International Journal of Electrical and Electronics Engineers
Volume 18, Issue No. 01, Jan-June 2026
ISSN (0) 2321-2055 ISSN (P) 2321-2045

AUTOMATED SOURCE CODE DUPLICATION DETECTION USING MACHINE LEARNING TECHNIQUES

Boddu Sivanjali, Deeve Parimala, Shaik Kaif, and Bathini Naresh
Department of ECE,Tirumala Engineering College,Narsaraopet,AP-522426

Abstract— **To develop an efficient and reliable method for detecting code similarities by comparing multiple analysis techniques. This approach can assist educators, developers, and organizations in identifying potential duplication, ensuring code integrity, and fostering ethical coding practices.**

The design and development of a code duplication detection system is crucial for ensuring software quality, safeguarding intellectual property, and promoting ethical coding practices. Code duplication can significantly impact the integrity of academic and professional projects.

Traditional methods of detecting code similarity are time-consuming and prone to errors, as they rely on manual inspection. This project proposes a machine learning-based approach to automatically detect duplicates in source code using publicly available datasets.

Various techniques, including token-based comparison and machine learning classification, are applied to analyze code structure and semantics. To enhance model performance, parameter tuning and scalable system design are incorporated.

I. INTRODUCTION

In today's digital era, software development has become an integral part of almost every field, including education, business, healthcare, and research. With the rapid advancement of technology, programming has gained significant importance, leading to the generation of a massive amount of source code on a daily basis. Students, developers, and organizations continuously create and share code for various applications, resulting in large repositories of software programs. Managing, analyzing, and maintaining such large volumes of code has become a challenging task [3]. One of the major issues associated with large-scale code development is the presence of duplicate or highly similar source code. Source code duplication refers to the occurrence of identical or nearly identical code fragments within one or more programs. This duplication may arise due to several reasons, such as reuse of existing code, following similar programming logic, or rewriting code with minor modifications like changing variable names, formatting, or structure. While limited code reuse can improve development speed, excessive duplication can negatively impact software quality by increasing redundancy, reducing readability, and making maintenance more complex [4]. Detecting duplicate code manually is a time-consuming and error-prone process, especially when dealing with a large number of code files. Traditional approaches for code duplication detection mainly rely on simple text-based comparison techniques. These methods check for exact matches or perform basic string comparisons between code segments. However, such techniques are not effective in identifying logically similar code that has been slightly modified. Even small changes, such as renaming variables, altering comments, or rearranging statements, can prevent traditional systems from recognizing similarity [5]. To address these limitations, more advanced techniques are required that can analyze the structure and patterns within the source code rather than just its textual representation. This is where Machine Learning plays a significant role. Machine Learning techniques enable systems to learn from data and identify patterns automatically without being explicitly programmed for every scenario. By applying machine learning methods to source code analysis, it becomes possible to detect deeper similarities between programs, even when the code is not exactly identical [6].In this project, an automated system is developed to detect source code duplication using Machine Learning techniques. The system works by taking source code files as input and performing a series of processing steps, including data preprocessing, feature extraction, and similarity analysis. During preprocessing, unnecessary elements such as comments and extra spaces are removed, and the code is transformed into a structured format. Feature extraction techniques are then applied to convert the code into numerical representations that can be analyzed by machine learning algorithms [7]. The system utilizes appropriate algorithms to compare different code segments and measure the degree of similarity between them. Based on this analysis, the system can identify whether two or more code files are duplicates or share similar patterns. This automated approach not only improves the efficiency of code analysis but also provides more accurate and reliable results compared to traditional methods. Furthermore, the proposed system is scalable and can handle large datasets, making it suitable for real-world applications such as academic evaluation, software development, and code quality analysis. By reducing manual effort and improving detection accuracy, the system contributes to better management and understanding of source code. In conclusion, the increasing demand for efficient code analysis tools has made source code duplication detection an important area of research. The integration of Machine Learning techniques in this domain provides a powerful solution for identifying similarities in code and overcoming the limitations of traditional approaches. n today's digital era, software development has become an integral part of almost every field. A massive amount of source code is generated daily, making code management a challenging task.

One of the major issues is source code duplication. Duplicate code increases redundancy, reduces readability, and

makes maintenance difficult. In academic environments, it is also linked to plagiarism.

Traditional detection methods rely on simple text comparison, which fails when code is modified. Small changes like variable renaming or formatting can hide duplication.

To overcome this, machine learning techniques are used.This project aims to develop such a system, offering an effective and automated method for detecting duplicate source code.

II. Literature Survey

D. Gitchell and N. Tran (1999) in their work titled "Sim: A utility for detecting similarity in computer programs" developed a tool to detect similarities between source codes using structural comparison techniques, mainly for plagiarism detection. Their approach analyzes program structure rather than exact text matching, making it robust to minor code changes. This work laid the foundation for many modern plagiarism detection tools [1]. B. S. Baker et al. (1999) in "Compressing differences of executable code" proposed a technique to efficiently represent and analyze differences in executable code. Their method focuses on minimizing storage while comparing binary differences. It is useful in software updates and version control systems [2]. Z. Wang et al. (2000) in "BMAT—A binary matching tool for stale profile propagation" developed a tool for comparing binary programs to detect similarities at compiled level. This method works even when source code is unavailable. It is widely useful in reverse engineering and performance optimization [3]. Z. Li et al. (2006) in "CP-miner: Finding copy-paste and related bugs in large-scale software code" proposed a method to detect duplicated code segments and identify bugs caused by copy-paste practices. Their system efficiently scans large codebases to find repeated patterns. It helps improve software quality by reducing redundancy-related errors [4]. A. Ahtiainen et al. (2006) in "Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises" introduced a tool for detecting plagiarism in Java programs by comparing multiple submissions. It supports automatic comparison of student assignments in educational environments. The tool is flexible and can be customized for different academic needs [5]. T. Mikolov et al. (2013) in "Exploiting similarities among languages for machine translation" proposed techniques to leverage similarities across languages in translation models. Their work introduced shared representations for languages. It reduces the need for large parallel datasets [6]. J. Chung et al. (2015) in "Gated feedback recurrent neural networks" proposed improved RNN architectures to enhance sequence learning and feedback mechanisms. Their model improves information flow in deep networks. It helps in better learning of sequential data patterns [7]. M.-T. Luong et al. (2015) in "Multi-task sequence to sequence learning" proposed multi-task learning approaches to improve sequence modeling performance. Their model shares knowledge across tasks. This improves generalization and efficiency [8]. J. Mueller and A. Thyagarajan (2016) in "Siamese recurrent architectures for learning sentence similarity" proposed Siamese neural networks for measuring similarity between sequences. Their model uses twin networks to compare input pairs effectively. It is widely applied in NLP and code similarity detection tasks [9]. H. Palangi et al. (2016) in "Deep sentence embedding using LSTM networks" used LSTM-based embeddings to capture semantic meaning for similarity tasks. Their approach improves understanding of long-term dependencies in sequences. It enhances performance in information retrieval and similarity analysis [10]. Q. Feng et al. (2016) in "Scalable graph-based bug search for firmware images" proposed graph-based methods for detecting bugs in firmware using scalable techniques. Their approach models program structure as graphs. It improves bug detection in embedded systems [11]. X. Xu et al. (2017) in "Neural network-based graph embedding for cross-platform binary similarity detection" used graph embeddings to compare binaries across platforms. The model captures structural and semantic features. It improves accuracy in cross-platform comparisons [12]. A. Vaswani et al. (2017) in "Attention is all you need" introduced the Transformer model using attention mechanisms, revolutionizing sequence modeling. It eliminates the need for recurrent networks. This model significantly improves performance in NLP tasks [13]. M. Johnson et al. (2017) in "Google's multilingual neural machine translation system" introduced a multilingual NMT system capable of zero-shot translation. It can translate between languages without direct training data. This improves scalability of translation systems [14]. M. Artetxe et al. (2017) in "Unsupervised neural machine translation" proposed translation models that do not require parallel corpora, improving multilingual learning. Their approach relies on monolingual data. It reduces dependency on labeled datasets [15]. F. Zuo et al. (2018) in "Neural machine translation inspired binary code similarity comparison" proposed using NMT-inspired models for comparing binary code beyond function pairs. Their model treats binary instructions like language sequences. This improves similarity detection across different architectures [16]. J. Gao et al. (2018) in "VulSeeker: A semantic learning-based vulnerability seeker" introduced a model to detect vulnerabilities in binaries using semantic learning. It identifies security flaws based on code behavior. This helps in improving software security analysis [17]. K. Redmond et al. (2018) in "A cross-architecture instruction embedding model" introduced instruction embedding techniques inspired by NLP for binary analysis. It converts instructions into vector representations. This enables effective comparison across architectures [18]. J. Devlin et al. (2018) in "BERT: Pre-training of deep bidirectional transformers" proposed a transformer-based model that improves contextual understanding in NLP tasks. It uses bidirectional context for better feature extraction. BERT is widely used in similarity and classification tasks [19]. K. Seki (2019) in "On cross-lingual text similarity using neural translation models" explored similarity detection across different languages using NMT. The method translates texts into a common representa-

tion space. This enables effective comparison of multilingual documents [20].

III. OBJECTIVES

To develop an automated system for detecting duplication and similarity in source code and to preprocess source code by removing unnecessary elements such as comments and formatting differences. • To convert source code into a structured format suitable for analysis using feature extraction technique and to apply Machine Learning algorithms to identify patterns and measure similarity between code files. • To improve the accuracy of duplicate code detection compared to traditional text-based methods, which can also reduce manual effort involved in analyzing large numbers of code files. • To build a scalable system capable of handling large datasets efficiently and provides reliable results that help in improving code quality and maintainability.

IV. PROPOSED METHODOLOGY

A. Start

The process begins with the initialization of the system. At this stage, all required modules, libraries, and configurations are loaded into the system environment. This includes loading the machine learning model, setting up preprocessing functions, and preparing the interface for user interaction. Proper initialization ensures that the system runs smoothly without errors during execution [24].

B. UserAuthentication

In this step, the user is required to log into the system using valid credentials such as username and password. This ensures that only authorized users can access the system and perform duplication detection tasks. Authentication enhances the security of the system and prevents unauthorized usage. It also helps in maintaining user-specific data and tracking activity if required.

C. Upload Code Snippets

Once the user is authenticated, they are allowed to upload the source code files or manually enter code snippets into the system. These code snippets serve as the input data for analysis. The system supports different formats of source code and ensures that the uploaded files are properly stored and prepared for further processing. This step is crucial as the accuracy of the system depends on the quality of input data.

D. Code Preprocessing

The uploaded code undergoes preprocessing to remove unnecessary variations and improve consistency. This step includes several sub-processes: • Removal of Comments: All comments in the code are removed since they do not contribute to logic and may affect similarity analysis. • Normalization: The code is standardized by converting it into a uniform format. This may include converting text to lowercase, removing extra spaces, and formatting code consistently. • Tokenization: The code is broken down into

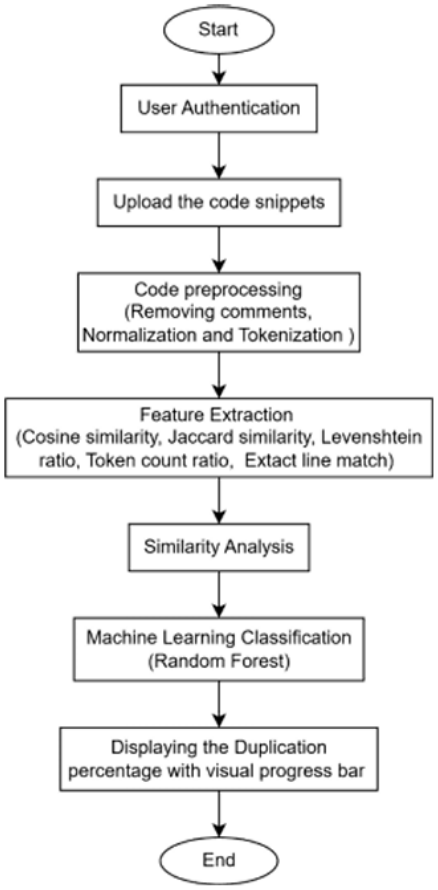

Fig. 1. Flowchart of Automated Source Code Duplication Detection System

smaller elements called tokens, such as keywords, operators, and identifiers. This helps in structured analysis of the code. Preprocessing ensures that irrelevant differences are eliminated and only meaningful content is considered for similarity detection [24].

E. Feature Extraction

In this stage, important features are extracted from the preprocessed code using various similarity techniques. These features represent different aspects of similarity between code snippets:
(a) Cosine Similarity Measures similarity between numerical feature vectors derived from token frequency representation. It captures structural resemblance in vector space.
(b) Jaccard Similarity Measures overlap between sets of tokens. It is effective in identifying common
token usage patterns. (c) Levenshtein Ratio Calculates edit distance between code snippets. It measures how many insertions, deletions, or substitutions are required to transform one code into another.
(d) Token Count Ratio Compares the proportion of tokens between two code files. It identifies structural size similarity.
(e) Exact Line Match Detects identical lines between code snippets, identifying direct copy-paste instances. Each sim-

ilarity metric generates a numerical score. These scores collectively form a structured feature vector representing the relationship between the compared code samples.

F. Similarity Analysis

After extracting individual similarity scores, the system performs combined similarity analysis. In this stage: • All similarity metrics are aggregated. • Patterns across different similarity measures are evaluated. • A structured feature vector is created.This stage ensures that duplication detection does not rely on a single metric but considers multiple perspectives of similarity. Similarity analysis improves robustness and reduces dependency on isolated measures [25].

G. Machine Learning Classification (Random Forest)

The feature vector generated in the previous stage is passed to the trained Random Forest classifier. Random Forest can: • Handles multiple numerical features efficiently • Reduces overfitting using ensemble learning • Provides high classification accuracy • Works well with structured datasets The model has been trained on labelled data. During prediction: • The classifier evaluates similarity patterns • It assigns a probability score The decision is based on learned similarity behaviour rather than fixed thresholds [26].

H. Displaying the Duplication Percentage with Visual Progress Bar

Finally, the system presents the output to the user in a clear and understandable format. The duplication level is displayed as a percentage, indicating how similar the code snippets are. In addition, a visual progress bar is used to represent the similarity level graphically. This visual representation enhances user experience and makes it easier to interpret the results quickly. The system ensures that the output is accurate, user- friendly, and informative.

I. End

The workflow terminates after displaying results. The system is then ready to process new submissions.

V. RESULT AND DISCUSSION

The system is capable of detecting both exact and approximate code duplication. It performs well even when the code is modified through:
• Changes in variable names
• Differences in formatting
• Minor structural variations
The use of multiple similarity techniques ensures that the system captures different aspects of code similarity. For example, while exact line matching identifies identical code segments, techniques like cosine similarity and Levenshtein ratio detect deeper logical similarities. The Random Forest classifier further enhances the accuracy of the results by combining multiple feature inputs and making informed predictions. This reduces the chances of incorrect classification and improves the reliability of the system.

```
Confusion Matrix:
[[300   9]
 [  6 377]]
Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.97      0.98       309
           1       0.98      0.98      0.98       383

    accuracy                           0.98       692
   macro avg       0.98      0.98      0.98       692
weighted avg       0.98      0.98      0.98       692
```

Fig. 2. Terminal Output Showing Perfomance Metrics and Confusion Matrix

1) System Behavior: The system exhibits stable and efficient behavior across different inputs. Key observations include:
• Consistency: The system produces consistent results for similar inputs, ensuring reliability.
• Efficiency: The processing time is reasonable, even when analyzing multiple code snippets.
• Scalability: The system can handle larger datasets with minor adjustments, making it suitable for real-world applications.
• Robustness: The system is capable of handling variations in code without significant loss in accuracy.

The performance of the system indicates that the proposed methodology is effective in detecting source code duplication. The combination of preprocessing, feature extraction, and machine learning techniques ensures high accuracy and reliability. The system successfully overcomes many limitations of traditional approaches and provides a practical solution for code similarity detection in academic and professional environments.

VI. CONCLUSION

In the present digital era, the rapid increase in programming activities has made it difficult to manually identify duplicate or similar source code. This project addresses this challenge by developing an automated system for source code duplication detection using Machine Learning techniques. The proposed system follows a structured approach that includes preprocessing, feature extraction, similarity analysis, and machine learning-based classification. During preprocessing, unnecessary elements such as comments and formatting differences are removed to ensure accurate analysis. The system then extracts multiple similarity features using techniques like cosine similarity, Jaccard similarity, Levenshtein ratio, token ratio, and exact line matching, which together provide a comprehensive understanding of code similarity. A key advantage of this system is the use of the Random Forest classifier, which improves accuracy by analyzing multiple features and reducing overfitting. This enables the system to detect both exact duplication and modified code with changes in variable names, formatting, or structure. The results show that the proposed system

performs better than traditional methods in terms of accuracy, precision, recall, and F1-score. The output is presented as a duplication percentage along with a visual representation, making it easy for users to interpret the results. Overall, the project successfully develops an efficient, scalable, and reliable solution for source code duplication detection. It can be effectively applied in academic institutions, coding platforms, and software development environments to improve code quality and promote ethical programming practices.

VII. Future scope

Although the proposed system provides accurate and reliable results, there are several opportunities for further improvement and extension to enhance its functionality and real-world applicability. One of the primary enhancements is to extend the system to support cross programming languages such as Java coding is evaluated with C coding, which will increase its usability across different development environments. Additionally, integrating advanced deep learning techniques such as neural networks, transformers, and code embedding models can help in capturing deeper semantic relationships in source code, enabling the detection of complex and logically similar code patterns even when the implementation differs significantly.The system can also be improved by incorporating structural analysis techniques such as Abstract Syntax Trees (AST) and graph-based representations, which provide a deeper understanding of code structure and logic. This will allow the system to detect similarity at a semantic level rather than relying only on textual and token-based analysis.

References

[1] D. Gitchell and N. Tran, “Sim: A utility for detecting similarity in computer programs,” ACM SIGCSE Bull., vol. 31, no. 1, pp. 266–270, 1999.

[2] B. S. Baker, U. Manber, and R. Muth, “Compressing differences of executable code,” in Proc. ACMSIGPLAN Workshop Compiler Support Syst. Softw. (WCSS), Apr. 1999, pp. 1–10.

[3] Z. Wang, K. Pierce, and S. McFarling, “BMAT—A binary matching tool for stale profile propagation,” J. Instruct.-Level Parallelism, vol. 2, pp. 1–20, May 2000.

[4] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-miner: Finding copy-paste and related bugs in large-scale software code,” IEEE Trans. Softw. Eng., vol. 32, no. 3, pp. 176– 192, Mar. 2006.

[5] A. Ahtiainen, S. Surakka, and M. Rahikainen, “Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises,” in Proc. 6th Baltic Sea Conf. Comput. Educ. Res., Koli Calling, Feb. 2006, pp. 141–142.

[6] T. Mikolov, Q. V. Le, and I. Sutskever, “Exploiting similarities among languages for machine translation,” 2013, arXiv:1309.4168.

[7] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Gated feedback recurrent neural networks,” in Proc. Int. Conf. Mach. Learn., Jun. 2015, pp. 2067–2075.

[8] M.-T. Luong, Q. V. Le, I. Sutskever, O. Vinyals, and L. Kaiser, “Multi-task sequence to sequence learning,” 2015, arXiv:1511.06114.

[9] J. Mueller and A. Thyagarajan, “Siamese recurrent architectures for learning sentence similarity,” in Proc. AAAI Conf. Artif. Intell., Mar. 2016, vol. 30, no. 1, pp. 1- 7

[10] H. Palangi, L. Deng, Y. Shen, J. Gao, X. He, J. Chen, X. Song, and R. Ward, “Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval,” IEEE/ACM Trans. Audio, Speech, Language Process., vol. 24, no. 4, pp. 694–707, Apr. 2016.

[11] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, “Scalable graphbased bug search for firmware images,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., Oct. 2016, pp. 480–491.

[12] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, “Neural network-based graph embedding for cross-platform binary code similarity detection,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., Oct. 2017, pp. 363–376.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in Proc. Adv. Neural Inf. Process. Syst., vol. 30, 2017, pp. 1–11.

[14] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, G. Corrado, M. Hughes, and J. Dean, “Google’s multilingual neural machine translation system: Enabling zeroshot translation,” Trans. Assoc. Comput. Linguistics, vol. 5, pp. 339–351, Oct. 2017.

[15] M. Artetxe, G. Labaka, E. Agirre, and K. Cho, “Unsupervised neural machine translation,” 2017, arXiv:1710.11041.

[16] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, “Neural machine translation inspired binary code similarity comparison beyond function pairs,” 2018, arXiv:1808.04706.

[17] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, “VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary,” in Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), Sep. 2018, pp. 896–899.

[18] K. Redmond, L. Luo, and Q. Zeng, “A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis,” 2018, arXiv:1812.09652.

[19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” 2018, arXiv:1810.04805.

[20] K. Seki, “On cross-lingual text similarity using neural translation models,” J. Inf. Process., vol. 27, pp. 315–321, 2019, doi: 10.2197/ipsjjip.27.315.

[21] K. Seki, “On cross-lingual text similarity using neural translation models,” J. Inf. Process., vol. 27, pp. 315–321, 2019, doi: 10.2197/ipsjjip.27.315.

[22] S. H. H. Ding, B. C. M. Fung, and P. Charland, “Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization,” in Proc. IEEE Symp. Secur. Privacy (SP), May 2019, pp. 472–489.

[23] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, “Investigating graph embedding neural networks with unsupervised features extraction for binary analysis,” in Proc. Workshop Binary Anal. Res., 2019, pp. 1–11.

[24] X. Zhang, W. Sun, J. Pang, F. Liu, and Z. Ma, “Similarity metric method for binary basic blocks of cross-instruction set architecture,” in Proc. Workshop Binary Anal. Res., vol. 10, 2020, pp. 1–12.

[25] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, “Order matters: Semanticaware neural networks for binary code similarity detection,” in Proc. AAAI Conf. Artif. Intell., Apr. 2020, vol. 34, no. 1, pp. 1145–1152.

[26] Ministry of Internal Affairs and Communications. 2021 Information and Communications White Paper. Accessed: Oct. 3, 2021. [Online]. Available: https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/r03/html/nd105220.html

[27] K. Seki, “Cross-lingual text similarity exploiting neural machine translation models,” J. Inf. Sci., vol. 47, no. 3, pp. 404–418, Jun. 2021.

[28] Y. Masubuchi, M. Hashimoto, and A. Otsuka, “SIBYL: A method for detecting similar binary functions using machine learning,” IEICE Trans. Inf. Syst., vol. E105.D, no. 4, pp. 755–765, 2022.

[29] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, “jTrans: Jump-aware transformer for binary code similarity detection,” in Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal., Jul. 2022, pp. 1–13

[30] N. Ito, M. Hashimoto and A. Otsuka, "Feature Extraction Methods for Binary Code Similarity Detection Using Neural Machine Translation Models," in IEEE Access, vol. 11, pp. 102796-102805.