljan (Eds.), *Proceedings of the 3rd International Conference ''The Future of Information Sciences: INFuture2011-Information Sciences and e-Society''* (pp. 355-363). Zagreb, Croatia: University of Zagreb.

Karnalim, O. (2016). Detecting Source code plagiarism on introductory programming course assignments using a bytecode approach. *Proceedings of the 10th International Conference on Information & Communication Technology and Systems (ICTS)* (pp. 63-68). Surabaya: IEEE.

Kustanto, C., & Liem, I. (2009). Automatic source code plagiarism detection. *Proceedings of the 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing* (pp. 481-486). Daegu: IEEE.

Lim, J.-S., Ji, J.-H., Cho, H.-G., & Woo, G. (2011). Plagiarism detection among source codes using adaptive local alignment of keywords. *Proceedings of the 5th International Conference on Ubiquitous Information Management and Communication* (pp. 24-34). Seoul: ACM.

Ohno, A., & Murao, H. (2011). A two-step in-class source code plagiarism detection method utilizing improved CM algorithm and SIM. *International Journal of Innovative Computing, Information, and Control, 7*(8), 4729-4739.

Parker, A., & Hamblen, J. O. (1989). Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, *32*(2), 94-99.

Parr, T. (2014). *ANTLR*. Retrieved 12 July 2015, from http://www.antlr.org.

Prechelt, L., Malpohl, G., & Philippsen, M. (2002). Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science, 8*(11), 1016-1038.

Rabbani, F. S., & Karnalim, O. (2017). Detecting source code plagiarism on .NET programming languages using low-level representation and adaptive local alignment. *Journal of Information and Organizational Sciences, 41*(1), 105-123.

Ramirez-de-la-Cruz, A., Ramirez-de-la-Rosa, G., Sanchez-Sanchez, C., Jimenez-Salazar, H., Rodriguez-Lucatero, C., & Luna-Ramirez, W. A. (2015). High level features for detecting source code plagiarism across programming languages. *Proceedings of the Cross-Language Detection of SOurce COde Re-use Conference* (pp. 10-14). Gandhinagar, India.

Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 76-85). San Diego: ACM.

Smith, T. F., & Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology, 147*, 195-197.

Sraka, D., & Kaucic, B. (2009). Source code plagiarism. *Proceedings of the 31st International Conference on Information Technology Interfaces, ITI' 09* (pp. 461-466). Cavtat, Croatia: IEEE.

Wise, M. J. (1993). *Running rabin-karp matching and greedy string tiling.* Basser Departement of Computer Science, Sydney University.

Wise, M. J. (1996). YAP3: Improved detection of similarities in computer programs and other texts. *ACM SIGCSE Bulletin, 28*(1), 130-134.

To cite this article: Karnalim O. (2017). Python source code plagiarism attacks on introductory programming course assignment. *Themes in Science and Technology Education*, 10(1), 17-29.

URL: http://earthlab.uoi.gr/theste