2020 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)

Identifying Social Networks of Programmers using Text Mining for Code Similarity Detection

Konstantinos F. Xylogiannopoulos
Dept. of Computer Science
University of Calgary
Calgary, Canada
https://orcid.org/0000-0003-2376-898X

Panagiotis Karampelas
Dept. of Informatics & Computers
Hellenic Air Force Academy
Dekelia, Greece
https://orcid.org/0000-0003-1684-7612

***Abstract*—The availability of code in many online repositories and collaborating platforms has posed new challenges in source code attribution not only for plagiarism detection but also in other settings such as in the use of insecure copied code in commercial application, etc. The sophistication of different type of attacks in the code sequence used especially by the students requires more effective code similarity detection algorithms. In this paper, a novel source code detection method is proposed that can identify programmers' social network based on advanced pattern detection text mining techniques. The proposed methodology has significant advantages against existing methods since ARPaD algorithm can detect all common patterns between all possible code sequences in one run. Therefore, the computational time is massively reduced to $O(mn \log n)$. In order to assess the performance of the methodology, a new dataset was created by assigning to 46 students a code project with specific instructions. The assessment results have been visualized, producing the social network graphs of possible collaboration teams.**

Keywords*—code similarity detection, social network analysis, code plagiarism detection, text mining, LERP-RSA, ARPaD

I. Introduction

The social web has fostered a number of novel interactions among people of every discipline either related to computer science or not. Various types of electronic social networks, online collaborating platforms, photo and video sharing websites and review platforms are some of the types of social platforms that emerged in the social web. Through the years, these platforms appear to have been transformed in more focused platforms that satisfy particular needs of their participants. In this socially connected world, programmers have developed their own online social places in which they exchange information, knowledge, advice, even source code. In this context, a great number of specialized social platforms have arisen in the service of programmers such as Questions and Answers websites, Code Repositories, Programmers' Blogs and several other platforms that facilitates the communication among them.

Popular collaboration platforms such as StackOverflow [1] reports more than 100 million visits per month having answers for more than 20 million questions related to coding. The parent company StackExchange [2] reports an average rate of approximately 420 million visits per month in all the 173 different Q&A communities hosted, a great deal of them related to programming. Other less specialized communities, such as Quora [3], Reddit [4] and Google Groups [5] host numerous coding communities formed by programmers, dedicated to solve programming problems, sharing programming knowledge and teaching younger members of the communities. In addition, code hosting platforms such as GitHub [6], BitBucket [7] and SourceForge [8] host numerous programs for any existing programming environment and language. GitHub alone reports more than 100 million repositories up to 2019.

As it can be understood by the abovementioned statistics, there is a plethora of code fragments all over the Internet accessible to all. In this sense, writing a program, responding to a programming exercise or solving a coding issue has become easier than ever before. However, in some cases this broad availability of code in the wild may raise various issues depending on the context of use. For example, when the code found in any of the abovementioned platforms has been used to respond to a programming exercise by a student, then there is a plagiarism case, since the purpose of the exercise is the professor to decide whether the student is able to deliver a meaningful algorithm and programming code that solves the assigned problem and not to grade code developed by a professional who posted it in a repository. Moving in a professional setting, there are cases in which developers search and find code in the wild that is then embedded to a business application with immeasurable repercussions for the security of the application [9]. Identifying such code promptly is very important since, in the case of the student, the professor can easily decide whether the student committed plagiarism or not, while in the other case, it will be possible for the software tester to timely remove the code in question before causing any irreversible damage to the whole application. Similar code detection is also valuable in malicious code detection since viruses usually come with slight variations in order to outsmart the antivirus systems or in vulnerability code detection when attempting to locate similar code that is known to be vulnerable [9].

The work proposed in this paper, utilizes a novel approach in analyzing source code taking advantage of LERP-RSA data structure [25-27] and ARPaD [25, 26] algorithm in order to identify all repeated patterns found in the code under investigation. The common code detected is further automatically analyzed and the corresponding results are visually presented showing similarities in the different code sequences. The advantages and contributions of the specific approach can be summarized as follows:

- All the existing common code patterns are analyzed and found irrespectively of their length using a commodity computer which to the best of our knowledge cannot be done with any other approach with complexity $O(mn \log n)$.
- With a simple meta-analysis process, the calculated results can be easily visualized and further analyzed using different visualization techniques and user-defined thresholds depending on the requirements of

the analyst without repeating the data structure construction and the detection algorithm execution.

- A contemporary dataset with 46 code samples in Python has been created that exhibits a wide range of similarities and can be used as a reference dataset for testing other code similarity detection or plagiarism detection techniques.

The paper is organized as follows: Section II analyzes the existing work found in literature related to code similarity detection and code plagiarism detection. In Section III, the proposed code detection methodology is outlined, describing the different phases of the analysis. Section IV presents how the specific dataset has been created and discusses the findings of the code similarity detection with the proposed methodology and the last Section presents the conclusion and the next steps.

II. Related Work

Code similarity is a well-attended problem that has occupied many researchers, mainly in the context of code plagiarism in order to identify students who cheat in their programming assignments. However, code similarity is a broader issue since it may have several other implications such as (a) to identify security issues introduced by the copied code; (b) to detect variants of malware as it was previously mentioned; (c) to detect unauthorized use of copyrighted material in commercial software without proper license, etc. The principles behind code similarity detection may differ depending on the code at hand. For example, binary code similarity detection can be used when the binary code is available in a specific context [22] while reversed engineered code can be used in malware analysis and classification [28]. Finally, source code similarity is studied when there is a need to detect potential cases of plagiarism in programming assignments [23, 24]. A variation of our approach has been assessed in malware analysis [28] providing promising results and in this paper the specific methodology has been altered, improved and extended in order to cater for the needs of source code plagiarism detection.

Code plagiarism detection attempts to identify similar code used in different code sequences especially in programming assignments. More specifically, in [10-15], a list of the most typical cases of code alteration applied by students to avoid detection is presented. These alterations include the use of different variable names, the altering of formatting, the change of comments, the application of alternate indentation, the use of different operators, the reordering of code blocks, and other techniques that aim to spoof the code plagiarism detection process. In response to these practices, a number of code plagiarism techniques have been proposed by the researchers achieving different success rates.

Cosma & Joy [16] suggest that there are two main categories of algorithms that have been used for code plagiarism detection based on fingerprints and string-matching. Fingerprints are basically collected statistics related to the source file such as number of keywords, characters per line, etc. which are then used to compare two different programs. The more common statistics, the greater is the similarity of the programs. String-matching algorithms take advantage of the known syntax of a specific programming language and analyze the code of the programs based on common structures identified [17]. As the researchers report, this technique may be more sensitive to various attacks such as "code-shuffling", etc. [16] and thus may not be very efficient. Our approach addresses the specific issue since it does not utilize small n-grams as the majority of these techniques but instead detects all the repeated code patterns among the code sequences and combines the identified results showing better persistence to the attacks.

An n-gram approach is proposed in [18] where n-grams of different size are used to calculate the similarity between the source code in a large dataset of programming assignments. The researchers considered the larger n-grams as an indication of greater similarity but they found out that the specific approach as they report did not outperform other known similarity functions such as Okapi BM25. In contrast to the aforementioned n-grams technique, our approach detects all repeated patterns regardless of their length, without the need to define a specific n-gram size. However, this has been proven to be extremely space and time efficient based on the complexity described in our paper. Despite that we also discover that the use of only very long patterns could be misleading when insertion attacks have been performed, yet, the discovery of every possible repeated pattern can easily bypass such attempts.

Other plagiarism detection techniques employ genetic programming [19] in order to improve the similarity functions and thus to achieve better results in code similarity detection. The experiment held with a specific similarity function was considered successful since the results showed that the technique outperformed traditional code similarity detection tools, however, as the researchers reported further development of such techniques for novel similarity functions is required in order to be robust.

Another approach is proposed in [20], where instead of using the source code of a program, the intermediate code (bytecode) of the programs is used for comparison purposes. It is claimed that this approach is more efficient since bytecode avoids a great number of plagiarism attacks however it is not possible to be applied in all the programming languages since there are many that do not use bytecode and are translated directly in native machine code.

There are also techniques that combine different stages of analysis for code similarity detection. In [21], the researchers use an "attribute-counting system" in identifying code similarity which is then improved by applying clustering in order to improve the detection performance. The specific method is a two-stage approach as the one proposed in our case, however, applying totally different strategies in the implementation of both phases.

Our proposed approach falls under the string-matching techniques and attempts to alleviate the shortcomings of the known methods by combining common patterns found among the programs, with different length, in different positions and thus illustrating a more accurate representation of the similarities.

III. Code Similarity Detection

The proposed methodology is based on our previous techniques used for text mining for plagiarism detection [29] and malware classification [28]. In those initial attempts, we tried to detect text similarities based on previous knowledge of patterns that could characterize a text similar to another or parts of code previously classified as malware. Yet, those methods were significantly easier than the problem we are trying to address in this paper, since the text or code similarity checking was performed based on one-to-one analysis, such as in the case of previously labeled data for malware code detection.

The problem we are trying to tackle is significantly more complicated for many reasons. First of all, we do not have any previous knowledge or labeled data regarding the dataset. Moreover, a one-to-one comparison is not possible, firstly because there are no referenced, labeled, data and, therefore, the time complexity of a brute force approach is at least $O(n^2)$, based on the number of sequences (code snippets) to examine, without taking in consideration other parameters such as the length of the sequences. Another problem is that we have to cluster the sequences based on their algorithmic logic rather than their exact text similarity. For this purpose, using short patterns with length four, five, six or a few more characters is completely meaningless, because such short patterns can easily be formed by programming languages reserved words, e.g., in Python, if, for, def, else, range, print, input, etc.

In order to address this problem, the methodology follows a sequence of steps. First the data need to be cleaned and prepared for analysis. Then the Longest Expected Repeated Pattern Reduced Suffix Array (LERP-RSA) [25-27] data structure is used that allows to the All Repeated Patterns Detection (ARPaD) [25, 26] algorithm to be executed and detect all common patterns. The total complexity of the methodology is calculated based on LERP-RSA construction and ARPaD execution, which it has been proven in [25-27] to be $O(mn\log n)$ with regard to the input size of m sequences of length n. Finally, the results of the algorithm need to be further analyzed with a meta-analysis in order to produce meaningful output. The process is described in details in the following sub-sections.

A. Data Cleansing

The first step of the methodology is to clean and prepare our dataset. Since we care about identifying algorithmic logic in the code, first we need to clean our sequences from any no-reserved programming language word such as variable and function names. Only reserved words and language specific symbols are preserved as it is also suggested in [18] that follows a similar strategy. Additionally, any formatting character such as tab, space, line feed, carriage return, etc. is removed. Finally, we end up with a single line string representing the original code, one for each sequence.

B. Multivariate LERP-RSA Construction

The next step in our methodology is the construction of the multivariate data structure LERP-RSA. The strings created from the code sequences are stored in the LERP-RSA data structure using three different columns. The first two columns hold the standard information of the LERP-RSA which is the suffix string and the position in which the specific substring has been found. The additional column stores the index of the sequence.

It needs to be mentioned here that LERP-RSA uses additional initial parameters that allows us to optimize data structure construction. More specifically, the use of the Longest Expected Repeated Pattern (LERP) value which defines the longest pattern that it is expected to be repeated and, therefore, be common among sequences. Yet, in our case we define this parameter based on the longest pattern that we want to identify, regardless if any longer may exist. After creating the initial list of sequences and defining the parameters, then all produced suffix strings are merged and lexicographically sorted in ascending order based on the suffix string column. The specific procedure depends on the hardware availability and it can run in parallel by sorting the suffix strings based on their initial character, which can accelerate the specific step execution significantly.

C. Common Patterns Discovery

After the creation of the multivariate LERP-RSA data structure, the ARPaD algorithm is executed. An important initial parameter for ARPaD is the Shorter Pattern Length (SPL) which is a lower bound for the patterns length that we want to discover. For example, in this paper, we do not care for very short patterns, as mentioned above, rather than we care for patterns of 20 or more characters. The algorithm identifies all the repeated patterns found in the stored data structure irrespective of the length of the pattern but between the SPL and LERP values. Depending on the use of parallel LERP-RSA construction with the use of subclasses, the algorithm can also be executed in parallel and produce results significantly faster. It is important to mention that the results will contain patterns that occurred at the same sequence. In this case, the pattern is not important since similarities between only one sequence cannot be used to define a network clustering. This is why, the last phase of meta-analysis of the results is so important in order to eliminate meaningless patterns.

D. Pattern Results Meta-analysis

As mentioned above, a meta-analysis is fundamental to clean the results and produce meaningful output that can be used to cluster our sequences and create their social network. In this step, first, we need to remove from our results repeated patterns that could either occur in the same sequence or in more than one sequence but multiple times. Second, we must remove overlapping patterns, since they introduce noise in our results. For example, if the pattern $aaa\cdots aaa$ of 25 continuous a appears at position 20 then the pattern $aaa\cdots aaa$ of 24 continuous a appears at positions 20 and 21, the pattern $aaa\cdots aaa$ of 23 continuous a appears at positions 20, 21 and 22, etc. Although in many other problems, this is a very important knowledge in our case is irrelevant since we care about the longest possible patterns that exist among different sequences. The third step is to calculate the overall pattern coverage for each sequence. For example, if two patterns of length 30 have been found for a sequence then the total overlapping length for the sequence will be 60. The fourth step in the meta-analysis is to create a graph structure, using an adjacency list of the discovered patterns (as edges) among all different sequences (vertices). This will allow us to visualize the results in a very meaningful way as it will be presented in the experimental analysis section.

IV. Experimental Analysis

A. Dataset

In order to evaluate the proposed methodology, an individual programming assignment was introduced in a class of 46 students. The students were provided with a detailed description of the assignment requesting the implementation of a specific algorithm. More specifically, it was suggested to the students to use a specific data structure to store the relevant data of the assignment and then to create a method for the implementation of the required calculations. The students were also encouraged to collaborate in small teams for preparing the assignment, without reporting their collaboration to conform with General Data Protection Regulation standards and institutional regulation. Moreover, a consent form to use their code output for the specific analysis was also distributed and signed by all the students according to the ethical guidelines of the institution.

The collected code sequences were then anonymized by introducing random names and random data values wherever was necessary. Subsequently, the source code sequences were processed according to the proposed methodology in order to detect code similarity among them. The specific anonymized dataset has been uploaded in github in the following address:

http://github.com/pkarampelas/Code-Similarity-Detection-Dataset/

TABLE I. TABLE TYPE STYLES

Dataset Attributes	*Values*
Number od code sequences	46
Longest length of code sequences	748
Minimum length of code sequences	203
Average length of code sequences	307
Longest length after data cleansing	173
Minimum length after data cleansing	73
Average length after data cleansing	109

Table I presents the dataset statistics before and after the cleansing phase without formatting characters in between the code instructions.

Concerning the infrastructure, a laptop with an Intel i7 CPU has been used, with quad core processor and 16GB RAM. A solid-state disk of 256GB has been also used to store, process and present analysis results. Several visualizations have been created that emphasize the interconnection between sequences.

B. *Results & Discussion*

First of all, the series of plots (Fig. 1-4) represents the sequences with their length as a narrow grey line and the common patterns plotted on top with different colors and transparency level. This group of plots varies by presenting patterns with length greater than or equal to 20, 30, 40 and 50. The second group of plots (Fig. 5-8) illustrates graphs representing the social networks, created from common patterns, that have several minimum total overlapping length thresholds such as 30, 40, 50 and 60. In all of them, the minimum length of common patterns displayed is equal to 20. As we can observe in Table II, 11 patterns found with length great than 52 and 17 with length between 30 and 50. Additionally, 55 more patterns detected with length between 20 and 30 adding up to a total of 84 patterns. Furthermore, we can observe in Table III that in total there are 138 sets of sequences formed with minimum total overlapping length 20.

The SPL parameter used for the experiment is 20 while the LERP is 100. This parameter initialization allows ARPaD to detect all repeated patterns with length between 20 and 100, as it can be observed in Table II. As mentioned before using only short patterns could be extremely misleading in the sense that practically all sequences will end up being related somehow (Fig.1 and Fig. 5). Moreover, the use of only very long patterns could also be misleading by missing important relations between sequences having several short patterns in common (Fig. 4 and Fig. 8). Thus, we care to scan between different thresholds of common pattern length, e.g., 20, 30, 40 and 50 (Fig. 1-4 respectively) and minimum total overlapping length, e.g., 30, 40, 50 and 60 (Fig. 5-8 respectively). For example, sequences 40 and 45 are highly correlated since they have two common patterns with lengths 87 and 48. Practically, there is a single character (insertion attack) in sequence 45 that separates these two patterns, otherwise both sequences would be exactly identical (Fig. 4) since the total overlapping length is 135 and the sequences have length 135 and 136 respectively (Table III). On the contrary, sequences 2 and 8 have not any long common pattern, yet, they are also highly correlated as we can observe in “Fig. 1”, since they have three common patterns with lengths 33, 29 and 24. These patterns are separated from few characters in between (insertion attack) leading to a total overlapping length of 84 characters out of 91 for sequence 2 and 94 for sequence 8 (Table III), which practically means a similarity above 90%.

Except the sequence sets described above, we can observe some more interesting sets in “Fig. 4” for patterns with length greater than 50. These sets include two sets of sequences {27, 46} and {5, 19} with a common pattern of length 88, set {16, 28} with a common pattern length 86, set {20, 38} with a common pattern of length 82, set {5, 7} with a common pattern length 80, set {41, 22} with a common pattern length 60 and set {12, 14} with a common pattern length 52. Moreover, there are two more supersets, set {5, 7, 19} with a common pattern of length 64 and set {6, 16, 28, 39} with a common pattern length of 54. All these sets can be easily observed as social networks in plots “Fig. 7” and “Fig. 8”. All these sets described have been formed from single patterns that are extremely long comparing to the total length of the transformed and cleaned code that varies between 73 and 173 characters with a mean of 109 characters (Table I). This signifies very strong relations among the different code sequences since it is practically impossible to have such long patterns in the code without some kind of social connection between programmers.

However, in these plots we can observe something else, equally interesting, which is the form of more complex social networks based on shorter patterns “Fig. 1-2”. For example, we have the set {9, 38} with total length 78 formed from three smaller patterns of lengths 20, 24 and 34 and set {27, 38} with total length of 73 formed from three patterns with lengths 21, 24 and 28. These sets are formed from smaller patterns because of very few characters which were cleverly inserted somewhere in between a considerably larger pattern in order to hide the connection, yet, our methodology managed very easily to detect them. Moreover, these sets are examples of largely formed social networks, because set {9, 38} connects sets {20, 38}, {27, 46} and {27, 38} as we can observe in plot “Fig. 8”.

Although the aforementioned examination process can easily be executed manually by investigating visually the results, there is the need to automate the process and avoid using arbitrary length thresholds, e.g., 30, 40, etc. as we did here for presentation purposes. This automation can be achieved be performing a top down scan on Table III, after sorting it in descending order on the “Total Length” column. This column is practically the weights for each edge of the graph between the vertices of the column “Sequences Sets”.

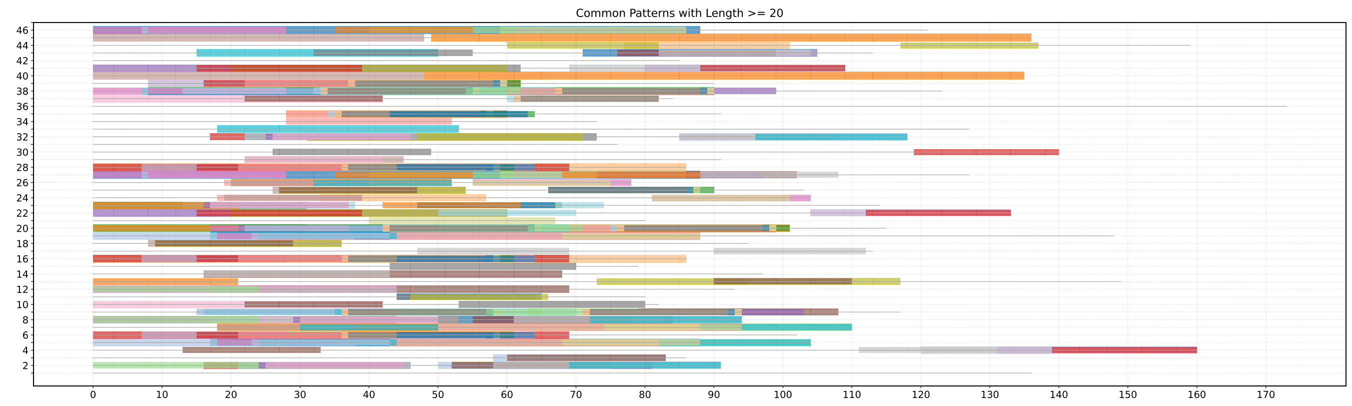

Fig. 1. Sequences and Common Patterns with Length greater than or equal 20

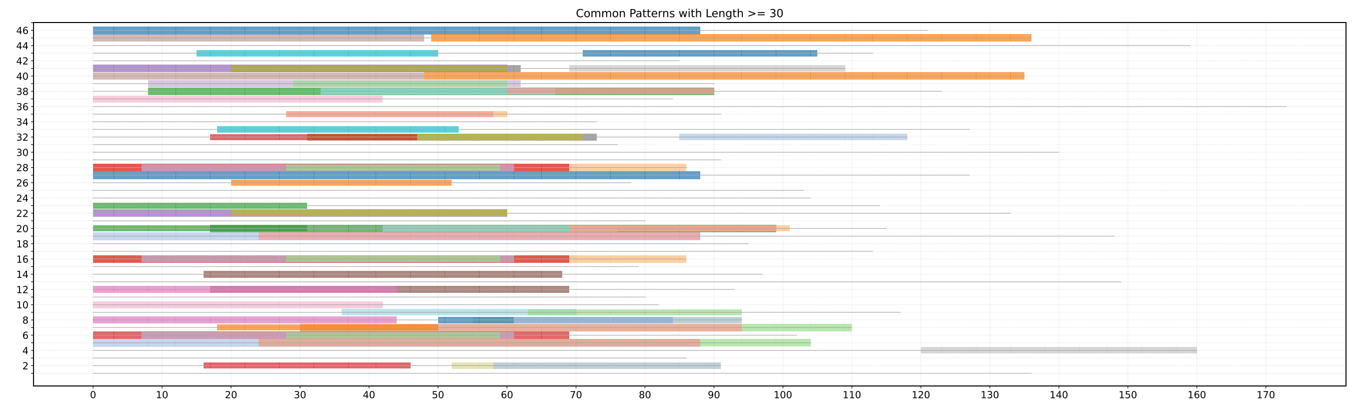

Fig. 2. Sequences and Common Patterns with Length greater than or equal 30

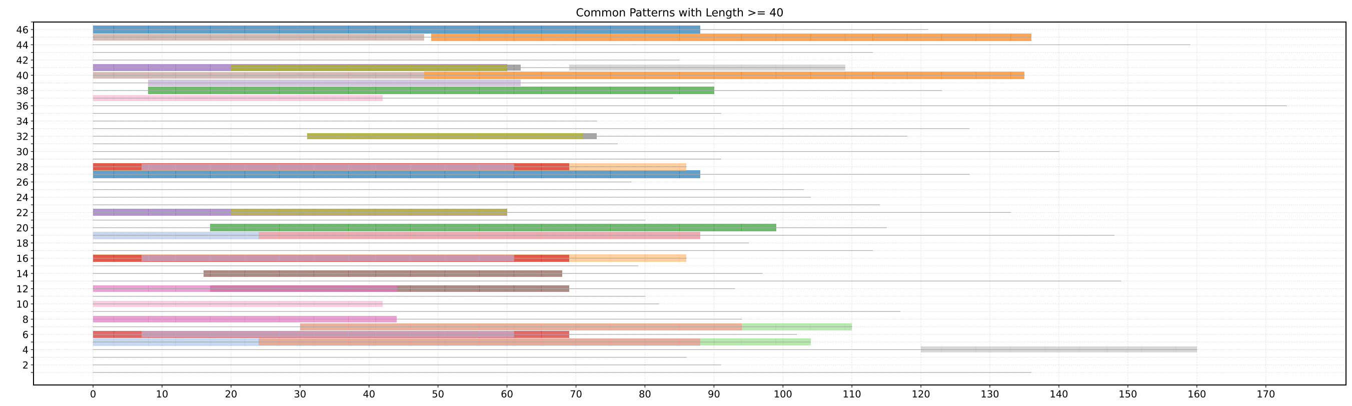

Fig. 3. Sequences and Common Patterns with Length greater than or equal 40

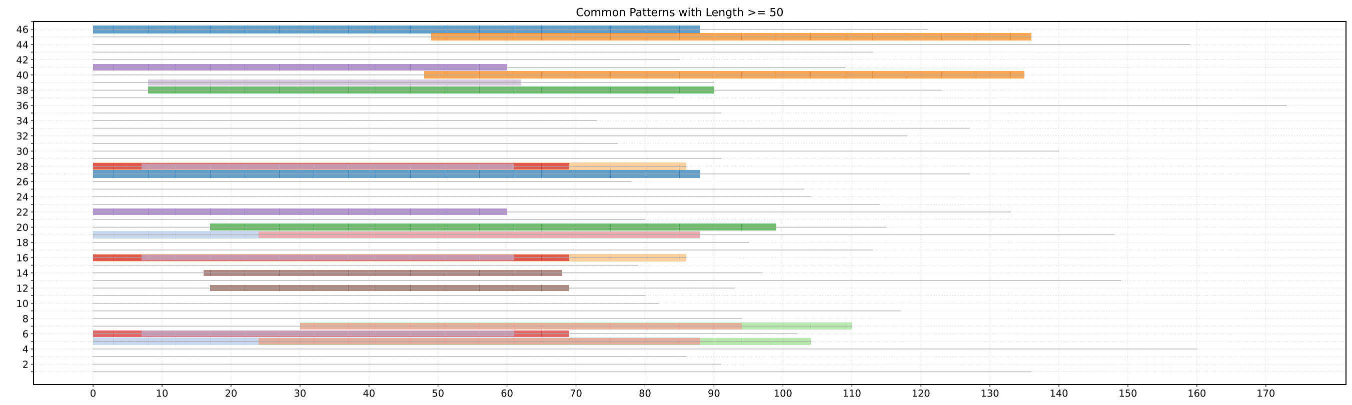

Fig. 4. Sequences and Common Patterns with Length greater than or equal 50

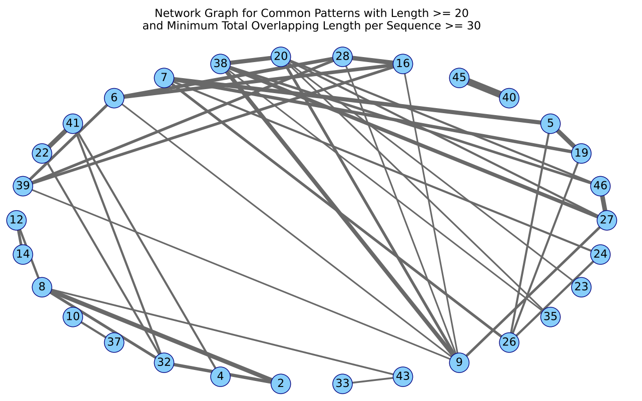
Network Graph for Common Patterns with Length >= 20
and Minimum Total Overlapping Length per Sequence >= 30

Fig. 5. Social network for total overlapping length >= 30

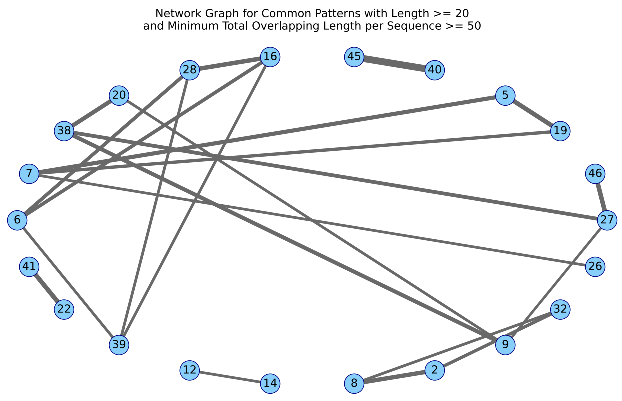
Network Graph for Common Patterns with Length >= 20
and Minimum Total Overlapping Length per Sequence >= 50

Fig. 7. Social network for total overlapping length >= 50

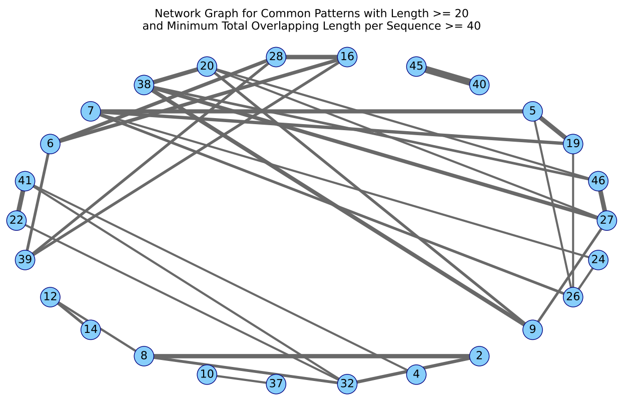
Network Graph for Common Patterns with Length >= 20
and Minimum Total Overlapping Length per Sequence >= 40

Fig. 6. Social network for total overlapping length >= 40

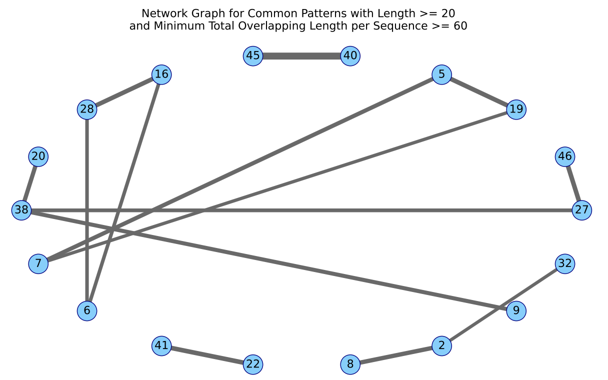
Network Graph for Common Patterns with Length >= 20
and Minimum Total Overlapping Length per Sequence >= 60

Fig. 8. Social network for total overlapping length >= 60

TABLE II. PATTERNS DETECTED WITH LENGTH GREATER THAN OR EQUAL 30

Pattern	**P.L.**	**Sequences**
`=[,,,,]=def(,):=forin:if>:==chr(ord()+[])print(,end=)=+else:=chr(ord()+[])print(,end=)=+`	88	27, 46
`=[,,,,]==input(:)=def(,):=forin:if>:==ord()=ord()+[]=+else:=ord()+[]=+=chr()print(,end=)`	88	19, 5
`,end=)forin:.(ord())=forin:=()+([]).()if==:=else:+=forin:.(chr())forin:print(,,end=)(,)`	87	40, 45
`=input()=[,,,,]=def():=forin:=ord()=+[]=chr()print(,end=)=+if>:=if==:()else:print(:)()`	86	16, 28
`def(,):=forin:if>:==ord()+[]=chr()print(,end=)=+else:=ord()+[]=chr()print(,end=)=+`	82	20, 38
`:=forin:if>:==ord()=ord()+[]=+else:=ord()+[]=+=chr()print(,end=)if==:(,)else:(,)`	80	5, 7
`=input()=[,,,,]=def():=forin:=ord()=+[]=chr()print(,end=)=+if>:=if==:`	69	16, 28, 6
`:=forin:if>:==ord()=ord()+[]=+else:=ord()+[]=+=chr()print(,end=)`	64	19, 5, 7
`=[,,,,]=input()=def():=forin:=ord()=+[]print(chr(),end=)=+if`	60	41, 22
`)=[,,,,]=def():=forin:=ord()=+[]=chr()print(,end=)=+if`	54	16, 28, 6, 39
`print(:,end=)def(,):=forin:print(chr(ord()+[]),end=)`	52	12, 14
`=input()=[,,,,]iflen()==:=def(,):=[]=[]=[]print(`	48	40, 45
`=input(:)=[,,,,]=print(:,end=)def(,):=forin:`	44	8, 12
`def():=input()if==:=()forinrange(,len()):=`	42	10, 37
`):=forin:=ord()=+[]print(chr(),end=)=+if==`	42	32, 41
`if==:print(:,end=)()else:print(:,end=)()`	40	41, 4
`):=forin:=ord()=+[]print(chr(),end=)=+if`	40	32, 41, 22
`if==:=print(chr(),end=)if==:(,)else:(,)`	39	8, 2
`]=[]def(=,=[,,,,]):forin:=ord().()=`	35	33, 43
`+[]=chr()print(,end=)=+else:=ord()`	34	9, 20, 38
`+[]+=if==:=print(chr(),end=)if==:(`	34	8, 43
`print(chr(),end=)if==:(,)else:(,)`	33	32, 8, 2
`def(,):=if==:=forin:if>:==ord()=`	32	26, 7
`:=ord()+[]=chr()print(,end=)=+if`	32	35, 20
`=[,,,,]==input(:)def(,):=forin:`	31	20, 23
`:=ord()=+[]=chr()print(,end=)=+`	31	6, 39, 9, 16, 28
`=print(:)def(,):=forin:=ord()=`	30	32, 2
`:=ord()+[]=chr()print(,end=)=+`	30	35, 20, 38

TABLE III. SEQUENCE SETS WITH TOTAL LENGTH GREATER THAN OR EQUAL TO 20

Sequences Set	Total Length	Sequences Set	Total Length	Sequences Set	Total Length	Sequences Set	Total Length	Sequences Set	Total Length	Sequences Set	Total Length
27, 46	88	32, 8	54	21, 46	27	22, 39	24	8, 5	22	18, 4	20
19, 5	88	32, 2	63	19, 38	26	16, 41	24	8, 7	22	25, 4	20
40, 45	135	26, 7	52	5, 38	26	41, 28	24	35, 37	22	13, 37	20
16, 28	86	35, 20	32	32, 23	25	16, 22	24	20, 37	22	18, 37	20
20, 38	82	20, 23	31	2, 23	25	28, 22	24	37, 38	22	25, 37	20
5, 7	80	9, 6	31	19, 20	25	34, 6	24	5, 46	21	10, 13	20
16, 6	69	9, 39	31	20, 5	25	34, 39	24	27, 5	21	10, 18	20
28, 6	69	16, 9	31	2, 3	25	16, 34	24	19, 46	21	25, 10	20
19, 7	64	9, 28	31	35, 39	25	34, 28	24	27, 19	21	24, 7	40
41, 22	89	35, 38	30	20, 6	25	32, 43	23	20, 13	21	32, 29	20
16, 39	54	2, 43	29	20, 39	25	8, 3	23	13, 23	21	2, 29	20
28, 39	54	4, 22	29	16, 20	25	43, 3	23	35, 23	21	29, 23	20
6, 39	54	9, 27	50	20, 28	25	8, 29	23	9, 46	21	27, 4	20
12, 14	52	25, 18	28	24, 44	24	24, 26	44	20, 46	41	11, 44	20
8, 12	44	27, 38	73	25, 35	24	43, 30	23	27, 20	41	24, 5	20
10, 37	42	46, 38	49	25, 6	24	32, 17	22	41, 30	21	26, 5	40
32, 41	42	16, 35	28	25, 39	24	17, 2	22	4, 30	21	24, 19	20
41, 4	40	35, 28	28	16, 25	24	8, 17	22	30, 22	21	26, 19	40
32, 22	40	35, 6	28	25, 20	24	17, 43	22	8, 23	21	9, 7	20
8, 2	84	10, 15	27	25, 28	24	44, 13	22	9, 19	21	9, 26	20
33, 43	35	8, 14	27	2, 12	24	32, 5	22	9, 5	21	22, 23	20
9, 20	54	25, 13	27	41, 6	24	32, 7	22	4, 37	20	6, 23	20
9, 38	78	18, 13	27	6, 22	24	2, 5	22	10, 4	20	16, 23	20
8, 43	34	27, 21	27	41, 39	24	2, 7	22	4, 13	20	28, 23	20

Scanning the table and using different thresholds on these weights can provide us with the appropriate social collaboration networks that we want to identify. Moreover, these absolute weights can be further exploited by calculating percentage weights based on the actual length of each sequence. To give an illustrative example, set {7, 19} edge has weight 64 but sequence 7 has length 110 while sequence 19 has length 148. Therefore, sequence 7 has 58% coverage while sequence 19 has 43% coverage. This information can be further examined to give more insights on our social networks and create a directed graph by assigning possible influences, such as, sequence 7 used a code snippet from sequence 19 instead of the opposite.

V. CONLCUSIONS

The paper proposes a novel source code detection method that can identify all the existing common code patterns irrespectively of their length using a commodity computer. The method utilizes an advanced data structure (LERP-RSA) in order to store the code sequences and facilitate ARPaD algorithm to thoroughly analyze and detect the common patterns.

The detected results are then visualized in order to help the analyst to understand the correlations between the common patterns in the code sequences and the corresponding social networks of the programmers who have used similar code is created. The results have shown that the method is capable to address various insertion attacks who are usually employed by students who want to spoof the results of the traditional code plagiarism detection systems combing all the common patterns identified in the different code sequences.

Another important contribution of the work, is the construction of a dataset of similar code snippets that will be available online and can be used as a reference dataset by other researchers in order to assess different code similarity techniques.

Finally, the proposed method can yet be improved using more sophisticated metrics to calculate the overall similarity among the code snippets and produce more accurate results.

REFERENCES

[1] StackOverflow.com, https://stackoverflow.com/company (accessed Sep. 10, 2020)

[2] StackExchange.com, https://stackexchange.com/about (accessed Sep. 10, 2020)

[3] Quora.com, https://www.quora.com/about (accessed Sep. 10, 2020)

[4] Reddit.com, https://www.redditinc.com (accessed Sep. 10, 2020)

[5] Google Groups, https://groups.google.com/forum/#!overview (accessed Sep. 10, 2020)

[6] GitHub.com, https://github.com (accessed Sep. 10, 2020)

[7] Atlassian BitBucket, https://bitbucket.org/product (accessed Sep. 10, 2020)

[8] SourceForge.net, https://sourceforge.net (accessed Sep. 10, 2020)

[9] Xu X, Liu C, Feng Q, Yin H, Song L, Song D. Neural network-based graph embedding for cross-platform binary code similarity detection. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security 2017 Oct 30 (pp. 363-376).

[10] Arwin, C., & Tahaghoghi, S. M. (2006, January). Plagiarism detection across programming languages. In Proceedings of the 29th Australasian Computer Science Conference-Volume 48 (pp. 277-286).

[11] Whale, G. (1986, January). Detection of plagiarism in student programs. In Proceedings of the 9th Australian Computer Science Conference (pp. 231-241). Australian Computer Society.

[12] Freire, M., Cebrián, M., & Del Rosal, E. (2007). AC: An integrated source code plagiarism detection environment. arXiv preprint cs.IT/0703136.

[13] Joy, M. S., Sinclair, J. E., Boyatt, R., Yau, J. K., & Cosma, G. (2013). Student perspectives on source-code plagiarism. International Journal for Educational Integrity, 9(1).

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

[15] Gibson, J. Paul. "Software reuse and plagiarism: a code of practice." In Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education, pp. 55-59. 2009.

[16] Cosma, G., & Joy, M. (2012). Evaluating the performance of lsa for source-code plagiarism detection. Informatica, 36(4).

[17] Kustanto, C., & Liem, I. (2009, May). Automatic source code plagiarism detection. In 2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing (pp. 481-486). IEEE.

[18] Burrows, S., & Tahaghoghi, S. M. (2007, December). Source code authorship attribution using n-grams. In Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University (pp. 32-39). Citeseer.

[19] Ciesielski, V., Wu, N., & Tahaghoghi, S. (2008, July). Evolving similarity functions for code plagiarism detection. In Proceedings of the 10th annual conference on Genetic and evolutionary computation (pp. 1453-1460).

[20] Karnalim, Oscar. "Detecting source code plagiarism on introductory programming course assignments using a bytecode approach." In 2016 International Conference on Information & Communication Technology and Systems (ICTS), pp. 63-68. IEEE, 2016.

[21] Moussiades, L., & Vakali, A. (2005). PDetect: A clustering approach for detecting plagiarism in source code datasets. The computer journal, 48(6), 651-661.

[22] Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A., & Zou, W. (2018, September). αdiff: cross-version binary code similarity detection with dnn. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (pp. 667-678).

[23] Novak, M., Joy, M., & Kermek, D. (2019). Source-code similarity detection and detection tools Used in academia: a systematic review. ACM Transactions on Computing Education (TOCE), 19(3), 1-37.

[24] Chilowicz, M., Duris, E., & Roussel, G. (2009, May). Syntax tree fingerprinting for source code similarity detection. In *2009 IEEE 17th International Conference on Program Comprehension* (pp. 243-247). IEEE.

[25] Xylogiannopoulos, K. F. “Data structures, algorithms and applications for big data analytics: single, multiple and all repeated patterns detection in discrete sequences.” PhD thesis, University of Calgary, 2017

[26] Xylogiannopoulos, K.F., Karampelas, P., Alhajj, R. “Repeated patterns detection in big data using classification and parallelism on LERP reduced suffix arrays” Appl. Intell. 45(3), 2016, pp. 567– 561

[27] Xylogiannopoulos, K.F., Karampelas, P., Alhajj, R. “Analyzing very large time series using suffix arrays” Appl. Intell. 41(3), 2014, pp.941–955

[28] Xylogiannopoulos, K. F., Karampelas, P., Alhajj, R., (2019) “Text Mining for Malware Classification Using Multivariate All Repeated Patterns Detection.” ASONAM 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining – Foundations and Applications of Big Data Analytics (Vancouver, BC, Canada), pp. 887-894

[29] Xylogiannopoulos, K. F., Karampelas, P., Alhajj, R., (2018) “Text Mining for Plagiarism Detection: Multivariate Pattern Detection for Recognition of Text Similarities.” ASONAM 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining – Foundations and Applications of Big Data Analytics (Barcelona, Spain), pp. 938-945