electronics

MDPI

Article

A Survey of Binary Code Similarity Detection Techniques

Liting Ruan [1], Qizhen Xu [1,*], Shunzhi Zhu [1], Xujing Huang [1] and Xinyang Lin [2]

[1] College of Computer and Information Engineering, Xiamen University of Technology, Xiamen 361024, China; ltruan@stu.xmut.edu.cn (L.R.); szzhu@xmut.edu.cn (S.Z.); xujing.huang@xmut.edu.cn (X.H.)
[2] Xiamen Zhonglian Century Corporation, Xiamen 361024, China; linxy@zhonglian.com
* Correspondence: qzxu@xmut.edu.cn

Abstract: Binary Code Similarity Detection is a method that involves comparing two or more binary code segments to identify their similarities and differences. This technique plays a crucial role in areas such as software security, vulnerability detection, and software composition analysis. With the extensive use of binary code in software development and system optimization, binary code similarity detection has become an important area of research. Traditional methods of source code similarity detection face challenges when dealing with the unreadable and complex nature of binary code, necessitating specialized techniques and algorithms. This review compares and summarizes various techniques and methods of binary code similarity detection, highlighting their strengths and limitations in handling different characteristics of binary code. Additionally, the article suggests potential future research directions. As research and innovation in this technology continue to advance, binary code similarity detection is expected to play an increasingly significant role in fields like software security.

Keywords: binary code analysis; deep learning; software security; similarity detection

Citation: Ruan, L.; Xu, Q.; Zhu, S.; Huang, X.; Lin, X. A Survey of Binary Code Similarity Detection Techniques. *Electronics* **2024**, *13*, 1715. https://doi.org/10.3390/electronics13091715

Academic Editor: Hung-Yu Chien

Received: 16 March 2024
Revised: 22 April 2024
Accepted: 25 April 2024
Published: 29 April 2024

1. Introduction

With the expansion of software scale and complexity and the vast and varied codebases, it is necessary to analyze, compare, and identify similarities within them, thereby achieving greater accomplishments in the fields of software security. Binary code similarity detection techniques compare two or more binary code fragments to find their similarities. These similarities can be measured based on multiple aspects, such as the structure, syntax, semantics, or behavior of the code.

Binary code similarity detection has a wide range of applications, such as bug hunting [1–16], malware detection [17–19], patch generation [20], cross-version information porting [21,22], software composition analysis [23], and vulnerability detection [24]. However, determining binary code similarity is challenging. Firstly, the characteristics of binary code make it more difficult to read and understand compared to source code. Secondly, a significant amount of program semantic information, such as function names, variable names, and data structures, is lost during the compilation process. Finally, the generated binary code can undergo significant changes when using different compilers, changing compiler optimization options, various optimization options, or when targeting different operating systems and CPU architectures.

In summary, this paper makes the following contributions:

- This paper identifies and discusses the challenges faced by traditional source code similarity detection methods when applied to binary code, such as code obfuscation, constant software updates, patching, and differences in coding styles across projects.
- This paper provides a comprehensive summary of current research progress in binary code feature extraction methods, including static analysis, dynamic analysis, hybrid analysis, and deep learning techniques.

- This paper suggests the development of new analysis strategies and technologies to handle challenges like function inlining, obfuscated code, patch presence, and differences in coding styles across projects. It emphasizes the importance of improving the accuracy and efficiency of binary code similarity analysis through advanced techniques.

The rest of the paper is organized as follows: Section 2 outlines the basic principles of binary code. Section 3 discusses the challenges encountered in this domain. Section 4 introduces methods of analysis, including static, dynamic, and hybrid approaches that combine both static and dynamic analysis, as well as learning-based approaches. Section 5 explores the potential impact of recent research in this field and proposes possible future research directions. Through in-depth study, we aim to gain a better understanding of the core issues in binary code similarity detection and provide effective support for the advancement of this field.

2. The Basic Concepts of Binary Code Similarity Detection

Binary Code Similarity Detection (BCSD) refers to the technique of comparing and analyzing semantic differences in binary files or executable programs. This technique aims to compare the semantic differences between two binary programs, rather than merely relying on byte-level comparisons. BCSD attempts to understand the meaning and behavior of programs, which helps in identifying code reuse, plagiarism, software piracy, malware variants, or even assessing the impact of software patches and updates, thereby playing a role in fields such as software development, reverse engineering, and security analysis.

However, binary code similarity analysis is fundamentally different from source code similarity analysis. In binary code similarity analysis, the only input source may be an executable binary file or program, which significantly limits the amount of information available for analysis. The binary code generation process encompasses multiple stages, involving the conversion from source code to the final executable file, a process influenced by factors such as the compiler, optimization options, and target platform, as shown in Figure 1 [24]. The specific steps and their potential impacts on generating a binary file are as follows:

1. Source Code Input: The initial step involves inputting the source code, which may be written in C, C++, or another programming language.
2. Compiler Selection: A suitable compiler is selected based on compilation requirements, such as GCC, Clang, VC, etc. For example, the compiler can inline functions (replace function calls with the function body itself), eliminate dead code (unused code), reorder instructions or merge loops, etc.
3. Optimization Options Configuration: Optimization levels (e.g., O0, O1, O2, O3, Os) are chosen based on requirements, affecting the performance and size of the compiled product. Selecting varying optimization levels directly impacts the performance and size of the compiled outputs. For instance, advanced optimization settings like O3 can enhance execution efficiency, yet they might also lead to an increase in the size of the generated binaries. Conversely, optimizations designated by Os focus on minimizing the file size, which could potentially compromise execution performance to some extent.
4. Target Platform Specification: The target platform for the compilation is specified, including the operating system (such as macOS, Linux, Windows), CPU architecture (such as x86, ARM, MIPS), and word size (32-bit or 64-bit). Cross-version binaries may not only alter the syntax of a program but can also change its semantics, presenting an additional significant challenge for binary similarity analysis.
5. Compilation Process: Using the selected compiler and settings, the source code is compiled into assembly code, generating target files for the specified platform. However, obfuscation techniques may also be applied during compilation, further increasing the complexity and uncertainty of the code.
6. Linking Process: The compiled target files and all dependencies, such as library files, are linked together to produce the final executable file. Modern software frequently

relies on dynamic link libraries (DLLs). Changes in these libraries may result in variations in the binary representation of the same code, thereby impacting the accuracy of binary similarity detection.

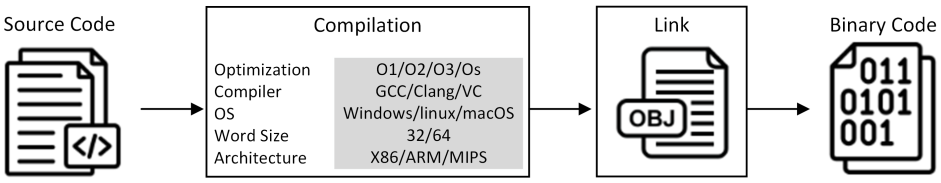

Figure 1. The compilation process.

The source code goes through a series of complex processing steps before it can be converted into a binary program that can be directly executed by a machine. The choices and settings within the entire compilation chain directly impact the performance, functionality, and compatibility of the final product. During the entire compilation process, differences in the choice of compiler, optimization configurations, target platform, CPU architecture, and bitness can result in significantly varied binary code from the same source code. These factors collectively make cross-platform binary code similarity analysis a highly challenging task. Each step in the process may introduce changes that cause the final binary to be structurally and functionally different from its original source code. Although the aforementioned improvements contribute to enhancing program performance, they also escalate the complexity involved in analyzing the similarity of binary code.

3. Key Technical Challenges

In this section, we summarized the following key technical challenges on binary code similarity detection.

3.1. Information Loss

It is evident that in the process of compiling source code into binary code, due to optimizations, structural reorganizations, and the removal of redundant information performed by compilers, the resulting binary code is likely to lose a significant amount of crucial information from the source code [24]. For example, binary code typically does not contain human-readable identifiers and explanatory information such as function names, variable names, and comments, which increases the difficulty of understanding and analyzing the code.

Moreover, when binary code lacks debugging information, even developers with a deep understanding of the programming language face significant challenges. Debugging information provides vital details about the program's structure, variables, functions, etc., which are essential for developers to understand the internal working principles and logical architecture of the program. The absence of debugging information means that analysts cannot easily track the execution process of the code or identify connections between different code segments, making the analysis and debugging of binary code more difficult and complex.

In summary, given the characteristics of binary code and the lack of crucial information, analyzing and understanding binary code indeed poses greater challenges and complexities compared to source code.

3.2. The Impact of Instruction Set Architecture

In the x86 architecture, the instruction set consists of a series of specific instructions used for executing particular computations and operations. The x86 system also has unique registers and memory access patterns, such as the use of stack pointers. In contrast, the ARM architecture significantly differs from x86 in terms of its instruction set, register set, and memory access patterns. Likewise, the MIPS architecture possesses its distinct instruction set, register layout, and memory access mechanisms.

The variations in architectures imply that the same source code may yield completely different binary code representations when compiled under different systems. This diversity in code manifestations introduces complexity in comparing binary files across platforms. Therefore, it is essential to meticulously evaluate the discrepancies in instruction sets, register configurations, and memory access processes across various architectures to guarantee both the precision and reliability of such comparisons.

3.3. The Impact of Compilers

Significant differences exist between control flow graphs generated by different compilation configurations, including compilers, optimization options, and target platforms. These differences are primarily determined by the optimization strategies of compilers and the characteristics of target platforms. Different compilers can apply distinct optimization treatments to the same source code, resulting in variations in the structure of the generated machine code. Additionally, modifications in optimization settings can significantly influence code generation. Specific optimization preferences might induce changes or enhancements in control flow, thus modifying the structure of the final control flow graph.

Moreover, the unique architecture and instruction set of different target platforms necessitate compilers to generate specific machine code tailored to each. For example, a compiler might optimize for the characteristics of the x86-64 architecture, while for the MIPS architecture, it would generate machine code suited to that system.

3.4. The Impact of Obfuscation

The impact of obfuscation techniques is profound as they can significantly alter the original Control Flow Graph (CFG) of a program, thereby increasing the difficulty of performing binary code similarity analysis [25]. Techniques such as function-level obfuscation (-fla), instruction substitution (-sub), and control flow flattening (-bcf) are employed to enhance the complexity of analyzing and understanding binary code, offering a degree of protection against reverse-engineering efforts. These obfuscation methods not only increase the complexity of the code but also significantly raise the challenge of analyzing and understanding the program's structure.

This presents a considerable obstacle for binary similarity analysis. Traditional binary similarity matching techniques, which often rely on static analysis, depend on the structural features of CFGs to identify similarities between code fragments. However, obfuscation techniques, by altering the structure of CFGs, can cause even binaries derived from the same source code to differ substantially due to obfuscation, making it challenging for conventional matching techniques to recognize them as similar.

Therefore, in the face of obfuscation techniques, the field of binary similarity detection necessitates more advanced analytical methods. These could include, but are not limited to, machine learning technologies capable of learning deeper features from obfuscated code, or dynamic analysis methods that analyze the runtime behavior of programs directly. Such approaches can circumvent the effects of static code obfuscation, enabling a more accurate match for similarity. This advancement in analytical techniques underscores the ongoing challenge and response between the development of obfuscation methods and the efforts to effectively detect and analyze obfuscated binary code.

3.5. The Impact of Dynamic Link Libraries and External Dependencies

In binary similarity detection, managing Dynamic Link Libraries (DLLs) and external dependencies introduces significant technical challenges due to the uncertainty of external library versions, dynamic loading mechanisms that complicate static analysis, and variations in DLL versions that impact execution and dependencies. Additionally, dependencies on specific operating systems, hardware configurations, or environmental conditions can influence program behavior, affecting the reliability of detection methods.

Therefore, to tackle these challenges, approaches such as dynamic analysis and advanced deep learning techniques are essential. These strategies help in accurately identify-

ing and comparing binary files amidst the complexities introduced by DLLs and external dependencies, highlighting the need for sophisticated technologies in effective binary similarity detection.

3.6. The Impact of Function Inlining

Inline functions enhance program execution efficiency by replacing function calls with the actual code of the function, thus eliminating the overhead of function calls. This is particularly effective for frequently called small functions. However, inline functions do not differ significantly in code structure from regular functions, making their identification highly challenging. At the assembly level, instructions of inline functions are often intermingled with surrounding code rather than being isolated in a separate code block like regular functions, which complicates the precise identification of the start and end points of inline functions at the assembly level.

Furthermore, the challenge of recognition increases when the instructions of inline functions are not sequentially arranged due to instruction alignment and pipeline optimization. This is because compiler optimizations can cause the code to be rearranged and reorganized to leverage the processor's pipeline and cache features, resulting in the instructions of inline functions being potentially non-contiguous in memory. Additionally, within binaries, there are no established expert patterns, such as prologue/epilogue instructions, for inline functions [26].

Therefore, accurately identifying inline functions requires in-depth static code analysis and a profound understanding of the assembly level. By analyzing the context around function call sites and conducting a detailed review of the assembly code, it is possible to explore which functions are suitable for inlining and how to optimize inline code to enhance performance.

4. Methods of Binary Code Similarity Detection

This article categorizes the current works on binary code similarity detection based on the methodologies for extracting binary code features, including static analysis-based approaches, dynamic analysis-based approaches, hybrid analysis-based approaches, and learning-based analysis approaches, as shown in Table 1.

4.1. Static Analysis-Based Approaches

Static analysis methods extract structural information by disassembling binary code and analyzing control flow graphs for similarity measurement. This approach facilitates an in-depth analysis based on the code's structure, syntax, and semantics without executing the program. The algorithms involve graph matching and analysis of static code fragments, making them well suited for post-compilation code analysis. While static analysis exhibits a degree of robustness in handling code transformations and obfuscations, most methods predominantly focus on the syntactic aspects of instructions rather than their semantics. Furthermore, graph matching algorithms not only demand significant computational efforts but also struggle to guarantee the optimality of matches, leading to limited accuracy. Currently, a significant portion of research on binary code similarity employs static analysis techniques.

Table 1. Classification of binary code similarity analysis method.

Category	Method Type	Method	Accuracy	Efficiency	Recall	Precision	Security	Scalability	Resource Consumption
Static Analysis	Graph-based Matching	Multi-MH [3], discovRE [4], Genius [6]	High	High	Low	High	Low	Low	Low
	Static Slicing	Esh [5], GitZ [10], Zeek [27], Xmatch [9], BinGo [7], FirmUP [12]							
Dynamic Analysis	Runtime Behavior	IMF [28], MockingBird [29], CoP [30]	High	Low	High	High	High	Low	Low
	Dynamic Program Slicing	BinSim [31]							
Hybrid Analysis	Simulation	BinGo-E [32], CACompare [33], BinMatch [34], Patchecko [35]	High	Modern	High	High	Modern	Low	Low
Machine Learning	Statistical Feature Learning	Gemini [11], αDiff [14], VulSeeker [15], IoTSeeker [36], VulSeeker-Pro [37], BiN [38], FIT [39], TikNib [40]	High	High	High	High	Low	High	High
	Automatic Feature Learning	Asm2Vec [41], InneyEye [42], SAFE [43], MIRROR [44], Instr [45], OrderMatters [46], Trex [47], PalmTree [48], DeepBinDiff [49], COMBO [50], jTrans [51], kTrans [52]							

4.1.1. Graph-Based Comparison Methods

This method involves converting binary code into a graphical representation and then applying various comparison algorithms (such as edit distance, longest common subsequence, etc.) to assess similarity, as shown in Figure 2. Techniques in this category include Multi-MH [3], discovRE [4], and Genius [6], et al. The initial graph-based comparison method was introduced by researchers at Zynamics in 2006 with BinDiff [53], which matches functions by performing a series of graph isomorphism checks on call graphs and utilizes control flow graphs (CFGs) for matching basic blocks. Building on these studies, Binslayer introduced the Hungarian algorithm in 2013 to optimize the graph matching process, thereby enhancing the precision of the matching results [54].

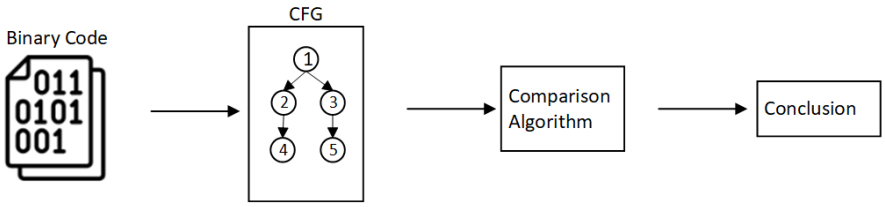

Figure 2. Flowchart of graph-based comparison methods.

4.1.2. Static Slicing (Strands)

This method involves decomposing a graph into smaller, comparable components for subsequent comparison, as shown in Figure 3. Examples of this method include Esh [5], GitZ [10], Zeek [27], Xmatch [9], BinGo [7], and FirmUP [12], et al. In 2014, David Y and Yahav E introduced a tracelet-based code search technique that transforms control flow graphs (CFGs) into a series of fixed-length paths [1]. These fixed-length paths, known as tracelets [1], capture the control flow characteristics of a program, making them available for further analysis, matching, or comparison. Subsequently, these tracelets are matched using rewriting techniques.

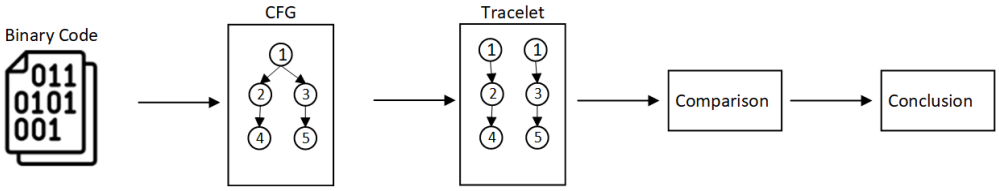

Figure 3. Flowchart of static slicing (strands).

4.2. Dynamic Analysis Methods

Dynamic analysis methods assess the performance and functionality of software applications by executing and monitoring their behavior. This approach primarily involves conducting tests on the program in real or simulated environments using a specific set of input cases to collect information on its input–output and behavioral data, including the program's input–output, system calls, memory access patterns, and exceptional events. Dynamic analysis exhibits a higher tolerance for code transformation and obfuscation but requires a balance between security and performance costs.

Early studies, such as Blanket Execution [55], execute two versions of binary functions with the same inputs and calculate similarity based on the differences in their behavior. The assessment of binary code similarity involves comparing outputs and state changes during the function execution. BinHunt [56] and iBinHunt [57] utilize symbolic execution and theorem proving techniques to verify the equivalence of different basic blocks or strands. Symbolic execution, which substitutes concrete variables with symbolic representations and performs symbolic operations to explore program execution paths, aids in detecting equivalence between codes. Dynamic analysis is further subdivided into runtime behavior analysis and dynamic program slicing techniques.

4.2.1. Runtime Behavior Analysis

Techniques such as IMF [28], Patchecko [35], MockingBird [29], and CoP [30] all employ runtime behavior analysis methods. IMF [28] and Patchecko [35] utilize fuzz testing to evaluate the similarity of binary code, a technique that injects random or semi-random data into programs to trigger potential vulnerabilities and anomalies, thereby revealing functional and behavioral differences between codes. MockingBird [29] applies dynamic instrumentation, inserting monitors or trackers during program execution to obtain detailed information about program behavior. This method is capable of logging function calls, memory access, and system calls, providing a basis for assessing binary code similarity. Meanwhile, CoP [30] employs symbolic execution to collect program behavior paths, symbolically representing program paths and using symbolic reasoning to explore all potential execution paths, effectively identifying behavioral differences and similarities between codes.

4.2.2. Dynamic Program Slicing Techniques

BinSim [31] utilizes system calls for dynamic program slicing and employs symbolic execution to assess program equivalence. Dynamic slicing techniques extract relevant code fragments from the execution path of a program, facilitating the comparison of functionalities and behaviors across different program versions.

These methods employ a variety of technical strategies to collect and analyze program behavior data, with the goal of evaluating similarities and functional differences between binary codes. This process is crucial for revealing functional discrepancies, identifying potential vulnerabilities, and detecting security risks.

4.3. Hybrid Analysis-Based Approaches

Hybrid analysis methods integrate dynamic and static analysis, aiming to comprehensively consider code coverage, detection accuracy, and the scalability of the methods. These methods strive to compensate for the respective limitations of static and dynamic analysis to enhance the precision and robustness of detection. In the field of binary code similarity detection, techniques such as BinGo-E [32], CACompare [33], and BinMatch [34] adopt emulation strategies. After completing static analysis, they emulate the execution of target functions to extract semantic features for similarity comparison. Throughout this process, it is possible to capture the function execution paths, system call sequences, and other key behavioral characteristics, allowing for a more comprehensive and precise assessment of similarity between functions.

Particularly, Patchecko [35] employs a more complex hybrid analysis strategy. In the static detection phase, candidate functions are selected through static analysis; then, during the dynamic analysis phase, the execution trajectories of functions are captured using runtime DLL injection and remote debugging technologies. With the results of these dynamic analyses, Patchecko [35] can quantify the similarity between functions and validate the accuracy of static analysis.

In summary, hybrid analysis methods combine the advantages of dynamic and static analysis, improving the accuracy and comprehensiveness of code similarity detection. By supplementing static analysis with emulation or dynamic analysis, these methods more effectively understand and compare the behavioral and structural differences between binary codes, providing a more reliable solution for software security and vulnerability identification.

4.4. Learning-Based Analysis Approaches

Since the beginning of the 21st century, with the rapid development of technology and the economy, fields such as computer science have seen swift expansion, with emerging technologies like artificial intelligence and the metaverse gaining prominence. Deep learning, in particular, has become a hot topic of research. To date, numerous research teams and companies have made significant contributions in the area of deep learning. The emergence of various network models has not only offered new perspectives for problem-solving but

has also brought convenience. Although attempts to apply machine learning algorithms in early binary code similarity analysis work, such as Zeek [27], were made, these applications were generally limited to serving as classifiers or filters and did not become the core technology. Since the introduction of the Gemini [11] model in 2017, deep learning algorithms based on graph embedding, natural language processing, and self-attention networks have been widely applied in the field of binary code similarity analysis, as shown in Figure 4. These methods can be divided into those based on statistical feature learning and those based on automated feature learning. Compared to traditional static and dynamic analysis methods, learning-based approaches offer the following advantages:

1. Higher accuracy. By integrating various aspects of code, including syntactic, semantic, and structural features, the precision of the analysis is enhanced.
2. Better scalability. Learning-based methods are more flexible than complex graph matching algorithms or dynamic execution techniques, and the learning process can be accelerated with GPUs, significantly increasing efficiency.

Figure 4. Flowchart of learning-based analysis approaches.

Therefore, these approaches have attracted widespread attention within the industry and have become a major trend in recent research.

4.4.1. Learning Based on Statistical Features

The learning strategy based on statistical features primarily focuses on learning statistical information of nodes (i.e., basic blocks) within a graph and integrating this information as node attributes into the graph, thus forming what is known as an Attributed Control Flow Graph (ACFG). Subsequently, ACFGs are processed through graph neural networks to extract the graph embedding vectors of functions, serving as representations of the functions. Research projects such as Gemini [11], VulSeeker-Pro [37], α Diff [14], VulSeeker [15], IoTSeeker [36], BiN [38], and FIT [39] have adopted methods similar to structural Word2vec and Siamese networks for supervised learning to handle ACFGs and their derivatives.

4.4.2. Learning Based on Automated Feature Learning

In recent years, with the rapid advancement of machine learning and deep learning technologies, a succession of network models has been introduced. To avoid biases that may arise from manual feature selection, researchers have shifted towards automated feature learning methods for extracting semantic features from target code automatically. Studies such as Asm2Vec [41], InnerEye [42], Instr [45], SAFE [43], MIRROR [44], OrderMatters [46], Trex, DeepBinDiff, jTrans [51], and kTrans [52] are dedicated to the automatic identification of the code's semantic features. In 2016, the Genius [6] model introduced by Feng utilized spectral clustering techniques to group multiple ACFGs after their construction. Furthermore, researchers have transformed ACFGs into vector representations using popular encoding techniques (such as word2vec, CNNs, etc.), with the Gemini [11] model proposed in 2017 employing the graph embedding network Structure2Vec for vectorization. Subsequently, researchers have begun applying advanced natural language processing technologies to the field of binary code similarity detection. The core idea is to automatically learn feature vectors representing the semantics and semantic relations from the raw bytes or assembly instructions of binary code, such as Asm2Vec [41] using the PV-DM model to learn function embeddings, and SAFE automating function embedding learning with self-attention networks. Recent studies have shown that the Transformer model, as an efficient language representation tool, performs excellently in understanding the semantics of binary code and supporting downstream tasks, like jTrans [51] and kTrans [52]. Features obtained through automated

learning methods are independent of manual reverse engineering and are not limited by the skill level of manual reversing. This approach avoids biases that manual reversing might introduce, more effectively capturing the target code's characteristics, thereby achieving higher accuracy in the final stage of the similarity comparison. In practical application scenarios, a quintessential example is the development of BinaryAI, spurred by a paper published by Tencent's Keen Security Lab in 2020 [46]. This platform represents a cutting-edge binary security intelligence analysis tool. BinaryAI signifies a significant leap from function similarity analysis to component similarity analysis, offering capabilities for Software Composition Analysis (SCA). It can precisely and efficiently identify third-party components and their version numbers within binary files. Moreover, its support for function-level matching and customizable comparison ranges provides unparalleled flexibility in dealing with complex binary files. Currently, they have successfully upgraded BinaryAI to its third generation. Compared to its predecessors, this approach significantly enhances the accuracy of matching binary source code.

5. Future Trends and Research Directions

Currently, the field of binary code similarity detection is in a phase of rapid development. Despite facing numerous challenges, it also harbors significant opportunities. This paper outlines several potential open research directions in this area.

5.1. The Application of Deep Learning

Since the beginning of the 21st century, deep learning has emerged as a hot research area, with its application in binary code similarity detection gradually expanding. Especially, recent studies have shown that Transformer [58] models are capable of understanding the semantics of binary code, aiding in various downstream tasks. The use of Transformer-based binary code embedding techniques, leveraging their powerful self-attention mechanisms, effectively captures long-distance dependencies and complex patterns within the code. Examples of such approaches include PalmTree [48], jTrans [51], COMBO [50], and kTrans [51], which typically follow a pre-training and fine-tuning paradigm. They learn general binary code representations from a vast amount of unlabeled data and then fine-tune these representations for specific code analysis tasks.

Future research directions in deep learning for binary code similarity detection will primarily focus on the following areas.

5.1.1. Exploring Larger Domain Models

Although the Transformer model is favored for its ability to capture long-distance dependencies, high scalability, and capacity for transfer learning, it also presents some non-negligible drawbacks, including its high complexity, the need for manual integration of prior knowledge, and limitations in fully understanding instructions. Therefore, in the task of binary code similarity detection, there is still room for improvement in the current Transformer models. According to research on the scaling laws of Transformers by Jordan Hoffmann and others [59], it is known that as the size of the model and the volume of data increase, the scaling effect of the model becomes more significant. Hence, by further advancing pre-training techniques, it is possible to construct more powerful models specifically designed for binary code similarity detection tasks.

Such more powerful models have the potential to capture more complex patterns and dependencies within binary code, thereby significantly enhancing the accuracy of binary code similarity detection. By leveraging large datasets for pre-training, these models can learn deeper representations of binary code, offering new approaches and methods to address the challenges of binary code similarity detection.

5.1.2. Exploring More Cost-Effective Models

It is well-known that training and running large-scale language models often require a significant amount of computational resources. Therefore, exploring cost-effective binary

code embedding techniques has become a key research direction. In this context, there are mainly two potential approaches: (1) Investigating transfer learning strategies, where models pretrained on large general datasets are fine-tuned on smaller, binary code-specific datasets. This approach leverages knowledge learned from a broader domain, allowing the model to achieve better performance with lower computational resource consumption. (2) Exploring efficient model training techniques, such as the teacher–student paradigm. This method involves training a smaller, simpler model (the student model) to learn from a larger, more complex model (the teacher model), thereby obtaining better performance for the student model with reduced computational resources.

By adopting these cost-effective training methods, researchers and practitioners can more easily develop and deploy binary code embedding models, thereby enhancing performance and efficiency in tasks such as binary code similarity detection.

5.2. Enhancing the Accuracy of Reverse Engineering

In current research on binary code similarity analysis, widely used methods, including but not limited to Genius [6], Gemini [11], and jTrans [51], commonly rely on mature commercial disassembly tools like IDA Pro. Although IDA Pro is widely recognized as a leading tool for binary reverse engineering, it inevitably has certain limitations and shortcomings [60], mainly involving the following aspects:

5.2.1. For Entry Point and Function Boundary Identification

Most disassemblers rely on symbol tables to determine function boundaries and construct control flow graphs. However, in certain cases, particularly when symbol tables are inaccurate or missing, locating function boundaries becomes especially challenging. This issue is pronounced in the analysis of binary firmware, where entry points and base addresses are often unknown. Additionally, there may be functions with multiple entry points, necessitating further identification efforts. This is especially true for programs that are complex in design or utilize special compilation techniques; the lack of clear symbol information can lead to misjudgments and omissions during the disassembly process, thereby increasing the complexity and difficulty of binary code analysis. Therefore, developing techniques and methods that can effectively identify function boundaries and entry points without symbol table information are crucial for enhancing the accuracy and efficiency of binary code reverse engineering.

5.2.2. Code Obfuscation/Transformation

To protect their creations, both legitimate/benign program authors and malicious software developers may employ various technical measures, such as code obfuscation, encryption, or packaging. These measures are taken for several reasons; for example, legitimate software authors might aim to protect intellectual property or prevent their software from being illegally copied, modified, or republished as malware. On the other hand, malicious software developers use obfuscation techniques to evade detection and analysis by security analysis tools or researchers. These obfuscation and encryption techniques, serving different purposes, significantly increase the complexity and challenge of binary code analysis. Currently, only specific obfuscation techniques, like Obfuscator-LLVM, take the impact of code obfuscation into account. This implies that the vast majority of existing analysis tools and techniques may have significant limitations in dealing with advanced obfuscation techniques, especially the emerging and more complex methods of obfuscation.

Therefore, the development of advanced binary code analysis methods capable of effectively identifying and handling various obfuscation techniques is particularly important. This not only requires a deep understanding of the essence and trends of obfuscation techniques but also the development of new analysis algorithms and techniques to enhance the ability to decipher obfuscated code, thereby allowing for a more accurate analysis and understanding of the true functionality and behavior of binary programs.

5.3. Enhancing the Ability to Handle Different Configurations

While current binary code similarity analysis techniques have made remarkable progress in handling cross-architecture, cross-compiler, and various compilation optimization options, research in this field still faces a series of unresolved challenges. These challenges include, but are not limited to, function inlining, code obfuscation, software patch handling, and cross-project analysis.

Firstly, modern compilers' function inlining optimization strategies significantly increase the complexity of similarity analysis, as code fragments with similar functionalities may be embedded within different functions during the compilation process, presenting additional difficulties in accurately identifying similar code.

Secondly, the use of code obfuscation techniques by both malicious software developers and legitimate software authors also complicates analysis efforts. Obfuscation aims to make the code difficult to understand, thereby increasing the challenge of recognizing similarities.

Moreover, the constant updating and patching of software necessitate that similarity analysis techniques are capable of adapting to software version iterations and identifying changes introduced by patches. The cross-project analysis presents another significant challenge, as different projects often utilize diverse coding styles and structures, requiring similarity analysis techniques to have sufficient flexibility and adaptability.

To address these challenges, it is imperative to develop new analysis strategies and technologies that enhance the ability to handle function inlining, obfuscated code, patch presence, and differences in coding styles across projects, thereby improving the accuracy and efficiency of binary code similarity analysis.

6. Conclusions

This paper comprehensively reviews the fundamental principles of binary code similarity detection, addressing challenges such as information loss and multi-platform adaptability. It summarizes the current progress in research on binary code feature extraction, covering methods such as static analysis, dynamic analysis, and hybrid analysis, as well as techniques that incorporate deep learning. Further, it outlines future research directions, emphasizing the need for a deeper understanding of binary code semantics, capturing more profound representations of binary code, enhancing the accuracy of reverse engineering, overcoming existing technological limitations, and significantly improving the efficiency and accuracy of binary code similarity detection. Key initiatives include strengthening the identification of complex code obfuscation techniques, optimizing cross-version and cross-platform code analysis strategies, and employing more complex deep learning models to capture more intricate code features. By integrating and optimizing these strategies, binary code similarity analysis technology is expected to play a more critical role in key areas such as software security assessment, malicious code detection, and software composition analysis, thereby advancing the development of this field.

Author Contributions: Funding acquisition, Q.X. and S.Z.; Investigation L.R. and Q.X.; Project administration Q.X.; Supervision Q.X.; Visualization X.H.; Writing—original draft preparation, L.R.; writing—review and editing, Q.X., S.Z., X.H. and X.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Xiamen City Science and Technology Development Project under grant No. 3502Z20231042, the Xiamen City Natural Science Foundation under grant No. 3502Z202372046, Xiamen Institute of Technology High level Talent Program under grant No. YKJ22042R, and Fujian Provincial Department of Education Young and Middle aged Teacher Education Research Project under grant No. JAT232016.

Data Availability Statement: All data underlying the results are available as part of the article and no additional source data are required.

Conflicts of Interest: Xinyang Lin is employed by Xiamen Zhonglian Century Corporation. The remaining authors declare no conflicts of interest.

References

1. David, Y.; Yahav, E. Tracelet-based code search in executables. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, UK, 9–11 June 2014; PLDI '14; Association for Computing Machinery: New York, NY, USA, 2014; pp. 349–360. [CrossRef]
2. Pewny, J.; Schuster, F.; Bernhard, L.; Holz, T.; Rossow, C. Leveraging semantic signatures for bug search in binary programs. In Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC '14, New Orleans, LA, USA, 8–12 December 2014; Association for Computing Machinery: New York, NY, USA, 2014; pp. 406–415. [CrossRef]
3. Pewny, J.; Garmany, B.; Gawlik, R.; Rossow, C.; Holz, T. Cross-architecture bug search in binary executables. *IT Inf. Technol.* **2017**, *59*, 83. [CrossRef]
4. Eschweiler, S.; Yakdan, K.; Gerhards-Padilla, E. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In Proceedings of the 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, CA, USA, 21–24 February 2016; The Internet Society: Reston, VA, USA, 2016.
5. David, Y.; Partush, N.; Yahav, E. Statistical similarity of binaries. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, Santa Barbara, CA, USA, 13–17 June 2016; PLDI '16; Association for Computing Machinery: New York, NY, USA, 2016; pp. 266–280. [CrossRef]
6. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H. Scalable Graph-based Bug Search for Firmware Images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; Association for Computing Machinery: New York, NY, USA, 2016. [CrossRef]
7. Chandramohan, M.; Xue, Y.; Xu, Z.; Liu, Y.; Cho, C.Y.; Tan, H.B.K. BinGo: Cross-architecture cross-OS binary search. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, 13–18 November 2016; Association for Computing Machinery: New York, NY, USA, 2016; pp. 678–689. [CrossRef]
8. Huang, H.; Youssef, A.M.; Debbabi, M. BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS '17, Abu Dhabi, United Arab Emirates, 2–6 April 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 155–166. [CrossRef]
9. Feng, Q.; Wang, M.; Zhang, M.; Zhou, R.; Henderson, A.; Yin, H. Extracting Conditional Formulas for Cross-Platform Bug Search. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS '17, Abu Dhabi, United Arab Emirates, 2–6 April 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 346–359. [CrossRef]
10. David, Y.; Partush, N.; Yahav, E. Similarity of binaries through re-optimization. *SIGPLAN Not.* **2017**, *52*, 79–94. [CrossRef]
11. Xu, X.; Liu, C.; Feng, Q.; Yin, H.; Song, L.; Song, D. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, 30 October–3 November 2017; Thuraisingham, B., Evans, D., Malkin, T., Xu, D., Eds.; ACM: New York, NY, USA, 2017; pp. 363–376. [CrossRef]
12. David, Y.; Partush, N.; Yahav, E. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. *SIGPLAN Not.* **2018**, *53*, 392–404. [CrossRef]
13. Shirani, P.; Collard, L.; Agba, B.L.; Lebel, B.; Debbabi, M.; Wang, L.; Hanna, A. BINARM: Scalable and Efficient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Devices. In Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment—15th International Conference, DIMVA 2018, Saclay, France, 28–29 June 2018; Proceedings; Lecture Notes in Computer Science; Giuffrida, C., Bardin, S., Blanc, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2018; Volume 10885, pp. 114–138. [CrossRef]
14. Liu, B.; Huo, W.; Zhang, C.; Li, W.; Li, F.; Piao, A.; Zou, W. αDiff: Cross-version binary code similarity detection with DNN. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18, Montpellier, France, 3–7 September 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 667–678. [CrossRef]
15. Gao, J.; Yang, X.; Fu, Y.; Jiang, Y.; Sun, J. VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18, Montpellier, France, 3–7 September 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 896–899. [CrossRef]
16. Xu, Y.; Xu, Z.; Chen, B.; Song, F.; Liu, Y.; Liu, T. Patch based vulnerability matching for binary programs. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, Virtual Event, USA, 18–22 July 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 376–387. [CrossRef]
17. Krügel, C.; Kirda, E.; Mutz, D.; Robertson, W.K.; Vigna, G. Polymorphic Worm Detection Using Structural Information of Executables. In Proceedings of the Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005, Seattle, WA, USA, 7–9 September 2005; Revised Papers; Lecture Notes in Computer Science; Valdes, A., Zamboni, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3858, pp. 207–226. [CrossRef]
18. Bruschi, D.; Martignoni, L.; Monga, M. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *Detection of Intrusions and Malware & Vulnerability Assessment, Proceedings of the Third International Conference, DIMVA 2006, Berlin, Germany, 13–14 July 2006*; Proceedings; Lecture Notes in Computer Science; Büschkes, R., Laskov, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; Volume 4064, pp. 129–143. [CrossRef]
19. Bruschi, D.; Martignoni, L.; Monga, M. Code Normalization for Self-Mutating Malware. *IEEE Secur. Priv.* **2007**, *5*, 46–54. [CrossRef]

20. Baker, B.S.; Muth, R. Compressing Differences of Executable Code. In *ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS)*; Citeseer: Princeton, NJ, USA, 2012.
21. Wang, Z.; Pierce, K.; McFarling, S. BMAT—A Binary Matching Tool for Stale Profile Propagation. *J. Instr. Level Parallelism* **2000**, *2*, 1–20.
22. Flake, H. Structural Comparison of Executable Objects. In Proceedings of the Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop, DIMVA 2004, Dortmund, Germany, 6–7 July 2004; Proceedings; Flegel, U., Meier, M., Eds.; Gesellschaft für Informatik e.V.: Bonn, Germany, 2004; Volume P-46, pp. 161–173.
23. Jiang, L.; An, J.; Huang, H.; Tang, Q.; Nie, S.; Wu, S.; Zhang, Y. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. *arXiv* **2024**. [CrossRef]
24. Haq, I.U.; Caballero, J. A Survey of Binary Code Similarity. *ACM Comput. Surv.* **2022**, *54*, 51:1–51:38. [CrossRef]
25. Yu, Y.; Gan, S.; Qiu, J.; Qin, X.; Chen, Z. Research on binary code similarity analysis technology and its application in embedded device firmware vulnerability search. *J. Softw.* **2021**, *32*. 4137–4172. [CrossRef]
26. Lin, W.; Guo, Q.; Yin, J.; Zuo, X.; Wang, R.; Gong, X. FSmell: Recognizing Inline Function in Binary Code. In *Computer Security—ESORICS 2023, Proceedings of the 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, 25–29 September 2023*; Tsudik, G., Conti, M., Liang, K., Smaragdakis, G., Eds.; Springer Nature: Cham, Switzerland, 2024; pp. 487–506.
27. Shalev, N.; Partush, N. Binary Similarity Detection Using Machine Learning. In Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS '18, Toronto, ON, Canada, 15–19 October 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 42–47. [CrossRef]
28. Wang, S.; Wu, D. In-memory fuzzing for binary code similarity analysis. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 30 October–3 November 2017; pp. 319–330. [CrossRef]
29. Hu, Y.; Zhang, Y.; Li, J.; Gu, D. Cross-Architecture Binary Semantics Understanding via Similar Code Comparison. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 14–18 March 2016; Volume 1, pp. 57–67. [CrossRef]
30. Luo, L.; Ming, J.; Wu, D.; Liu, P.; Zhu, S. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Trans. Softw. Eng.* **2017**, *43*, 1157–1177. [CrossRef]
31. Ming, J.; Xu, D.; Jiang, Y.; Wu, D. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In Proceedings of the USENIX Security Symposium, Vancouver, BC, Canada, 16–18 August 2017.
32. Xue, Y.; Xu, Z.; Chandramohan, M.; Liu, Y. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation. *IEEE Trans. Softw. Eng.* **2019**, *45*, 1125–1149. [CrossRef]
33. Hu, Y.; Zhang, Y.; Li, J.; Gu, D. Binary Code Clone Detection across Architectures and Compiling Configurations. In Proceedings of the 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), Buenos Aires, Argentina, 22–23 May 2017; pp. 88–98. [CrossRef]
34. Hu, Y.; Zhang, Y.; Li, J.; Wang, H.; Li, B.; Gu, D. BinMatch: A Semantics-Based Hybrid Approach on Binary Code Clone Analysis. In Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 23–29 September 2018; pp. 104–114. [CrossRef]
35. Sun, P.; Garcia, L.; Salles-Loustau, G.; Zonouz, S.A. Hybrid Firmware Analysis for Known Mobile and IoT Security Vulnerabilities. In Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29–2 July 2020; pp. 373–384. [CrossRef]
36. Gao, J.; Yang, X.; Jiang, Y.; Song, H.; Choo, K.K.R.; Sun, J. Semantic Learning Based Cross-Platform Binary Vulnerability Search For IoT Devices. *IEEE Trans. Ind. Inform.* **2021**, *17*, 971–979. [CrossRef]
37. Gao, J.; Yang, X.; Fu, Y.; Jiang, Y.; Shi, H.; Sun, J. VulSeeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, Lake Buena Vista, FL, USA, 4–9 November 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 803–808. [CrossRef]
38. Wu, H.; Shu, H.; Kang, F.; Xiong, X. BiN: A Two-Level Learning-Based Bug Search for Cross-Architecture Binary. *IEEE Access* **2019**, *7*, 169548–169564. [CrossRef]
39. Liang, H.; Xie, Z.; Chen, Y.; Ning, H.; Wang, J. FIT: Inspect vulnerabilities in cross-architecture firmware by deep learning and bipartite matching. *Comput. Secur.* **2020**, *99*, 102032. [CrossRef]
40. Kim, D.; Kim, E.; Cha, S.K.; Son, S.; Kim, Y. Revisiting Binary Code Similarity Analysis Using Interpretable Feature Engineering and Lessons Learned. *IEEE Trans. Softw. Eng.* **2023**, *49*, 1661–1682. [CrossRef]
41. Ding, S.H.H.; Fung, B.C.M.; Charland, P. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 472–489. [CrossRef]
42. Zuo, F.; Li, X.; Young, P.; Luo, L.; Zeng, Q.; Zhang, Z. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, CA, USA, 24–27 February 2019; The Internet Society: Reston, VA, USA, 2019.

43. Massarelli, L.; Luna, G.A.D.; Petroni, F.; Baldoni, R.; Querzoni, L. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment—16th International Conference, DIMVA 2019, Gothenburg, Sweden, 19–20 June 2019; Proceedings; Lecture Notes in Computer Science; Perdisci, R., Maurice, C., Giacinto, G., Almgren, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11543, pp. 309–329. [CrossRef]
44. Zhang, X.; Sun, W.; Pang, J.; Liu, F.; Ma, Z. Similarity metric method for binary basic blocks of cross-instruction set architecture. In Proceedings of the 2020 Workshop on Binary Analysis Research, San Diego, CA, USA, 23 February 2020; Volume 10.
45. Redmond, K.; Luo, L.; Zeng, Q. A Cross-Architecture Instruction Embedding Model for Natural Language Processing-Inspired Binary Code Analysis. *arXiv* **2018**, arXiv:1812.09652.
46. Yu, Z.; Cao, R.; Tang, Q.; Nie, S.; Huang, J.; Wu, S. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. In Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, 7–12 February 2020; AAAI Press: Palo Alto, CA, USA, 2020; pp. 1145–1152. [CrossRef]
47. Pei, K.; Xuan, Z.; Yang, J.; Jana, S.; Ray, B. Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity. *arXiv* **2020**, arXiv:2012.08680.
48. Li, X.; Qu, Y.; Yin, H. Palmtree: Learning an assembly language model for instruction embedding. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, 15–19 November 2021; pp. 3236–3251.
49. Duan, Y.; Li, X.; Wang, J.; Yin, H. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In Proceedings of the 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, CA, USA, 23–26 February 2020; The Internet Society: Reston, VA, USA, 2020.
50. Zhang, Y.; Huang, C.; Zhang, Y.; Cao, K.; Andersen, S.T.; Shao, H.; Leach, K.; Huang, Y. Pre-Training Representations of Binary Code Using Contrastive Learning. *arXiv* **2022**, arXiv:2210.05102.
51. Wang, H.; Qu, W.; Katz, G.; Zhu, W.; Gao, Z.; Qiu, H.; Zhuge, J.; Zhang, C. jTrans: Jump-aware transformer for binary code similarity detection. In Proceedings of the ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, 18–22 July 2022; Ryu, S., Smaragdakis, Y., Eds.; ACM: New York, NY, USA, 2022; pp. 1–13. [CrossRef]
52. Zhu, W.; Wang, H.; Zhou, Y.; Wang, J.; Sha, Z.; Gao, Z.; Zhang, C. kTrans: Knowledge-Aware Transformer for Binary Code Embedding. *arXiv* **2023**. [CrossRef]
53. Xu, L. Security patch comparison techniques based on graph isomorphism theory. *J. Comput. Appl.* **2006**, *26*, 1623.
54. Kuhn, H.W. The Hungarian Method for the Assignment Problem. In *50 Years of Integer Programming 1958-2008—From the Early Years to the State-of-the-Art*; Jünger, M., Liebling, T.M., Naddef, D., Nemhauser, G.L., Pulleyblank, W.R., Reinelt, G., Rinaldi, G., Wolsey, L.A., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 29–47. [CrossRef]
55. Egele, M.; Woo, M.; Chapman, P.; Brumley, D. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, 20–22 August 2014; Fu, K., Jung, J., Eds.; USENIX Association: Berkeley, CA, USA, 2014; pp. 303–317.
56. Gao, D.; Reiter, M.K.; Song, D.X. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In Proceedings of the Information and Communications Security, 10th International Conference, ICICS 2008, Birmingham, UK, 20–22 October 2008; Proceedings; Lecture Notes in Computer Science; Chen, L., Ryan, M.D., Wang, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5308, pp. 238–255. [CrossRef]
57. Ming, J.; Pan, M.; Gao, D. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In Proceedings of the Information Security and Cryptology—ICISC 2012—15th International Conference, Seoul, Republic of Korea, 28–30 November 2012; Revised Selected Papers; Lecture Notes in Computer Science; Kwon, T., Lee, M., Kwon, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7839, pp. 92–109. [CrossRef]
58. Devlin, J.; Chang, M.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, 2–7 June 2019; (Long and Short Papers); Burstein, J., Doran, C., Solorio, T., Eds.; Association for Computational Linguistics: Tilburg, The Netherlands, 2019; Volume 1, pp. 4171–4186. [CrossRef]
59. Hoffmann, J.; Borgeaud, S.; Mensch, A.; Buchatskaya, E.; Cai, T.; Rutherford, E.; Casas, D.d.L.; Hendricks, L.A.; Welbl, J.; Clark, A.; et al. Training compute-optimal large language models. *arXiv* **2022**, arXiv:2203.15556.
60. Andriesse, D.; Chen, X.; van der Veen, V.; Slowinska, A.; Bos, H. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In Proceedings of the USENIX Security Symposium, Austin, TX, USA, 10–12 August 2016.