Kubiuk, Yevhenii; Kyselov, Gennadiy

Article

Development of an algorithm for code clone detection in source code based on abstract syntax tree

Reference: Kubiuk, Yevhenii/Kyselov, Gennadiy (2023). Development of an algorithm for code clone detection in source code based on abstract syntax tree. In: Technology audit and production reserves 4 (2/72), S. 33 - 36. https://journals.uran.ua/tarp/article/download/286472/280637/661682. doi:10.15587/2706-5448.2023.286472.

UDC 004.85:004.032.2
DOI: 10.15587/2706-5448.2023.286472

Yevhenii Kubiuk,
Gennadiy Kyselov

DEVELOPMENT OF AN ALGORITHM FOR CODE CLONE DETECTION IN SOURCE CODE BASED ON ABSTRACT SYNTAX TREE

The object of research of this work is the algorithm for searching for duplicates in the program code based on the Abstract Syntaxes Tree (AST). The main tasks solved within the framework of this study are the detection of duplicate code and the search for vulnerabilities in the program code.

The obtained results showed that the proposed algorithm is resistant to type 1 and 2 clones, which means its effectiveness in detecting similar code fragments with identical or variant text. However, for type 3 and 4 clones, the algorithm may show less efficiency due to the change in the AST structure for these types of clones.

Experimental studies of the proposed algorithm showed that the algorithm can detect matches between unrelated files due to the presence of typical AST chains present in many programs. This can lead to a certain level of false positives in the detection of duplicates.

Testing of the algorithm in the task of finding vulnerabilities showed that:

1. The best recognition is observed for the «SQL injection» vulnerability, but it also has the highest number of false positives.

2. Memory leak and null pointer dereferencing vulnerabilities are detected with equal effectiveness and false positives.

3. «Buffer overflow» has the lowest recognition rate but fewer false positives compared to «SQL injection».

The study showed that the use of AST allows for the effective detection of duplicate code and vulnerabilities in the software code. The developed tool can help software developers reduce maintenance efforts, improve code quality, and ensure software product security.

Keywords: *clone detection, abstract syntax tree, AST, hashing, vulnerability search, false alarms.*

Received date: 24.06.2023
Accepted date: 21.08.2023
Published date: 29.08.2023

How to cite
Kubiuk, Y., Kyselov, G. (2023). Development of an algorithm for code clone detection in source code based on abstract syntax tree. Technology Audit and Production Reserves, 4 (2 (72)), 33–36. doi: https://doi.org/10.15587/2706-5448.2023.286472

1. Introduction

In the modern world, the detection of duplicates in the program code is an important scientific and technical task, which requires adaptation of the developed algorithms to the requirements of speed and accuracy. The problem of code duplication has a significant impact on the efficiency of the software development process, as it leads to unnecessary maintenance effort, delays in making changes, and overall degradation of code quality. In addition, the presence of duplicates increases the risk of errors, because making changes to one piece of code may require repeated modification of all its copies.

The application of algorithms for finding duplicate code also plays an important role in the context of detecting vulnerabilities in software code. Duplicate code can be an indication of potential vulnerabilities because vulnerabilities can be transferred or replicated to multiple locations in the software. Thus, the use of duplicate detection algorithms helps to identify these clones and identify potential security issues, thereby improving the quality and reliability of the software product.

Duplicate code, or clone, can be defined as a piece of code that is similar to another piece of code in some way. According to the generally accepted taxonomy of code duplicates [1], the following types of clones are distinguished:

– Type-1: identical code except for tab characters.
– Type-2: code fragments are structurally and syntactically identical, only user-defined identifiers such as variable, type or function names and comments change.
– Type-3: combination of Type-1 and Type-2 clones with additional modifications of operators, functions and permutations in the code.
– Type-4: code with similar semantics that performs the same business task, but the code structure is different.

The problem of detecting duplicate code has been studied for a long time, so there is a significant amount of research in this area [2]. Despite this, to date there is no generally accepted classification of approaches to duplicate detection. Therefore, let's focus on the work [3], in which the authors identified the following approaches: textual, lexical, tree-based, metrics-based, semantic and hybrid.

Methods based on text [4] and lexical [5] approaches have the following limitations. They do not use information about the general structure of the code, which negatively affects the classification accuracy, and are also not effective for detecting Type-4 semantic clones.

ISSN 2664-9969

Methods based on metrics achieve high accuracy of duplicate detection, but they are not effective in the tasks of detecting vulnerabilities in software code [6, 7]. This is due to the fact that the presence of vulnerable code has a low impact on the value of the metrics, which is not sufficient to define a piece of code as plagiarism, or in the context of this task – as vulnerability.

In this work, the emphasis is on the tree-based approach, as it allows detecting clones of all four types, and also has higher accuracy compared to other approaches [8]. Another advantage of the tree approach is its successful application in the tasks of detecting vulnerabilities [9] in the code [10, 11]. This is explained by the fact that the tree-like structure describing the program code conveys not only information about the available tokens in the code, but also the semantic relationships between them.

The aim of research is to evaluate the effectiveness of tree-based methods in the context of identifying duplicate code and finding potential vulnerabilities in software code. The developed tool allows to provide an effective and accurate (~98.3 %) procedure for searching for plagiarism in the code, focusing on C/C++ programming languages. In addition, it is also capable of detecting code snippets that may contain potential vulnerabilities or security issues. This work is aimed at developing a tool that combines high efficiency and accuracy of duplicate detection with reliable detection of potential vulnerabilities in software code, which will contribute to improving the quality and security of software.

2. Materials and Methods

The object of research in this work is the algorithm for searching for duplicates in the program code, based on the use of an abstract syntax tree (AST). An AST is a data structure that represents the syntactic structure of software code, allowing it to be represented as a tree, where nodes correspond to syntactic constructs and edges show the relationships between them.

The algorithm developed in this paper uses AST to detect duplicate code. It analyzes the structural and syntactic features of the program code by comparing AST subtrees and finding similar fragments. This approach allows detecting not only textually similar fragments, but also clones with similar semantics, which increases the efficiency of the duplicate detection process.

The algorithm for finding duplicate code works as follows:

– *Building an AST*: For a file with code *A* (the source code against which other files are compared), an AST is first built. AST represents the code structure and its semantics.

– *Hashing of AST nodes*: Sequences of AST nodes of size *N* are hashed using the SHA256 algorithm [12]. Each node in the AST has its own unique hash.

– *Building an AST for file B*: Similar to step 1, an AST is built for the code file *B* to be checked for plagiarism.

– *Hashing AST Nodes for File B*: The AST Node Sequences of size *N* for File *B* are hashed using SHA256. Each node receives its own unique hash.

– *Comparison of hashes*: The number of hashes of file *B* that are present in file *A* is compared. This can be done by comparing two lists of hashes.

– *Calculation of the percentage of matches*: The percentage of matches is calculated by dividing the number of matching hashes by the total number of hashes of file *B*.

– *Threshold search*: Threshold *T* is used to decide whether file *B* is a plagiarism of file *A*. If the percentage of matches is greater than threshold *T*, then file *B* is considered plagiarized.

Schematically, this algorithm can be depicted in Fig. 1.

The size of the window *N* affects the quality of plagiarism recognition, so the following algorithm was used to determine the optimal value of the window size:

For each value of the window size from 1 to *K*, a plagiarism search was performed, and the average percentage of plagiarism was calculated for files with clone type 4 as well as for the original files. Accordingly, the first value will be considered the upper limit for plagiarism, and the second value – the lower limit. The optimal value of *N* will be the value at which the distance between the upper and lower limits is maximal.

Thus, the result of the algorithm for choosing the optimal value of the window size for hashing looks like in Fig. 2.

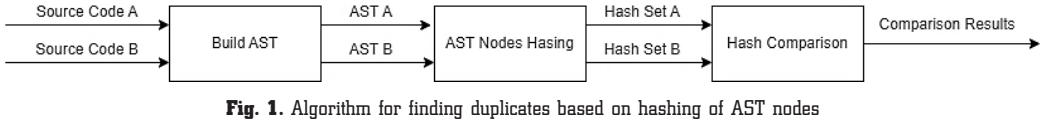

Fig. 1. Algorithm for finding duplicates based on hashing of AST nodes

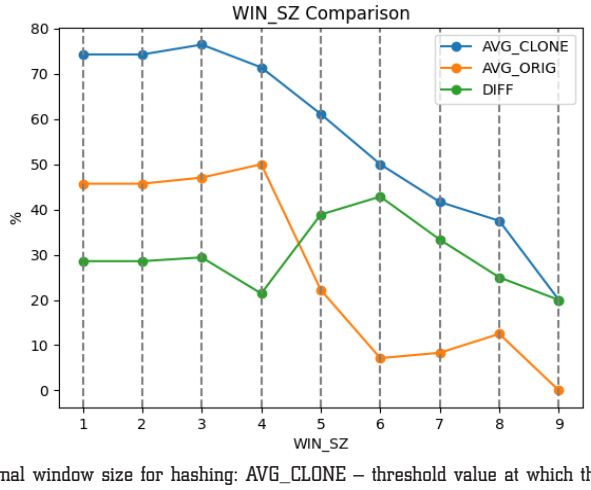

Fig. 2. Determination of the optimal window size for hashing: AVG_CLONE – threshold value at which the code is considered plagiarized; AVG_ORIG – threshold value at which the code is considered original; DIFF – AVG_CLONE and AVG_ORIG difference module

As can be seen from Fig. 2, when the window size is $N=6$, cloned and original code are most clearly distinguished and have the largest intercluster distance.

3. Results and Discussion

In the course of this study, two important problems were identified, namely, the search for duplicates in the program code and the search for vulnerabilities in the program code. In order to study and solve each of these problems, separate experiments were conducted.

The first experiment demonstrates the operation of the algorithm in the task of finding plagiarism in the code. As part of the experiment, a dataset with files of several types was created:

– *original.cpp* – the original file with the code;
– *type_1.cpp* – type 1 clone. Created on the basis of original.cpp with the addition of tab characters;
– *type_2.cpp* – type 2 clone. Created on the basis of original.cpp, in which the variable names were changed;
– *type_3.cpp* – type 3 clone. Created on the basis of original.cpp, in which the values of string literals, some calculation formulas, and tabulation were changed;
– *type_4.cpp* – type 4 clone. Original code that solves the same business problem as original.cpp;
– *original_2.cpp* – original file with code that solves a different business problem than original.cpp.

The dataset consisted of 15 files of each type. In Table 1 presents the results of the algorithm for the above dataset.

Table 1

The results of the algorithm for searching for duplicates

File *A*	File *B*	Average percentage of plagiarism	Recognition accuracy at $T=0.6$
original.cpp	*type_1.cpp*	100 %	100 %
original.cpp	*type_2.cpp*	100 %	100 %
original.cpp	*type_3.cpp*	77.78 %	97.78 %
original.cpp	*type_4.cpp*	61.11 %	95.11 %
original.cpp	*original_2.cpp*	22.57 %	98.67 %

According to Table 1 information, it can be determined that the algorithm is resistant to clones of type 1 and 2. This means that it effectively recognizes similar code fragments that have identical text or some variations in the text.

However, for clones of type 3 and 4, the algorithm may show somewhat lower efficiency, since the structure of the abstract syntactic tree may change significantly for these types of clones. This can complicate the process of recognizing and detecting such clones in the program code.

It is worth noting that matches between two unrelated files, which can be detected by the algorithm, are explained by the presence of typical chains of the abstract syntax tree, which are present in many different code fragments. These typical chains may result from common structural patterns or constructs found in many programs.

The second experiment demonstrates the operation of the algorithm in the task of finding vulnerabilities in software code. As part of the experiment, a dataset with the following types of files was used:

– *code_not_vuln.cpp* – code without vulnerabilities;
– *code_vuln_bo.cpp* – code containing an example of a buffer overflow vulnerability;
– *code_vuln_ml.cpp* – code containing an example of a memory leak vulnerability;
– *code_vuln_nd.cpp* – code containing an example of a null pointer dereferencing vulnerability;
– *code_vuln_si.cpp* – code containing an example of a SQL injection vulnerability;
– *bo.cpp* – an example of a buffer overflow vulnerability;
– *ml.cpp* – an example of a memory leak vulnerability;
– *nd.cpp* – an example of a null pointer dereferencing vulnerability;
– *si.cpp* – an example of a SQL injection vulnerability.

The dataset consisted of 25 type files each for the type *code_not_vuln.cpp* and *code_vuln_*.cpp*. The vulnerability files were presented in a single instance and contained a typical example of the vulnerability.

Table 2 presents the result of the algorithm.

Table 2

The result of the algorithm for finding vulnerabilities

File *A*	File *B*	Average percentage of plagiarism	Recognition accuracy at $T=0.6$
code_not_vuln.cpp	*bo.cpp*	7.14 %	96.00 %
code_not_vuln.cpp	*ml.cpp*	5.47 %	97.60 %
code_not_vuln.cpp	*nd.cpp*	5.56 %	98.72 %
code_not_vuln.cpp	*si.cpp*	11.14 %	94.24 %
code_vuln_bo.cpp	*bo.cpp*	75.40 %	88.16 %
code_vuln_ml.cpp	*ml.cpp*	80.00 %	89.60 %
code_vuln_nd.cpp	*nd.cpp*	82.86 %	87.52 %
code_vuln_si.cpp	*si.cpp*	97.41 %	97.28 %

Analyzing the Table 2, the following conclusions can be drawn regarding the recognition of different types of vulnerabilities. The best detection indicator revealed a vulnerability of the «SQL-injection» type. However, it is worth noting that this vulnerability also has the highest level of false positives among all considered vulnerabilities. This may be because the algorithm finds certain patterns that may look like SQL injection, but are not actually vulnerabilities.

The «memory leak» and «null pointer dereferencing» vulnerabilities are detected with about the same efficiency and have similar false positives. This can be explained by the fact that both vulnerabilities are related to freeing memory, which occurs using the same call (for example, the free() function). This similarity in detection and false positives may be due to common patterns or characteristics of these types of vulnerabilities.

From Table 2, it can be seen that the «buffer overflow» vulnerability has the lowest detection rate among all the considered vulnerabilities. This may be because the difference between vulnerable and non-vulnerable code with respect to buffer overflows can only be expressed through the correct choice of function, such as using strncpy instead of strcpy. However, it is worth noting that the false-positive rate for «buffer overflow» is lower than for «SQL injection».

So, on the basis of the results, it can be said that the recognition of different types of vulnerabilities has its own characteristics and it is worth taking into account specific contextual factors in order to achieve an optimal balance between the recognition efficiency and the number of false signals.

ISSN 2664-9969

The study showed that the use of AST allows for the effective detection of duplicate code and vulnerabilities in the software code. The developed tool can help software developers reduce maintenance efforts, improve code quality, and ensure software product security. Also, this tool can be used to ensure the highest quality of educational processes, for example, to check students' programming laboratory work.

The importance and practical orientation of the research is conditioned, among other things, by the presence of martial law in Ukraine. First, the rapid development of high-quality software code is necessary, and secondly, the training of IT specialists in the remote mode requires checking the independent tasks of students and trainees using formal approaches to control plagiarism in the software code and identifying code vulnerabilities.

When implementing the developed system, several important limitations and aspects that may affect the practical applicability and effectiveness of the obtained results should be taken into account:

1. Limited to only C/C++ programming language.
2. The system does not detect the location where the duplicate is present, but only notes its presence.
3. The values of the selected parameters of the algorithm may be less effective for another data set.

Further research can be aimed at improving the algorithm and increasing the accuracy of its operation, as well as expanding the list of supported programming languages. One potential area of improvement is to extend the capabilities of the algorithm so that it not only provides information about the percentage of plagiarism, but also points to the pieces of code that are plagiarized themselves. To do this, it is possible to develop an additional table that will establish a connection between code fragments and their corresponding hashes at the stage of hashing the nodes of the abstract syntactic tree.

4. Conclusions

In this work, experiments were conducted to identify duplicate code and search for vulnerabilities in the program code. The results of the study showed that the algorithm based on the abstract syntax tree (AST) demonstrates resistance to type 1 and 2 clones, that is, it effectively recognizes similar code fragments with identical or variant text.

However, for clones of type 3 and 4, which are characterized by a change in the AST structure, the algorithm may show less efficiency in detection. This is due to the difficulty of recognizing and detecting such clones that have distinct AST structures compared to normal textual changes.

It is also found that the algorithm can detect matches between unrelated files due to the presence of typical AST strings found in many programs. This can create a certain level of false positives, where the algorithm notices similarities, but there are actually no vulnerabilities.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this research, whether financial, personal, authorship or otherwise, that could affect the research and its results presented in this paper.

Financing

The research was performed without financial support.

Data availability

The manuscript has no associated data.

References

1. Koschke, R. (2007). Survey of research on software clones. *In Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik*. doi: https://doi.org/10.4230/DagSemProc.06301.13
2. Kim, M., Bergman, L., Lau, T., Notkin, D. (2004). An ethnographic study of copy and paste programming practices in OOPL. *Proceedings. 2004 International Symposium on Empirical Software Engineering. ISESE'04,* 83–92. doi: https://doi.org/10.1109/isese.2004.1334896
3. Ain, Q. U., Butt, W. H., Anwar, M. W., Azam, F., Maqbool, B. (2019). A Systematic Review on Code Clone Detection. *IEEE Access, 7,* 86121–86144. doi: https://doi.org/10.1109/access.2019.2918202
4. Kal Viertel, F. P., Brunotte, W., Strüber, D., Schneider, K. (2019). Detecting Security Vulnerabilities using Clone Detection and Community Knowledge. *International Conferences on Software Engineering and Knowledge Engineering,* 245–324. doi: https://doi.org/10.18293/seke2019-183
5. Nishi, M. A., Damevski, K. (2018). Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software, 137,* 130–142. doi: https://doi.org/10.1016/j.jss.2017.11.039
6. Kaliuzhna, T., Kubiuk, Y. (2022). Analysis of machine learning methods in the task of searching duplicates in the software code. *Technology Audit and Production Reserves, 4 (2 (66)),* 6–13. doi: https://doi.org/10.15587/2706-5448.2022.263235
7. Singh, M., Sharma, V. (2015). Detection of File Level Clone for High Level Cloning. *Procedia Computer Science, 57,* 915–922. doi: https://doi.org/10.1016/j.procs.2015.07.509
8. Yang, Y., Ren, Z., Chen, X., Jiang, H. (2018). Structural function based code clone detection using a new hybrid technique. *2018 IEEE 42nd annual computer software and applications conference (COMPSAC), 1,* 286–291. doi: https://doi.org/10.1109/compsac.2018.00045
9. *NVD*. Available at: https://nvd.nist.gov/ Last accessed: 22.07.2023
10. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S. et al. (2018). VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *Proceedings 2018 Network and Distributed System Security Symposium*. doi: https://doi.org/10.14722/ndss.2018.23158
11. Chrenousov, A., Savchenko, A., Osadchyi, S., Kubiuk, Y., Kostenko, Y., Likhomanov, D. (2019). Deep learning based automatic software defects detection framework. *Theoretical and Applied Cybersecurity, 1 (1)*. doi: https://doi.org/10.20535/tacs.2664-29132019.1.169086
12. Appel, A. W. (2015). Verification of a Cryptographic Primitive. *ACM Transactions on Programming Languages and Systems, 37 (2),* 1–31. doi: https://doi.org/10.1145/2701415

✉ ***Yevhenii Kubiuk,*** *Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, e-mail: eugen.kubiuk@gmail.com, ORCID: https://orcid.org/0000-0002-7086-0976*

Gennadiy Kyselov, *PhD, Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, ORCID: https://orcid.org/0000-0003-2682-3593*

✉ *Corresponding author*