X9: An Obfuscation Resilient Approach for Source Code Plagiarism Detection in Virtual Learning Environments

Bruno Prado, Kalil Bispo and Raul Andrade
Department of Computing, Federal University of Sergipe, Marechal Rondon Avenue, São Cristóvão, Brazil

Keywords: Plagiarism Detection, Computer Programming, E-learning, Source Code Similarity, Re-factoring, Obfuscation.

Abstract: In computer programming courses programming assignments are almost mandatory, especially in a virtual classroom environment. However, the source code plagiarism is a major issue in evaluation of students, since it prevents a fair assessment of their programming skills. This paper proposes an obfuscation resilient approach based on the static and dynamic source code analysis in order to detect and discourage plagiarized solutions. Rather than focusing on the programming language syntax which is susceptible to lexical and structural re-factoring, an instruction and an execution flow semantic analysis is performed to compare the behavior of source code. Experiments were based on case studies from real graduation projects and automatic obfuscation methods, showing a high accuracy and robustness in plagiarism assessments.

1 INTRODUCTION

In computer science graduation, practical exercises for programming language courses are essential to improve the learning process (Kikuchi et al., 2014), specially through e-teaching platforms, such as *Moodle* (Moodle, 2017). Although, the effectiveness of virtual learning environments consist on ensuring that students would not to be able to plagiarize, that is, to be evaluated by someone else's programming competence. Source code plagiarism consists in the partial or complete reuse of a third-party solution instead of writing its own source code. Despite possible copyright infringements, plagiarism in programming exercises prevents a fair evaluation of students and creates the burden of proof to ensure similarity assessment (Cosma and Joy, 2012; Joy and Luck, 1999).

Due to the long time spent in manual verification of source code, it is mandatory the use of automatic source code plagiarism detection tools to verify similarities between files. The major challenge for these tools is the detection of false positive cases in similar solutions of simple problems from introductory courses (Modiba et al., 2016). To properly nullify plagiarized solutions, the similarity report selects implementations using a defined threshold which must be reviewed by instructor.

In this paper, we propose a plagiarism detection tool based on both static and dynamic analysis. The use of multiple approaches, greatly reduces the chances of false positive cases, while the false negative condition still can be properly detected, due to multiple metrics analyzed. The main contributions of proposed approach are:

- Currently multiple languages are supported, such as *Assembly*, *C/C++*, *Go*, *Java*, *Pascal* and *Python*. There is no limitation including another languages, however, the target specific development tools output must be parsed and analyzed;
- Static analysis relies on extraction of emitted machine instruction by compiler infrastructure. This method greatly reduces the impact of code obfuscation and re-factoring, since compiler optimizes the source code ignoring syntactic modifications;
- By collection of system, libraries and software calls plus profiling traces, the dynamic analysis can check execution flow to focus in the behavior of software. This approach identifies the most relevant parts of application, detecting unused code (*dead code*) or advanced re-factoring techniques.

The paper is organized as follows: in section 2, the related work approaches are briefly described; in section 3, the proposed plagiarism detection flow is shown; in sections 4 and 5, the static and dynamic techniques employed to compare source codes are explained, respectively; in section 6, the methodology and experimental case studies are defined; and in sec-

Prado, B., Bispo, K. and Andrade, R.
X9: An Obfuscation Resilient Approach for Source Code Plagiarism Detection in Virtual Learning Environments.
DOI: 10.5220/0006668705170524
In *Proceedings of the 20th International Conference on Enterprise Information Systems (ICEIS 2018)*, pages 517-524
ISBN: 978-989-758-298-1

tion 7, the proposed approach achievements and future work are summarized.

2 RELATED WORK

Some software plagiarism tools are freely available for evaluation, such as: *JPlag* (KIT, 2017; Prechelt et al., 2002) which converts the source code in string of tokens to compare them using RKR-GST algorithm; *MOSS* (Measure Of Software Similarity) (Aiken, 2017; Schleimer et al., 2003) is based on a string matching algorithm which divides the software in k-grams, a contiguous sub-string with length of k; *Sherlock* (Joy and Luck, 2017; Joy and Luck, 1999) that reduces the software to tokens, listing similar pairs and their degree of similarities in a visual representation of graph; and *SIM* (Grune, 2017; Gitchell and Tran, 1999) measures the structural similarities using a string alignment approach. These tools are used in this paper to evaluate the proposed approach by comparing the similarity results in different case studies.

The token-based approaches (KIT, 2017; Aiken, 2017; Joy and Luck, 2017; Grune, 2017; Muddu et al., 2013) convert provided source codes in sequence of tokens to be compared using RKR-GST or fingerprinting algorithms. These methods are generally more susceptible to code obfuscation methods, such as identification renaming or code merging, for example, once they use source code text as input. Abstract Syntax Tree (AST) methods (Zhao et al., 2015; Kikuchi et al., 2014) convert AST in a hash code and sequence alignment, respectively. AST structuring improves tokenization effectiveness, providing a a more robust source code analysis, despite error rates in arithmetic operations and processing efficiency.

Semantic analysis verifies the equivalence between basic blocks in binary level using fuzzy matching (Luo et al., 2017; Luo et al., 2014) and a matrix of term-by-file containing information about terms frequency counts to find the underlying relationships (Cosma and Joy, 2012). Instead of checking for tokens or structure, the semantic approach is focused on the retrieval of different codes fragments which describe the same function. The limitations of methods are inherent to static analysis which can be impaired by indirect branches and choosing of optimal parameter settings.

The employment of unsupervised learning approach from features extracted from intermediate representation of a compiler infrastructure, such as *GCC* (Yasaswi et al., 2017), represents the extracted features in a n-dimensional space to compare them pairwise, using the Euclidean distance measurement. Despite the obfuscation resilience provided by intermediate representation processing and the machine learning techniques, the feature selection relies on the code optimization selected by the user and includes no dynamic information which can improve performance.

Unlike previous work, the dynamic analysis strategy checks run-time behavior of software in order to verify usage patterns and flow tracing. By the analysis of dynamic stack usage patterns (Park et al., 2015), this work claims that stack usage patterns are invulnerable to syntactic modifications. To retrieve stack information, the software is dynamically instrumented to capture function related instructions. Another method consists in check run-time semantic information (Choi et al., 2013) to investigate similarities in binary codes. By the use of a memory tracer, the region filtering eliminates irrelevant or unnecessary variables to allow sequence alignment comparison of memory regions. Although the effectiveness of run-time behavior analysis, these approaches do not take advantage of static code information which could improve the similarity measurement confidence, specially in I/O intensive tasks, such as reading input (problem) and write the output (answer) to files.

3 THE PROPOSED FLOW

An efficient assessment flow for practical programming exercises relies on the automation of all processes. In this automated environment, the forbidden source code exchange or team collaboration is discouraged due to high probability of penalties, even in large class sizes. Besides the automated evaluation processes, the key concept of proposed plagiarism detection flow is the semantic analysis of software. This analysis is performed by the use of both static information at instruction level and dynamic behavior at function invocation patterns and traces.

The ability to detect plagiarism is useless if the online submission system is unable to authenticate the students. Once properly identified by their institutional accounts, the students are more aware of the risks of engaging in source code plagiarism and they will balance the work to avoid detection versus the effort to solve the problem by themselves. In order to prove plagiarism, the source code similarity is the major metric to determine how to classify a pair of solutions. A similarity threshold must be defined to assess the plagiarism, but the instructor is able to visualize and to review the assessments. Even in the situations where the detected similarity is below than

defined threshold, the instructor can define a suspicious similarity range to verify by sampling if some unnoticed approach is being used to avoid detection.

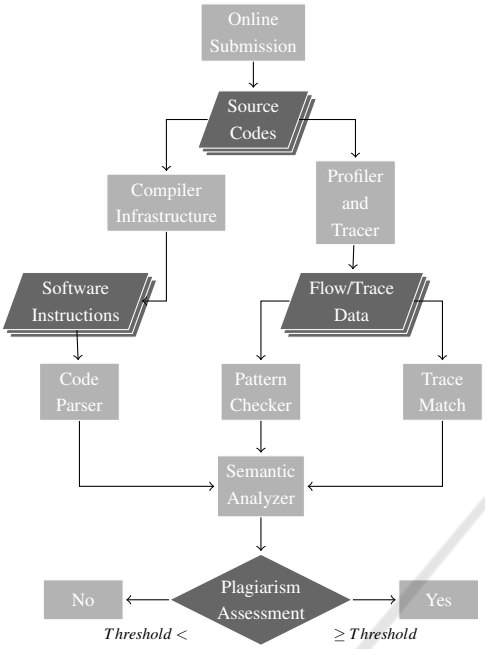


Figure 1: Plagiarism detection flowchart.

In Figure 1, a overview of proposed plagiarism detection flow can be summarized in the following processes:

- Online Submission: the user is authenticated to upload *Source Codes* which is compiled to check for syntactic errors or forbidden built-in libraries importation, such as algorithms and data structures;
- Compiler Infrastructure: to generate a set of *Software Instructions* from *Source Codes*, the available compiler optimization options are used in default, code size and execution time levels;
- Profiler and Tracer: by the capture of profiling information, libraries and system calls, the software execution flow can be analyzed for invocation patterns and tracing comparison;
- Code Parser: this step consists in the parsing of assembly files generated to retrieve the semantic behavior from source code functions;
- Pattern Checker: the software profiling identifies core functions of application, that is, the functions with largest running time and number of calls;
- Trace Match: all the software requests to the dynamic libraries or to the operating system are compared to determine system requests similarities;
- Semantic Analyzer: by the analysis of parsed codes, the invocation patterns and execution traces, the semantic similarity of a set of *Source Codes* is calculated;
- Plagiarism Assessment: in this phase, using the defined *Threshold* and *Source Codes* as input, the plagiarism is discarded (*No*) or confirmed (*Yes*) for all pairs of solutions in a HTML report.

4 INSTRUCTION LEVEL SEMANTIC ANALYSIS

A plagiarized source code could be defined as the improper reuse of software without changing the behavior of a third-party solution. This source code obfuscation to avoid the plagiarism detection is mostly based on syntactic and structural modifications which can be overcome by a resilient semantic analysis.

4.1 Behavior Assessment

Instead of checking for syntactic similarities, the proposed semantic analysis is focused on the extraction of software behavior from instructions emitted by compiler.

```
int search1(int x, int* V, unsigned int n) {
    unsigned int i = 0;
    int index = -1;
    while(i < n && index == -1) {
       if(V[i] == x) index = i;
       i++;
    }
    return index;
}

int search2(int* X, int v, int N) {
    int pos = -1;
    for(int a = 0; a < N && pos < 0; a = a + 1)
       pos = (X[a] != v) ? (-1) : (a);
    return pos;
}
```

Figure 2: Sequential search source code examples.

As can be seen in Figure 2, the two sequential search examples provided have a quite distinct syntax, but exactly the same behavior. This is a straightforward example of how difficult is to assess the source code plagiarism without checking underlying behavior.

A side-by-side comparison of the functions *search*1 and *search*2 is shown in Figure 3, where

```
search1:          search2:
    ⋮                 ⋮
    -                 + cltq
    ⋮                 ⋮
    -                 + jmp
    -                 + mov
    ⋮                 ⋮
    ! jae             ! jge
    cmpl              cmpl
    ! je              ! js
    ⋮                 ⋮
```

Figure 3: Side-by-side instructions comparison.

$-$ and $+$ denote a missing or added instruction, respectively, and ! highlights the difference between instructions. Applying Longest Common Subsequence (*LCS*) algorithm (Bergroth et al., 2000), a similarity of 92% between the functions is detected.

4.2 Similarity Score

All the functions of source codes are processed as position-independent code which allows the resiliency against interchange of source code placement and merging of sequentially called functions.

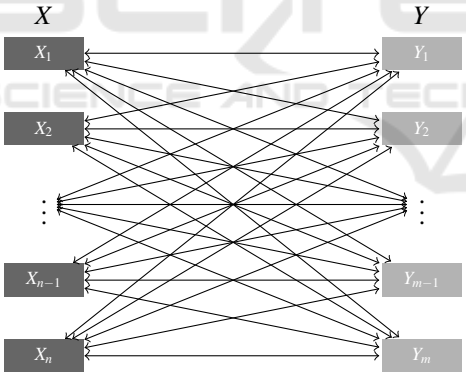


Figure 4: Position-independent similarity check.

Considering the source files X and Y, as shown in Figure 4, the similarity score of each function X_i versus Y_j, where $i \in [1,n]$ and $j \in [1,m]$, consists in the best match obtained from *LCS* algorithm. In Equation 1, the maximum similarity of comparison of function X_i with the functions of Y, that is, the best match of X_i and Y_j.

$$\rho(X_i, Y) = max \sum_{j=1}^{j=m} LCS(X_i, Y_j) \tag{1}$$

Once the function similarity is known, the next step is to calculate the weight ω of this function in whole source code. As detailed in Equation 2, the function weight ω is obtained through the division of size of function X_i by all X functions.

$$\omega(X_i) = \frac{|X_i|}{\sum_{k=1}^{k=n} |X_k|} \tag{2}$$

Finally, the similarity score between X and Y source codes is defined by the sum of weighted similarities of X_i functions against the Y source code, as defined in Equation 3.

$$\sigma(X, Y) = \sum_{i=1}^{i=n} \omega_i \times \rho(X_i, Y) \tag{3}$$

5 EXECUTION FLOW TRACE

No matter how advanced static source code analysis is, the plagiarist can include large amounts of unused or rarely called functions (*dead code*), just to deceive similarity comparison. Instead of checking software statically, the source code profiling aims to retrieve dynamic behavior by the analysis of execution flow, that is, by the fetching all function, libraries and system calls, plus profiling traces.

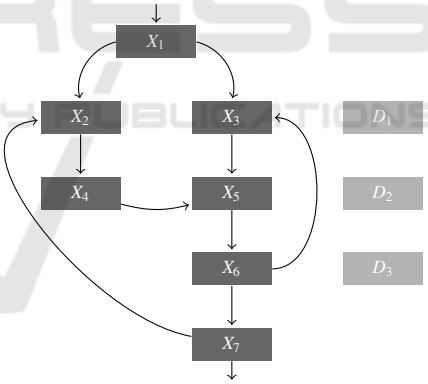


Figure 5: A call-graph generated from the execution.

The execution flow analysis consists in tracing of function X_i call history from software run, as illustrated in Figure 5. The dead code functions D_j can be easily detected due to null or very low influence in overall function call tracing. Hence, the run-time semantic information acquired from dynamic behavior is much harder to be obfuscated by re-factoring techniques.

5.1 Library and System Calls Capture

In a Linux environment, there are tracing tools to dynamically intercept the library calls and systems calls

requested by a process, using *ltrace* (Cespedes, 2017) and *strace* (strace project, 2017), respectively. This information is quite useful because it provides a black box view of how software interacts with the libraries and system.

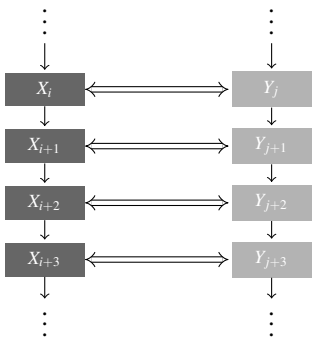


Figure 6: The library or system calls trace comparison.

Once the source code traces were retrieved from applications X and Y, the *LCS* algorithm is applied to compare the run-time calls, as shown in Figure 6. Acquired information shows the sequence of calls performed by each solution, providing a fingerprint of resource usage (e.g., memory allocation) and services requested (e.g., I/O operations).

5.2 Software Profiling

Besides the trace capabilities, the most valuable metric provided by software profiling tools, such as *GNU gprof* (GNU, 2017) or *Valgrind* (Valgrind, 2017), is the software invocation patterns. Instead of considering the size of source code, the application profiling allows the identification of core functions by determining the amount of calls performed and overall time elapsed.

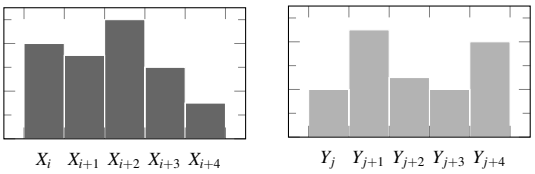


Figure 7: Example of X and Y invocation patterns.

By the comparison of most important functions, sorted by descending order of time spent in execution, the proposed approach focus in *core codes* (Park et al., 2015) which run-time behavior are hard to be replaced or removed. In Figure 7 it is shown an example of a set of functions X_i and Y_j invocation patterns from X and Y applications, respectively, which can be used to determine the execution profile.

6 EXPERIMENTAL RESULTS

To properly evaluate the proposed techniques, in next three subsections the methodology of performed experiments is detailed, the similarity comparison scores from available tools is shown and a discussion of experiments is held.

6.1 Methodology

The burden of proof (Cosma and Joy, 2012; Joy and Luck, 1999) is certainly the major challenge in plagiarism assessment. Some source code modifications can make it hard or impossible the demonstration of similarities due to the complexity of changes made. In order to avoid questions about applied effort in source code re-factoring techniques, the obfuscation tool *Stunnix C/C++* (Stunnix, 2017) was used in attempt to deceive plagiarism tools by: stripping spaces and newlines; *MD5* hashing of the identifiers; and converting strings to hex escape codes.

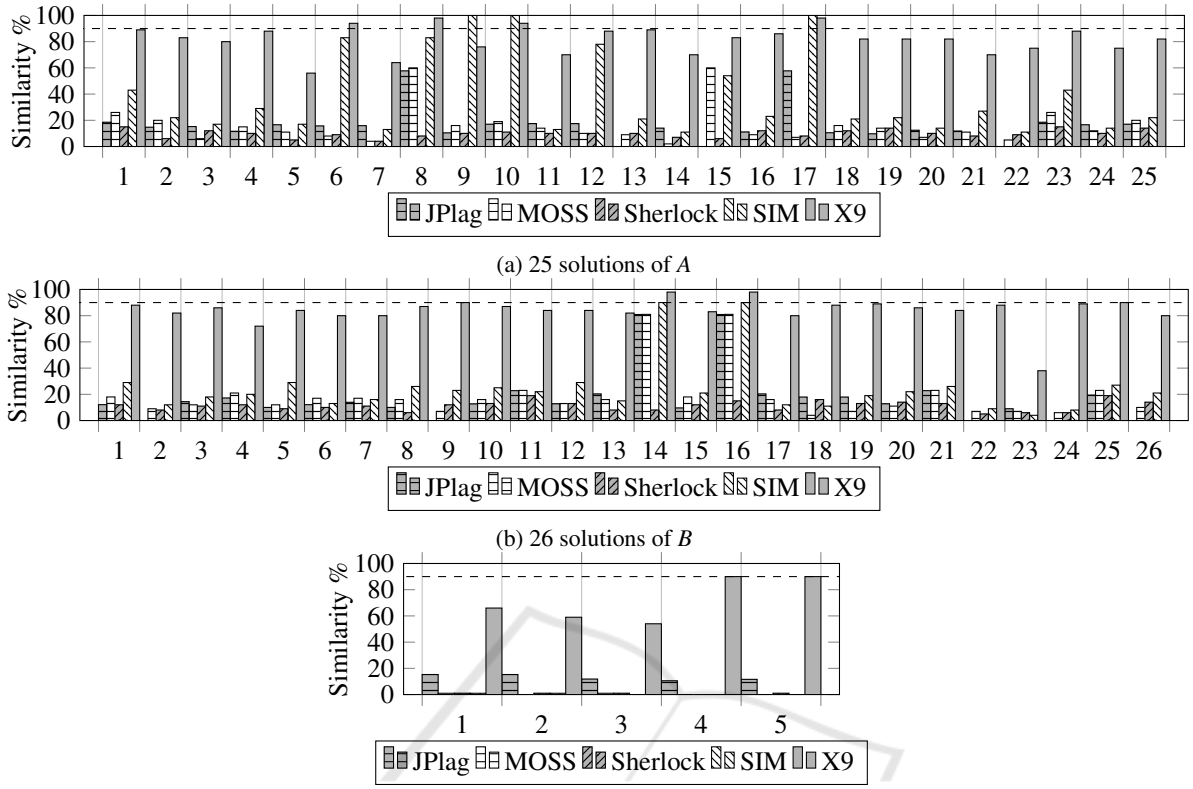
Four different possibilities can occur while code similarity is being performed. The intended scenario occurs when innocent solutions are freed (true negative = TN) and plagiarism is correctly detected (true positive = TP). On the other hand, there are unwanted scenarios in which plagiarized code is not uncovered (false negative = FN) and not guilty implementations are wrongly accused of plagiarism (false positive = FP). The accuracy of plagiarism detection, denoted by $\alpha \in [0,1]$ in Equation 4, can be defined as the sum of properly asserted solutions divided by the total number of solutions. A similarity threshold of 90% was chosen to classify a solution as plagiarism.

$$\alpha = \frac{TN+TP}{TN+FN+TP+FP} \tag{4}$$

To accomplish a realistic tool comparison, three sets of computer science graduation projects written in *C/C++* were selected from a solution database. These exercises addresses the implementation of a ship load sorting using *Mergesort* algorithm, labeled as set *A*; a *Knuth-Morris-Pratt* (*KMP*) gene finder for disease checking in DNA sequence, labeled as set *B*; and a instruction set simulator for a hypothetical *RISC* machine, labeled as set *C*. Considering only fully correct solutions to compare equivalent source codes, the sets had their assessments by X9 shown in Table 1.

Table 1: The assessments for sets *A*, *B* and *C*.

Set	TN	TP	FN	FP
A	23	1	0	1
B	24	1	0	1
C	4	0	0	1



(a) 25 solutions of *A*

(b) 26 solutions of *B*

(c) 5 solutions of *C*

Figure 8: The similarity analysis of sets *A*, *B* and *C*.

All the assessments in Table 1 were performed by human source code inspection for each pair of solutions. The false positive cases are closely related to reuse of source code fragments from lectures or references. Also, the low complexity of problems with lines of code ranging from 146 to 462 in set *A* and from 115 to 240 in set *B* could lead to this misclassification. In set *C*, a pair of implementations had 90% similarity level, although no strong evidence of plagiarism were found in investigation.

6.2 Tool Comparison

As described in methodology, the evaluation of plagiarism approaches is based on its accuracy, denoted by α, that is, the measurement of how distant is classification from correct assessment stated for sets *A*, *B* and *C*.

In Table 2, the tools accuracy for proposed case studies *A*, *B* and *C* are shown. None of the compared tools were capable to detect TP in sets *A* and *B*, while *SIM* tool was the only one to wrongly classify three solutions of set *A* as plagiarism.

For set *A*, in the TP cases which similarities were above of 10% (Figure 8a), *JPlag* and *SIM* achieved, respectively, 57.7% and 82% similarities, while X9 reported a 98% similarity. Considering set *B*, *Sherlock* was the only to report a low similarity of 1% for a TP solution, assessed by X9 with a 98% similarity, although *JPlag*, *MOSS* and *SIM* showed 80.9%, 81% and 90% similarities (Figure 8b), respectively. No TP solution was detected in set *C* by any tool (Figure 8c), just a FP case from X9 approach with a 90% similarity.

Table 2: The accuracy of tools in sets *A*, *B* and *C*.

Tool	*A*	*B*	*C*
JPlag	0.9600	0.9615	1.0000
MOSS	0.9600	0.9615	1.0000
Sherlock	0.9600	0.9615	1.0000
SIM	0.8400	1.0000	1.0000
X9	**0.9600**	**0.9615**	**0.8000**

To verify the resilience of plagiarism tools to source code obfuscation, the sets A', B' and C' were generated by *Stunnix C/C++* (Stunnix, 2017) from plain sets *A*, *B* and *C*, respectively. In this case study, the plagiarism tool evaluation focuses on the ability to detect an automated source code obfuscation, that is,

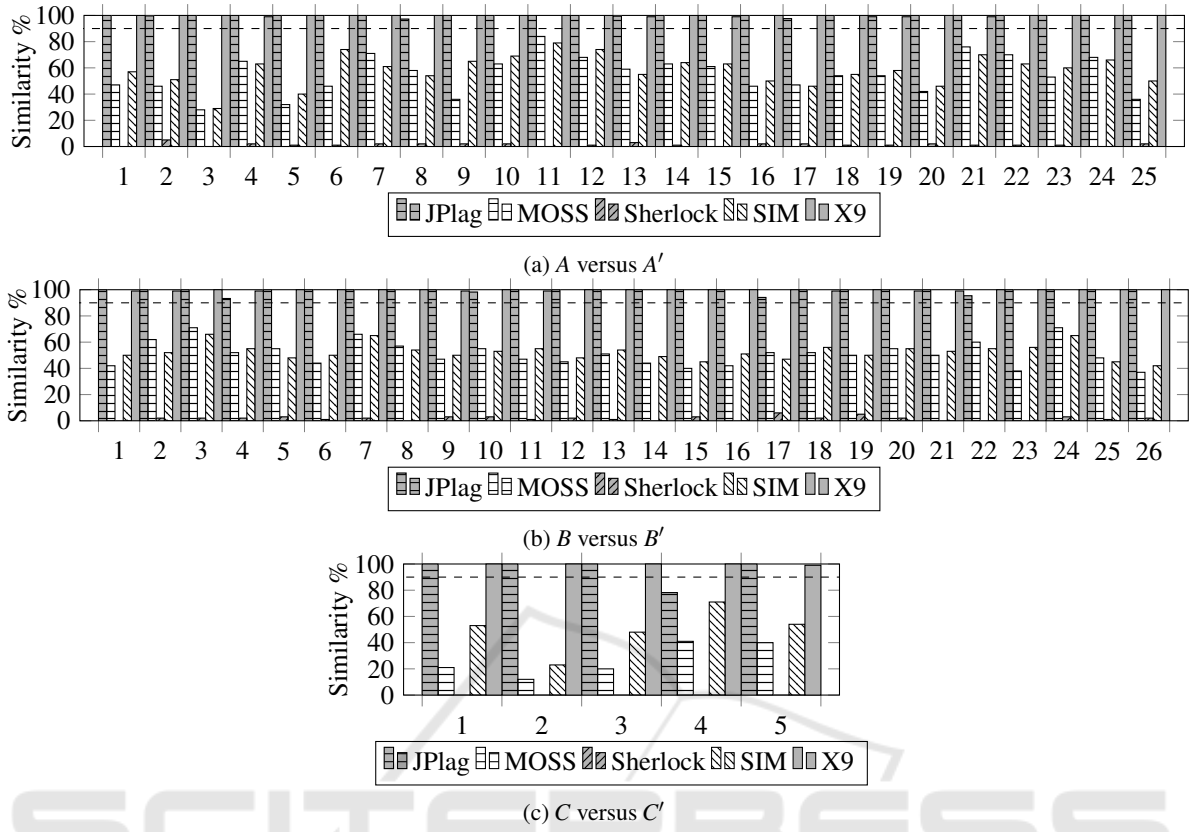


Figure 9: The similarity between plain and obfuscated sets.

the evaluation of $A \equiv A'$ (Figure 9a), $B \equiv B'$ (Figure 9b) and $C \equiv C'$ (Figure 9c) statements.

Table 3: The accuracy checking for sets A', B' and C'.

Tool	A'	B'	C'
JPlag	1.0000	1.0000	0.8000
MOSS	0.0000	0.0000	0.0000
Sherlock	0.0000	0.0000	0.0000
SIM	0.0000	0.0000	0.0000
X9	**1.0000**	**1.0000**	**1.0000**

The accuracy of tools used to detect plagiarism from obfuscated source codes is shown in Table 3. In all sets, X9 detected plagiarized code with at least 99% of similarity, thus, achieving a perfect accuracy. Considering both sets A' and B', *JPlag* was the only tool to identify the obfuscated solution with a 94% similarity at least. In set C', *JPlag* was the only tool to detect plagiarism in 4 TP solutions with a 100% similarity, while 1 FN case achieved a 78.2% level.

6.3 Discussion

None of the plagiarism analysis performed by related tools contradicted the assessments shown in Table 1, except for a clear classification issue from *SIM*, while analyzing the set A. These results can confirm the human assertions made about sets A, B and C, but the case study performed using automated obfuscation sets A', B' and C' avoided questions about the proof or the refutation of the plagiarism evaluations.

Undesired FN cases are critical for plagiarism assessment, since they define the reliability of automated checking. *JPlag* had the best performance, missing only 1 FN in obfuscated sets A', B' and C', while *Sherlock* achieved worst results, detecting none of plagiarized solutions. Despite good results, *JPlag* was unable to identify 2 TP cases in sets A and B which have been detected by X9 approach with a 98% similarity.

The token-based algorithm of *JPlag* was successful due to the fact of ignoring most of syntactic information of source code to generate tokens from declarations and operations performed. The 2 TP undetected cases confirm that *JPlag* is vulnerable to targeted source code level attacks, such as statement replacement or reordering, which can be highly successful with low effort. In order to deceive X9, it would be necessary a source code re-factoring to dra-

matically change the generated operations and the execution profile, thus, as hard as implementing a solution from scratch.

7 CONCLUSION

This paper has proposed a robust approach for source code plagiarism detection, combining static information from software instructions and dynamic behavior from acquired profile and traces. The semantic analysis techniques were effective in detecting plagiarized solutions in a real case study scenario. Despite minor false positive cases, the X9 was able to detect all plagiarized solutions, even when they were heavily obfuscated with a perfect accuracy level.

The source code similarity comparison is a challenging task and the proposed approach has plenty space for improvements, such as: the definition of a larger number of labeled solutions in experiments, including interpreted and script based languages, which can improve X9 evaluation; in semantic analysis, the results can be improved by ignoring known algorithms from lectures or mandatory library/system call requests, which would avoid FP cases; and the release of X9 tool for public usage as web service, allowing further feedback to enhance the development.

REFERENCES

Aiken, A. (2017). Moss - measure of software similarity.

Bergroth, L., Hakonen, H., and Raita, T. (2000). A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pages 39–48.

Cespedes, J. (2017). ltrace - a library call tracer.

Choi, Y., Park, Y., Choi, J., Cho, S.-j., and Han, H. (2013). Ramc: Runtime abstract memory context based plagiarism detection in binary code. In *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '13, pages 67:1–67:7, New York, NY, USA. ACM.

Cosma, G. and Joy, M. (2012). An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE Transactions on Computers*, 61(3):379–394.

Gitchell, D. and Tran, N. (1999). Sim: A utility for detecting similarity in computer programs. *SIGCSE Bull.*, 31(1):266–270.

GNU (2017). Gnu profiler.

Grune, D. (2017). The software and text similarity tester sim.

Joy, M. and Luck, M. (1999). Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133.

Joy, M. and Luck, M. (2017). The sherlock plagiarism detector.

Kikuchi, H., Goto, T., Wakatsuki, M., and Nishino, T. (2014). A source code plagiarism detecting method using alignment with abstract syntax tree elements. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6.

KIT, K. I. o. T. (2017). Jplag - detecting software plagiarism.

Luo, L., Ming, J., Wu, D., Liu, P., and Zhu, S. (2014). Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 389–400, New York, NY, USA. ACM.

Luo, L., Ming, J., Wu, D., Liu, P., and Zhu, S. (2017). Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering*, PP(99):1–1.

Modiba, P., Pieterse, V., and Haskins, B. (2016). Evaluating plagiarism detection software for introductory programming assignments. In *Proceedings of the Computer Science Education Research Conference 2016*, CSERC '16, pages 37–46, New York, NY, USA. ACM.

Moodle (2017). Moodle - open-source learning platform.

Muddu, B., Asadullah, A., and Bhat, V. (2013). Cpdp: A robust technique for plagiarism detection in source code. In *Proceedings of the 7th International Workshop on Software Clones*, IWSC '13, pages 39–45, Piscataway, NJ, USA. IEEE Press.

Park, J., Son, D., Kang, D., Choi, J., and Jeon, G. (2015). Software similarity analysis based on dynamic stack usage patterns. In *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*, RACS, pages 285–290, New York, NY, USA. ACM.

Prechelt, L., Malpohl, G., and Philippsen, M. (2002). Finding plagiarisms among a set of programs with jplag. *j-jucs*, 8(11):1016–1038.

Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA. ACM.

strace project (2017). strace - trace system calls and signals.

Stunnix (2017). Cxx-obfus: Stunnix c/c++ obfuscator.

Valgrind (2017). Valgrind: Instrumentation framework.

Yasaswi, J., Kailash, S., Chilupuri, A., Purini, S., and Jawahar, C. V. (2017). Unsupervised learning based approach for plagiarism detection in programming assignments. In *Proceedings of the 10th Innovations in Software Engineering Conference*, ISEC '17, pages 117–121, New York, NY, USA. ACM.

Zhao, J., Xia, K., Fu, Y., and Cui, B. (2015). An ast-based code plagiarism detection algorithm. In *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, pages 178–182.