A Study on Binary Code Similarity Detection

Bastien Schoonaert[1], Hyun-Jun Kim[1], Yun-Heung Paek[1]
[1]Dept. of Electrical and Computer Engineering and Inter-University Semiconductor Research Center (ISRC), Seoul National University
bastien.schoonaert@snu.ac.kr, hjkim@sor.snu.ac.kr, ypaek@snu.ac.kr

A Study on Binary Code Similarity Detection

바스티안 [1], 김현준 [1], 백윤흥 [1]
[1]서울대학교 전기정보공학부, 반도체 공동연구소

Abstract

Binary Code Similarity Detection (BCSD) plays a critical role in software security applications like vulnerability detection and malware analysis. This review surveys both traditional and machine-learning-based approaches to BCSD. Traditional methods, such as control flow graph matching and symbolic execution, have demonstrated effectiveness but suffer from scalability issues, particularly with obfuscated code. Modern machine learning techniques, including graph neural networks and deep learning models, offer improved adaptability across architectures and scalability. Despite these advancements, challenges remain in cross-platform detection, handling obfuscation, and deploying BCSD tools in real-world security scenarios. The review highlights recent innovations and outlines potential future directions for enhancing the robustness and efficiency of BCSD systems.

1. Introduction

Binary Code Similarity Detection (BCSD) has emerged as a crucial technique in several fields, such as vulnerability detection, malware analysis, and intellectual property protection. As software systems grow in complexity and the diversity of architectures increases, identifying similarities between binary code across different platforms becomes more challenging. Traditional methods of BCSD primarily focused on structural and syntactic analyses, leveraging control flow graphs (CFGs) and symbolic execution to detect similarities. However, these approaches often struggle with scalability, especially when dealing with complex or obfuscated code. In recent years, machine learning-based methods have gained traction due to their ability to learn patterns in binary code and adapt across various architectures. As BCSD continues to evolve, it becomes increasingly important to understand the strengths and limitations of all these approaches. This study provides a comprehensive review of existing BCSD methods, exploring both traditional and machine-learning-driven techniques, with an emphasis on deep learning-based techniques. The study also looks at common evaluation methods and outlines future directions to address the remaining challenges in the field.

2. Traditional BCSD approaches

Traditional Binary Code Similarity Detection (BCSD) techniques largely rely on structural and syntactic analysis of binary programs. A predominant approach is graph-based matching [1, 2, 3, 4], where binary programs' control Flow Graphs (CFGs) are extracted and compared to identify similarities. In these methods, each function in a binary is represented as a graph, with nodes corresponding to basic blocks of code and edges representing control flow between them. Graph isomorphism algorithms are then applied to match similar structures, identifying commonalities between different binaries or across different compilation settings.

Another approach is symbolic execution and theorem proving [1, 2]. These techniques aim to evaluate binary behavior by exploring possible execution paths in a program. Symbolic execution treats program inputs as symbolic variables, enabling analysis across multiple execution scenarios without concrete input values. By assessing how a binary operates under various conditions, symbolic execution identifies functional similarities between binaries. However, it suffers from the state explosion problem, where the number of potential execution paths increases exponentially, making the technique less scalable for large binaries or those with complex control flows.

Furthermore, basic block fingerprinting and signature-based methods [3] have been employed to quickly compare binaries by generating unique fingerprints for each block or function. These methods provide faster comparisons, but they are susceptible to code obfuscation and compiler optimizations, which can change the binary's structure without altering its functionality. Minor changes in instruction order or register usage can lead to a mismatch, making these methods less robust in real-world scenarios

involving obfuscated code or different compilation settings.

3. Machine Learning Approaches in BCSD

As traditional BCSD methods encountered scalability and efficiency challenges, particularly with complex binaries and cross-architecture detection [1, 4], machine learning (ML) approaches have emerged as a powerful alternative. These methods leverage the ability of ML models to automatically learn patterns in binary code, leading to more adaptable and efficient BCSD systems [3, 5, 6, 7, 8, 9, 10].

One prominent machine learning technique is graph neural networks (GNNs). GNNs are particularly effective for analyzing binary code as they can capture structural information from control flow graphs (CFGs) and data flow graphs. In the BEDetector system [3], a graph autoencoder (GAE) is employed to extract global structural features from CFGs, surpassing the limitations of traditional CFG-based matching methods by learning the latent structural features of the graph, while also incorporating semantic analysis inspired by natural language processing techniques.

Another important direction in machine learning for BCSD is embedding-based techniques, where the goal is to convert assembly instructions or intermediate representations of binaries into numerical embeddings that can be compared efficiently. For example, Asm2Vec [6] converts assembly code or intermediate code into vector representations, which are then used to measure the similarity between functions or blocks of code.

Recently, the use of deep learning models has gained the most attraction, particularly recurrent neural networks and Long Short-Term Memory (LSTM) networks. In [5], an LSTM network generates feature embeddings from binary files by learning from sequences of local features extracted from CFGs. These embeddings are then compared using a Siamese Neural Network, which is designed to compute the similarity between two binary files by learning semantic similarities between their embeddings.

4. Deep Learning for BCSD

While machine learning techniques have improved the scalability and efficiency of BCSD, the introduction of deep learning has significantly enhanced the field's ability to detect similarities for more complex binaries. One of the most significant challenges in BCSD is the ability to detect similarities across different instruction set architectures (ISAs) and platforms. Deep learning models, particularly those leveraging transfer learning and pre-trained embeddings, offer a robust solution to the challenge of identifying binary similarities between code compiled for different ISAs [7, 8, 9, 11].

Indeed, deep learning offers a more sophisticated approach to capturing both structural and semantic patterns in binary code. Unlike traditional machine learning models, deep learning systems can automatically extract features without the need for manual feature engineering, allowing them to adapt more effectively to cross-architecture similarity detection. One of the key advantages of deep learning in BCSD is its ability to generate rich embeddings of binary code that capture complex patterns, making it more resilient to changes introduced by different compilation settings, optimization levels, or instruction sets. Embeddings allow for the conversion of binary instructions or intermediate representations into dense vectors, which can be compared efficiently across architectures. In the case of BERT-inspired models, embeddings are not only architecture-agnostic but also capture deep semantic relationships between instructions.

BERT-inspired models leverage transfer learning and pre-trained embeddings, allowing for effective cross-platform similarity detection without requiring retraining for each new architecture.

The BinShot [7] system is a prominent example of using a BERT-based architecture to detect binary code similarities across different ISAs. BinShot employs a transferable similarity learning approach, where the model is pre-trained on a large corpus of assembly code, enabling it to generalize to new binaries compiled for different architectures. By leveraging BERT's ability to understand the contextual relationships between instructions, the model is capable of learning higher-level semantic patterns that persist across different platforms.

In addition to BinShot, jTrans [8] is another deep learning-based system that utilizes neural machine translation models to map binaries between different architectures. This model is trained on aligned binary pairs, learning to translate code from one architecture to another while preserving its semantic meaning. By combining the power of BERT-like pre-training with neural translation, jTrans demonstrates the potential of deep learning in bridging the gap between architectures and enabling robust cross-platform detection.

Similarly, Trex [9] introduces a cross-architecture binary code analysis method that focuses on instruction embeddings that generalize across architectures, improving the scalability of binary analysis for security and vulnerability detection.

5. Applications of BCSD

BCSD has a range of applications across various domains due to its ability to identify similar code fragments in binary

programs. BCSD can detect plagiarized code segments, even in cases where binaries have been obfuscated or compiled differently. This is particularly useful in software copyright enforcement, ensuring that reused code is properly attributed. BCSD also helps in recognizing similarities between known malware binaries and new or unknown samples. By identifying code similarities, it assists in detecting malware variants that reuse portions of malicious code, enabling faster classification and response.

However, one of the most critical applications of BCSD is vulnerability detection. BCSD enables the discovery of security flaws in binary programs by identifying similarities between known vulnerable code segments and other binaries. With the rapid growth of software, manual vulnerability detection is inefficient, and BCSD has emerged as a powerful solution to automate this process across various platforms and architectures, especially in the context of code reuse. Techniques for vulnerability detection using BCSD employ both traditional and machine learning approaches, along with specialized tools designed to detect known and novel vulnerabilities [1, 3, 4, 12].

Graph-based methods, such as VulHawk [12], use CFGs to match structural patterns of vulnerable code, while machine learning models are trained to identify patterns in vulnerable code segments by learning from a large dataset of known vulnerabilities. BEDetector [3] utilizes GAEs to combine semantic and structural features for accurate detection. These methods improve the efficiency of detecting vulnerable code clones or reused vulnerabilities across different systems.

Another promising technique is binary function similarity detection for vulnerabilities, where the focus is on detecting vulnerable code clones across versions or architectures. Tools like [1] and [4] take advantage of code similarity to find vulnerabilities introduced through code reuse or software updates. These tools employ embedding methods to detect similar functions that share vulnerability signatures, helping developers catch security issues before they propagate across different systems.

6. Evaluation Metrics and Benchmarking

Evaluating the effectiveness of Binary Code Similarity Detection (BCSD) systems is crucial for understanding their accuracy, scalability, and overall performance. Researchers typically rely on a set of well-established metrics and benchmarking datasets to assess the quality of BCSD methods. These metrics help compare different approaches, whether they are graph-based, machine-learning-driven, or hybrid methods. Common evaluation metrics include precision, recall, F1-score, and accuracy, which provide insight into how well a BCSD system identifies true positives while minimizing false positives and negatives [1, 3, 5].

Precision is a metric used to measure the proportion of correctly identified binary similarities (true positives) out of all detected similarities. It is particularly important in BCSD systems designed for vulnerability detection, where false positives can lead to wasted time and effort in investigating benign code. Recall, or sensitivity, measures the system's ability to correctly identify all relevant binary similarities, emphasizing its ability to detect true positives even when they are sparse. High recall is critical in tasks like malware detection or code clone analysis, where missing any instance of similarity could have serious security consequences. The F1 score balances both precision and recall, providing a single metric that captures the trade-offs between these two. It is often used to evaluate BCSD systems that need to balance the detection of true positives while avoiding too many false positives. Lastly, time and computational cost are important metrics when assessing BCSD tools, especially when dealing with large datasets.

When evaluating BCSD tools, researchers employ a variety of benchmarking datasets to ensure consistent comparison across systems. These datasets typically include real-world binaries, malware samples, and firmware binaries. For instance, BEDetector [3] was tested on real-world firmware files, where it successfully identified vulnerabilities in commonly used libraries and systems. As for vulnerability detection tools, the Juliet test suite and CVEfixes dataset, which contain known flaws and vulnerabilities, are often used to measure how effective these tools are at finding real-world vulnerabilities.

7. Future Directions

As Binary Code Similarity Detection (BCSD) continues to evolve, several promising directions could improve its effectiveness. Future research in BCSD aims to address the remaining challenges related to cross-platform detection, scalability, handling obfuscation, and enhancing the integration of machine learning models [3, 11, 12]. Current systems, while effective, still face challenges when dealing with binaries compiled for different architectures or optimized using different compilers. Handling obfuscation and code transformations is another significant challenge. Many BCSD systems struggle to detect similarities in binaries that have been deliberately obfuscated to hide malicious intent or proprietary code. Lastly, integration with real-world security tools and frameworks is a key future direction. While many BCSD techniques are developed in research settings, their deployment still seems limited. For example, there is potential for BCSD tools to be more tightly

integrated with existing security platforms, such as malware detection systems. A promising area of synergy is the combination of BCSD with directed fuzzing. BCSD can help identify vulnerable or high-risk code segments by detecting similarities to known vulnerabilities. Directed fuzzing can then focus on these specific areas to trigger potential security flaws and confirm the presence of vulnerabilities. At the same time, it provides a way to reproduce them, which makes it easier for developers to patch these vulnerabilities.

8. Conclusion

Binary Code Similarity Detection (BCSD) has advanced from traditional graph-based and symbolic methods to modern machine learning-driven approaches. While machine learning models have introduced significant improvements, challenges remain. Future work should focus on enhancing cross-architecture capabilities, improving the interpretability of machine learning models, and integrating BCSD with real-world security tools.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2023-00277326, 0.1), the Institute of Information & communications Technology Planning & Evaluation (IITP) under the artificial intelligence semiconductor support program to nurture the best talents grant funded by the Korean government (MSIT) (IITP-2023-RS-2023-00256081), the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone, 0.5), the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2024, Inter-University Semiconductor Research Center (ISRC).

References

[1] X. Zhang, W. Sun, J. Pang, F. Liu, and Z. Ma. 2020. "Similarity Metric Method for Binary Basic Blocks of Cross-Instruction Set Architecture". Proceedings 2020 Workshop on Binary Analysis Research.

[2] P. Keller, A. K. Kaboré, L. Plein, J. Klein, Y. Le Traon, and T. F. Bissyandé. 2021. "What You See is What it Means! Semantic Representation Learning of Code based on Visualization and Transfer Learning". ACM Transactions on Software Engineering and Methodology, 31, 2, Article 31 (April 2022).

[3] L. Yu, Y. Lu, Y. Shen, H. Huang, and K. Zhu. 2021. "BEDetector: A Two-Channel Encoding Method to Detect Vulnerabilities Based on Binary Similarity". In IEEE Access, vol. 9, 51631-51645.

[4] D. Kim, E. Kim, S. Cha, S. Son, and Y. Kim. 2023. "Revisiting Binary Code Similarity Analysis Using Interpretable Feature Engineering and Lessons Learned". IEEE Transactions on Software Engineering, 49, 4 (April 2023), 1661–1682.

[5] Z. Luo, T. Hou, X. Zhou, H. Zeng, and Z. Lu. 2021. "Binary Code Similarity Detection through LSTM and Siamese Neural Network". EAI Endorsed Transactions on Security and Safety, vol. 8, no. 29, p. e1, Sep. 2021.

[6] S. H. H. Ding, B. C. M. Fung and P. Charland. 2019. "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization". 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 472-489.

[7] S. Ahn, S. Ahn, H. Koo, and Y. Paek. 2022. "Practical Binary Code Similarity Detection with BERT-based Transferable Similarity Learning". In Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC '22). Association for Computing Machinery, New York, NY, USA, 361–374.

[8] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang. 2022. "JTrans: jump-aware transformer for binary code similarity detection". In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022). Association for Computing Machinery, New York, NY, USA, 1–13.

[9] K. Pei, Z. Xuan, S. Jana, and B. Ray. 2020. "Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity".

[10] X. Xu, S. Feng, Y. Ye, G. Shen, Z. Su, S. Cheng, G. Tao, Q. Shi, Z. Zhang, and X. Zhang. 2023. "Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis". In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 1106–1118.

[11] H. Wang, Z. Gao, C. Zhang, M. Sun, Y. Zhou, H. Qiu, and X. Xiao. 2024. "CEBin: A Cost-Effective Framework for Large-Scale Binary Code Similarity Detection". In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 149–161.

[12] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu. 2023. "VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search". In Network and Distributed System Security (NDSS) Symposium 2023, San Diego, CA, USA.