BenchCouncil Transactions on Benchmarks, Standards and Evaluations 4 (2024) 100163

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Research article

BinCodex: A comprehensive and multi-level dataset for evaluating binary code similarity detection techniques

Peihua Zhang [a,b], Chenggang Wu [a,b,c], Zhe Wang [a,c,*]

[a] SKLP, Institute of Computing Technology, China
[b] UCAS, China
[c] Zhongguancun Laboratory, China

ARTICLE INFO

Keywords:
Dataset
Binary code similarity detection
Compiler optimization
Code obfuscation

ABSTRACT

The binary code similarity detection (BCSD) technique can quantitatively measure the differences between two given binaries and give matching results at predefined granularity (e.g., function), and has been widely used in multiple scenarios including software vulnerability search, security patch analysis, malware detection, code clone detection, etc. With the help of deep learning, the BCSD techniques have achieved high accuracy in their evaluation. However, on the one hand, their high accuracy has become indistinguishable due to the lack of a standard dataset, thus being unable to reveal their abilities. On the other hand, since binary code can be easily changed, it is essential to gain a holistic understanding of the underlying transformations including default optimization options, non-default optimization options, and commonly used code obfuscations, thus assessing their impact on the accuracy and adaptability of the BCSD technique. This paper presents our observations regarding the diversity of BCSD datasets and proposes a comprehensive dataset for the BCSD technique. We employ and present detailed evaluation results of various BCSD works, applying different classifications for different types of BCSD tasks, including pure function pairing and vulnerable code detection. Our results show that most BCSD works are capable of adopting default compiler options but are unsatisfactory when facing non-default compiler options and code obfuscation. We take a layered perspective on the BCSD task and point to opportunities for future optimizations in the technologies we consider.

1. Introduction

The widespread presence of binary code across diverse domains, including traditional PC software, emerging IoT device firmware [1], and malicious software, highlights the criticality of conducting research exclusively focused on binary code to effectively address software security concerns. In recent years, the binary code similarity detection (BCSD) technique [2–28] has garnered substantial attention and proven its versatility across diverse fields, including vulnerability discovery, malware detection, software plagiarism detection, patch analysis, software supply chain analysis, etc.

With the continuous advancements in machine learning, especially deep learning, learning-based methods have emerged as a prominent approach in mainstream BCSD tools [9,10,12,14,18,26,28,29]. These methods harness the power of neural network architectures and techniques to extract intricate patterns and representations from binary code, resulting in significant improvements in accuracy. By leveraging large-scale training datasets and sophisticated neural network architectures, learning-based BCSD techniques have achieved state-of-the-art accuracy in various tasks.

However, recent BCSD works have faced a challenge in distinguishing their capabilities and accuracy due to the absence of a standard dataset. Our study (detailed in Section 2.3) reveals that 22 BCSD works, which were conducted in the past decade and published on top venues, utilized different datasets. Consequently, it has become difficult to assess the true effectiveness of these methods, hindering the ability to make informed decisions and impeding progress in the field. Given the broad range of applications and the growing number of works in the BCSD field, the establishment of a standardized dataset is crucial.

To address this issue, we propose a standardized BCSD dataset dubbed as BinCodex.[1] This dataset aims to evaluate BCSD works in diverse scenarios, enabling researchers to compare and assess different techniques more effectively. Additionally, a standardized dataset would facilitate the validation of new methods, foster collaboration among researchers, and contribute to the overall progress of BCSD.

* Corresponding author at: SKLP, Institute of Computing Technology, China.
E-mail address: wangzhe12@ict.ac.cn (Z. Wang).

[1] "Bin" signifies the focus on analyzing binary code, while "Codex" conveys the idea of a comprehensive collection of code samples.

https://doi.org/10.1016/j.tbench.2024.100163
Received 26 February 2024; Received in revised form 23 April 2024; Accepted 7 May 2024
Available online 21 May 2024

We first thoroughly inspected all possible change points of the binary code and divided them into 4 groups: different platforms (e.g., x86 and arm), different compilers (e.g., GCC and Clang), different compiler options (including default and non-default), and different code obfuscation techniques.

Among them, compiler optimization options play a vital role in shaping the structure of binary code, resulting in noticeable differences. Previous studies have identified default options (e.g., O0/1/2/3) as a critical challenge in binary diffing [28,30]. Additionally, apart from default optimization levels, non-default options can also change binary code significantly [31] but are challenging to exhaustively analyze due to their large combination space, which encourages BinCodex to explore. Lastly, code obfuscation techniques [32–38] can alter code to modify binary characteristics, posing potential challenges to BCSD techniques. Recent work [39] has identified the impact of inter-procedural obfuscation on BCSD techniques, but its comprehensive evaluation is yet to be fully explored, which also motivates the BinCodex.

It is non-trivial to construct a comprehensive dataset for the BCSD technique due to the challenges posed by program diversity, efficiency, and measurement diversity. For example, (1) To balance the dataset size with the program diversity, choosing which programs is a problem; (2) Since the binary can be easily changed from several aspects, the enumeration of all possible change point combinations is impossible because of the large searching space. (3) BCSD tools differ in the granularity of their features and the representation of those features. How to normalize these disparities without deducing their accuracy is challenging.

To address the first challenge, we create the dataset with careful consideration of several factors, including a large amount of code (over 10 million lines of code), the representation of different binary code types (e.g., system software, compiler, interpreters, commonly used libraries, firmware, and typical vulnerable code), the diversity of code samples, varying levels of similarity, and different granularity of similarity detection tasks (e.g., function level and basic block level).

To tackle the second challenge, we aim to reduce the searching space in several directions. Firstly, we select default options among different compilers, which helps eliminate unnecessary variations. Secondly, we utilize a search-based compiler tool to explore the non-default options within a single compiler, avoiding redundant evaluations. Lastly, we choose a commonly used optimization level as a baseline to evaluate code obfuscation techniques, which reduces the number of obfuscated binaries while still capturing their impact.

To overcome disparities in evaluation metrics, we addressed the third challenge by normalizing features and using a standardized distance measurement: *precision ratio*. By abandoning different metrics from various BCSD tools, we ensure consistency in the evaluation process, enabling fair comparisons and a more reliable assessment of effectiveness.

To achieve a more precise evaluation of BCSD tools, we developed BinCodex as a multi-level dataset incorporating various code transformation levels instead of merging all binaries into a single pool. This granular evaluation enables a more detailed understanding of the effectiveness of BCSD tools in different scenarios.

BinCodex is implemented and evaluated on the Linux system. Eight state-of-the-art binary diffing tools (Diemph [29], OPTango [40], jTrans [28], Asm2Vec [12], Safe [41], DeepBinDiff [10], VulSeeker [14], and BinDiff [42]) are evaluated. The results cover various BCSD tasks, including pure function pairing and vulnerable code detection, and employ different classifications for different types of tasks. The results highlight that most BCSD works perform well when default compiler options are used but face challenges with non-default options. Additionally, while many BCSD tools demonstrate adaptability to intra-procedural code obfuscation, they struggle with inter-procedural obfuscation techniques. The evaluation provides a deep understanding of the current state of BCSD works, identifies the necessity of a standardized BCSD dataset, and points to opportunities for future optimizations in the field.

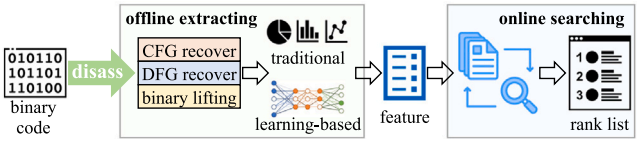

Fig. 1. The overall process of binary code similarity detection.

Our contributions can be summarized as follows:

- **A deep understanding of current BCSD works.** The paper is the first to provide a comprehensive summary of BCSD works based on their dataset characteristics. This understanding highlights the need for a standardized BCSD dataset.
- **A comprehensive BCSD dataset.** We present a BCSD dataset and propose three methods to enhance its comprehensiveness. These methods include dataset selection, searching space reduction, and metrics normalization. They ensure that the dataset includes representative programs, diverse features, and standardized metrics. We also perform multi-level evaluations to gain a detailed understanding of BCSD tools in different scenarios.
- **New insights from implementation and evaluation.** We evaluate BinCodex using eight state-of-the-art BCSD works, demonstrating its effectiveness in accurately assessing and comparing different methods. The insights gained from the evaluation contribute to fair evaluations, foster innovation, and advance the overall development of the BCSD field.

2. Background and motivation

2.1. Binary code similarity detection

Binary Code Similarity Detection (BCSD) is a technique used to analyze and compare binary code to identify similarities between binaries. It allows for quantitatively measuring differences and providing matching results at predefined levels of granularity, typically at the function level. As shown in Fig. 1, the process of BCSD typically begins with the disassembly of binaries, where the binary code is converted into assembly code, providing a representation that retains some semantic information of the program. This disassembly step serves as the foundation for most BCSD techniques. The workflow of BCSD techniques can be divided into two stages: offline feature extraction and online code search.

In the offline stage, tools extract features from binaries. Recent research focuses on determining which features should be extracted for effective BCSD. Based on the methods of the BCSD works, they can be classified into two categories [10]: traditional approaches and learning-based approaches.

- Traditional approaches extract low-level features from the binary code, such as opcode histograms. For example, Genius [43] and BinDiff [42] extract the number of string constants, numeric contacts, and different kinds of instructions as the identity of basic block and function, respectively. Besides, many works [6,7,16,25,44–46] have tried to extract semantic-level features as the identity of binary code, such as using I/O syntax to describe a basic block [6].
- Learning-based approaches leverage machine learning techniques to automatically learn discriminative features from the binary code. Various models have been used to extract features and learn representations that capture the underlying patterns in the code. For example, Asm2Vec [12] regards the assembly language as a special language, abstracts each element (e.g., opcode and operands) in the instructions as tokens in the natural language, and generates the representation of each token through training and clustering.

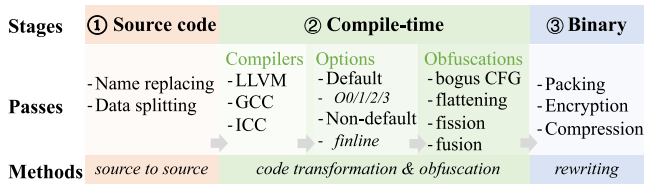

Fig. 2. Binary generation process, change points included.

Both traditional and learning-based approaches have their strengths and limitations. Traditional approaches often struggle with complex code transformations. Learning-based approaches, on the other hand, can adapt to diverse code representations and exhibit better robustness against code variations. However, they require large amounts of labeled data for training and can be computationally intensive.

In the online stage, tools calculate the similarity between the extracted features to identify matched pairs. This process can be seen as a search process, where the scale of the searching space is influenced by the granularity of the defined features. For instance, if the granularity is set at the function level, the searching space will be determined by the number of functions in the binaries. On the other hand, if the granularity is set at the basic block level, the searching space will be further expanded, considering the larger number of basic blocks in the code.

Additionally, the representation of the features impacts the method used for similarity calculation. When dealing with vector-related features, distance metrics such as Euclidean distance or cosine similarity are commonly employed. These metrics quantify the dissimilarity or similarity between feature vectors. On the other hand, when working with graph-related features, graph-matching methods come into play. These methods aim to find correspondences between nodes or subgraphs of the extracted features, taking into account both structural and semantic similarities.

2.2. Code transformation

The BCSD techniques meet challenges due to the ease with which binary code can be altered. Even instructions with identical semantics but different registers can have different binary representations. To this end, we first summarized the binary code generation process in Fig. 2, emphasizing the various points where changes can occur.

Source code transformations (Fig. 2 ①) primarily involve data obfuscation techniques that alter the format of data within a program. These transformations aim to prevent direct matching of data, often used to conceal sensitive information like private keys. However, in the BCSD scenario, these transformations are usually excluded as BCSD focuses on binary code rather than specific data values. Therefore, data obfuscation techniques targeting data format are not directly applicable to BCSD.

Dynamic code rewriting approaches (Fig. 2 ③), inspired by the concept of packing [47,48], focus on encoding or encrypting code as data. However, these techniques are also usually excluded in the context of BCSD due to they can be automatically unpacked [49–51] or be memory-dumped [52–55], and lose the transformation effect.

In contrast, compile-time transformations (Fig. 2 ②) focus on modifying the code during compilation without any further runtime modifications. These transformations have a significant impact on the structure of the resulting binary, making them a key area of interest in BCSD research. Compile-time transformations can manifest in several ways:

- Different compilers may implement the same optimization technique differently, leading to variations in the resulting binary code.
- Compiler options, including both non-default and default options, can influence the generated binary code (detailed in Section 2.2.1).
- Code obfuscations, which intentionally introduce complexity and disguise code, can result in substantial differences in binary code (detailed in Section 2.2.2).

2.2.1. Compiler optimization

Compiler optimization, which is originally used to improve the software performance (e.g., function inline) or reduce the binary size (e.g., dead-code elimination), has the potential to substantially modify the binary code, thereby exerting a significant impact on its differences. The binary code compiled from the same source code with different optimizations can exhibit a remarkably distinct code layout. Therefore, previous works have regarded compiler optimization as one of the key challenges to address in the BCSD task. For example, both Zeek [30] and jTrans [28] consider evaluating whether their methods can withstand the binary differences caused by different compiler optimization levels (including O0/1/2/3 and Os) as an important setup.

Except for the default compiler options, which integrate several optimization techniques, non-default optimization options also have a large effect on changing the binary code. Recent works have found they can expand the binary code difference significantly, which can be larger than the difference between O0 and O3 [31,40].

2.2.2. Software obfuscation

Software obfuscation transforms the program without changing its functionality to make it hard to analyze. There is an arms race between software obfuscation and BCSD. Software obfuscation does not want BCSD techniques to match un-obfuscated with obfuscated code successfully, and vice versa. There have been various techniques proposed for software obfuscation. For ease of introduction, we categorize them by obfuscation granularity:

- **Instruction level:** Instruction substitution (SUB) [33,35] replaces the original instruction with equivalent instruction(s). O-LLVM [35] designed 10 different strategies for arithmetic and logical operations.
- **Basic block level:** Bogus control flow (BCF) [33,35,63] inserts dead code into the original control flow and often utilizes permanent true or false predicates to prevent these codes from being executed, thereby ensuring the original functionality of the program.
- **Function level:** Control flow flattening (FLA) [33,35] converts the control flow of the function into the "`switch-case`" form, which is hard to analyze, and maintains the original jump relationship by controlling the values of the cases.
- **Module level:** Function fission [39] splits a function into multiple sub-functions. Conversely, function fusion [39] combines two functions into a single function. These code obfuscation techniques have proved their potential to alter function semantics significantly.

2.3. Motivation

We first conducted a comprehensive analysis of 22 BCSD works published in top venues over the past decade. These works were summarized based on their dataset characteristics and their ability to handle different code transformations during evaluations.

Our findings, as summarized in Table 1, reveal that there is a lack of a standardized dataset for BCSD techniques. Instead, the evaluated datasets were scattered across 36 different datasets or programs. It is worth noting that none of the 22 BCSD works utilized an identical dataset for evaluation. Each work employed its own self-constructed dataset, which allowed for detailed design considerations but lacked persuasiveness in terms of dataset consistency.

Since code transformation is a common source contributing to binary code differences, testing the resilience against transformation has become a common evaluation step for BCSD. As shown in Table 1,

Table 1
BCSD works from top venues in the last decade, summarized by their dataset characteristics and code transformation adaptability in their evaluations.

	Approach	Year	Venue	Dataset[a]	Code transformation adaptability			Diffing characteristics		
					Compiler[b]	Options[c]	Obfuscation[d]	Gran[e]	Metric	Platform[f]
1	DiEmph [29]	2023	ISSTA	2, 7, 8, 18, 27, 29, 32	❶❷	◐	–	F	Precision	①
2	sem2vec [56]	2023	TOSEM	2, 7, 10, 11, 20, 24, 27, 36	❶❷	◐	SUB, BCF, FLA	F	Precision	①
3	VulHawk [57]	2023	NDSS	7-11, 22, 25, 27, 29, 32, 35	❶	◐	–	F	AUC	① ② ③
4	OPTango [40]	2023	ISSRE	9, 30, 31	❶	●	–	F	Recall	①
5	TIKNIB [58]	2022	TSE	9, 16	❶❷❸	◐	SUB, BCF, FLA	F	self-defined	① ② ③
6	jTrans [28]	2022	NDSS	1, 9	❶❷	◐	–	F	Recall	①
7	ISRD [59]	2021	ICSE	5, 28, 36	❶❷	◐	–	F, B, I	Precision, Recall	①
8	Asteria [60]	2021	DSN	3, 12, 27	❶	○	–	F	ROC, AUC	① ② ④
9	DeepBinDiff [10]	2020	NDSS	7, 10, 11	❶	◐	–	B	Recall, Precision	①
10	Asm2Vec [12]	2019	S&P	4, 7, 9, 18-20, 24, 27, 29, 32, 36	❶❷	◐	SUB, BCF, FLA	F	Precision	①
11	SAFE [41]	2019	DIMVA	2, 6-10, 15, 21, 26, 27, 34	❶❷	◐	–	F	ROC, AUC, Precision, Recall	① ②
12	InnerEye [13]	2019	NDSS	2, 7, 10, 11, 27	❷	◐	–	B	ROC, AUC	① ②
13	alphaDiff [27]	2018	ASE	9, 14	❶❷	◐	–	P	Recall	① ②
14	VulSeeker [14]	2018	ASE	4, 7, 9, 12, 27	❶	◐	–	F	Precision	① ② ③
15	FirmUp [61]	2018	ASPLOS	9, 12	–	○	–	F	Precision	① ② ③
16	BinSequence [62]	2017	ASIACCS	23, 36	❸	○	–	F	Precision	①
17	IMF-SIM [46]	2017	ASE	7	❶❷❹	◐	SUB, BCF, FLA	F	Precision	①
18	BinGo [6]	2016	FSE	9	❶❷	◐	–	F	Precision	① ②
19	Genius [43]	2016	CCS	4, 7, 9, 12, 27	❶❷	◐	–	F	Recall, FPR	① ② ③
20	Multi-k-MH [16]	2015	S&P	4, 7, 12, 27	❶❷	◐	–	B	TP, FP	① ② ③
21	CoP [44]	2014	FSE	13, 17, 27, 33	❶❹	◐	BCF, FLA	P	self-defined	①
22	BinSlayer [11]	2013	PPREW	7	❶	○	–	P	Precision	①

[a] Including 36 micro-datasets. 1: ArchLinux repositories, 2: Binutils, 3: Buildroot, 4: BusyBox, 5: Bzip2, 6: CCV, 7: Coreutils, 8: Curl, 9: CVEs, 10: Diffutils, 11: Findutils, 12: Firmwares, 13: Gecko, 14: GitHub repositories, 15: GNU Scientific Library, 16: GNU software packages, 17: Gzip, 18: ImageMagick, 19: Libcurl, 20: Libgmp, 21: Libhttpd, 22: Libmicrohttpd, 23: Libpng, 24: LibTomCrypt, 25: Mtools, 26: OpenMPI, 27: OpenSSL, 28: PreComp, 29: PuTTY, 30: SPEC CPU 2006, 31: SPEC CPU 2017, 32: SQLite, 33: Thttpd, 34: Valgrind, 35: Wget, 36: zlib.

[b] Including 4 compilers. ❶: GCC, ❷: Clang, ❸: MSVC, ❹: ICC.

[c] Consideration of compiler options. ○: None, ◐: Only default options, ●: Both default and non-default options.

[d] Consideration of code obfuscations. -: None, SUB: instruction substitution, BCF: bogus control flow, FLA: control flow flattening, IBV: Insert bogus variables, SSO: split structure object.

[e] Granularity for BCSD. P: program, F: function, B: basic block, I: instruction.

[f] The implementation platform of the BCSD works. ①: x86, ②: ARM, ③: MIPS, ④: PowerPC.

many BCSD works have considered binaries compiled with different mainstream compilers, and most of them concentrate on the GCC and Clang.

Under a specific compiler, most of them only considered the default optimization options such as O0 to O3, but failed to explore non-default optimization options (only 1 work). However, the trend of compiling binaries using non-default options settings other than the default options has been increasing in recent years [31], for example, the virus binaries used non-default options to hide their binary code features. To this end, it is essential to evaluate the adaptability of the BCSD tools under non-default compiler options.

In terms of code obfuscation, only a few works have considered it, and the evaluation is often limited to intra-procedural obfuscation techniques (e.g., at the statement, basic block, or function level), which do not fundamentally change the semantics of each function. Thus existing BCSD techniques can still capture the obfuscation effect. However, recent advancements in code obfuscation techniques have increasingly focused on inter-procedural obfuscations, which have shown the ability to alter function semantics [39], which is a crucial factor in defeating BCSD techniques.

As for the granularity, most works focus on function-level, while some also consider basic block and instruction-level differences for their specific detection purpose. Besides, the matrix that measures the accuracy of BCSD works also varies. Some of them use standard metrics like true positive (TP) or false positive (FP). Precision@k and Recall@k are also used to measure the proportion of relevant results among the top k retrieved items. Some works also define the metric based on their specific features, which does not apply to others.

In the BSCD tool landscape, most tools are primarily developed for the x86 platform, with some considering cross-platform compatibility. While it is theoretically possible to adapt x86-focused tools to other platforms, such adaptations would involve additional efforts such as integrating disassembler backends and platform-specific API construction.

As BCSD techniques continue to improve in feature extraction and binary code representation, particularly with the application of deep learning, their ability to capture semantics becomes more robust. Consequently, while many of these works claimed superiority over others, the lack of dataset standardization raises the possibility that dataset mismatch could be a contributing factor to their comparative results. Therefore, there is a need for *a unified dataset that covers a wide range of representative programs and datasets*. A standardized dataset would facilitate fair comparisons among different BCSD techniques, enabling researchers to make more reliable and meaningful claims about their effectiveness.

Table 2
The detail of BinCodex dataset, including the origin of the dataset and the detail of multi-level workloads.

Dataset		Multi-level setting					
Info	Statistics	Workloads	Task	#Binaries	Compiler	Metrics	Options/Tools
Coreutils SPEC CPU 2006	113 progs 2.0M LOC	BIN-Default	Adaptability to default compiler options.	whole dataset	GCC Clang	Precision@1	O0-O3, Os, Ofast O0-O3
SPEC CPU 2017 Libraries	7.8M LOC 522K LOC	BIN-Nondefault	Adaptability to non-default compiler options.	whole dataset	GCC	Precision@1	BinTuner [31]
Firmwares #CVE	418K LOC 69	BIN-Obfuscated	Adaptability to different obfuscation methods.	whole dataset	Clang	Precision@1	O-LLVM [35], Khaos [39]
#Binaries #Functions	3.1K 68.9M	BIN-Vulnerable	Vulnerable function searching.	CVEs only	Clang	Precision@1/10/50	O-LLVM [35], Khaos [39]

Besides, the dataset should be able to cover different kinds of transformations to *fully measure the adaptability of BCSD techniques against possible transformations, including non-default compiler options and inter-procedural code obfuscations*. This viewpoint aligns with the literature published from both offensive and defensive perspectives:

- Many BCSD works have acknowledged the impact of inter-procedural optimizations (e.g., function inlining) on diffing accuracy [7,43,59,61,62,64–68].
- Existing research has demonstrated that inter-procedural obfuscation can decrease the accuracy of BCSD tools (up to 60%) with minimal overhead (less than 5%) [39].
- Existing research [31,40] has also proved that non-default options can decrease the accuracy of BCSD tools.

Given that all current BCSD tools only have a common implementation on the x86 platform, this paper follows the same setting to evaluate as many BCSD tools as possible. By concentrating on this specific instruction set, the dataset can provide a more targeted assessment of the capabilities and limitations of BCSD tools.

3. Methodology

Considering the above factors, this paper proposes a unified dataset that incorporates a wide range of programs and possible transformations, particularly non-default compiler options, and inter-procedural code obfuscation techniques, and uses it to explore and evaluate existing BCSD techniques.

3.1. Challenges

Designing a dataset for BCSD poses several challenges, which can be summarized as follows:

- **Creating a representative dataset (C1):** On the one hand, the dataset's codebase should be extensive enough to encompass a wide range of code features. On the other hand, the dataset needs to encompass representative programs from various application scenarios commonly encountered in BCSD.
- **Generating diverse code variants efficiently (C2):** Enumerating all possible code variants is impractical due to the numerous change points discussed in Section 2.2. For instance, even a single change point, such as compiler options in GCC, includes over 200 options resulting in more than 2^{200} combinations. Exhaustively combining these options to compile the same source code would result in an infeasible number of binary variants. How to generate a dataset with diverse features without exhaustively enumerating all possible combinations is challenging.
- **Normalizing features and similarity calculation (C3):** BCSD tools differ in the granularity of their features and the representation of those features. Furthermore, the similarity calculation methods employed by each tool may vary. How to normalize these disparities without deducing their accuracy is challenging.

The following subsections detail the design of BinCodex and address the above challenges. By doing so, a meticulously designed BCSD dataset can be constructed, enabling accurate evaluation of BCSD tools.

3.2. The BinCodex dataset

The Origin of Datasets (C1). To overcome the C1 challenge, we conducted an extensive analysis of widely used software across different domains and carefully selected representative programs from various application scenarios commonly encountered in BCSD. As Table 2 shows, our dataset includes programs from diverse domains, such as utility programs (e.g., Coreutils), compilers (e.g., GCC in SPEC CPU), language interpreters (e.g., Perl interpreter in SPEC CPU), JavaScript engines (e.g., QuickJS), network protocols (e.g., LibCurl), web applications, libraries (OpenSSL), embedded firmware (e.g., BusyBox), and artificial intelligence applications (e.g., alpha-beta tree search and Monte Carlo tree search).

By including programs from such diverse domains, our dataset accurately reflects the challenges and complexities faced by BCSD techniques in real-world scenarios. It provides a comprehensive evaluation of the accuracy of BCSD tools, as it encompasses all the types of programs used in existing BCSD works in Table 1. This ensures that BinCodex serves as a valuable dataset for evaluating and comparing different BCSD techniques.

Searching Space Reduction (C2). The enumeration of all compiler options and obfuscation techniques for all compilers is impossible because of the large searching space. To this end, our reduction mainly contains the following 3 steps.

Inter-compiler reduction. In our dataset, we first chose to focus on the GCC and Clang compilers for alignment with existing BCSD research and their widespread usage with extensive optimization capabilities. Based on that, we reduce the inter-compiler searching space by *concentrating binary variants under their default compiler options*. For example, as shown in Table 2, GCC and Clang are both used when considering the default compiler options (*BIN-Default* workload), and only GCC is used when considering the non-default compiler options (*BIN-Nondefault* workload).

The exclusion of Clang in the *BIN-Nondefault* workload was a deliberate decision made for the purpose of introducing a specific challenge and evaluating the adaptability of BCSD tools to non-default compiler options using a single compiler. By concentrating on GCC for the non-default compiler options, we aimed to isolate and assess the challenges posed by non-default options in binary code analysis. This approach allowed us to investigate the specific impact of non-default options on BCSD without the additional variability introduced by using multiple compilers in this particular workload.

Intra-compiler reduction. Exhaustively combining non-default options to compile the same source code would result in an infeasible number of binary variants. Besides, different programs are likely to use different combinations of options since their code patterns are different. For example, the `funroll-loops` option can change binary for programs with loops but has no effect on those programs without loops. To acquire the program-specific combinations, we tailor *search-based*

iterative compilation for the auto-tuning of non-default compiler options. Specifically, we utilize the BinTuner [31] tool, which uses the genetic algorithm to reveal the optimal effects on binary code differences. By adopting such an efficient approach, we can strategically select a subset of compiler flags, optimization levels, and code transformations to generate a diverse set of code variants without the need for exhaustive enumeration.

Obfuscation reduction. In our dataset, we have included obfuscation techniques as an important aspect of evaluation, considering the sensitivity of BCSD techniques to code obfuscation. To represent this, we selected two well-known obfuscation tools based on the LLVM infrastructure.

The first obfuscation tool is O-LLVM [35], a popular compiler-based obfuscation tool widely used in software engineering, systems security, and programming language research. O-LLVM offers three *intra-procedural obfuscation* methods of different granularity: `SUB`, `BCF`, and `Fla`.

Additionally, we incorporated the *inter-procedural* obfuscation tool called Khaos [39] in our dataset. Khaos focuses on changing function semantics, a key factor in defeating BCSD techniques. It provides two obfuscation primitives named `Fission` and `Fusion`.

When generating obfuscated binaries, we established a baseline by using the commonly used compiler option instead of generating obfuscated binaries for all options. Specifically, we chose the O2 optimization level as the baseline. This decision was made because O0 and O1 optimization levels are less commonly used in real-world applications, while the O3 optimization level may remove the obfuscation effect of O-LLVM (further discussed in Section 6).

Metrics (C3). To overcome the disparities of evaluation metrics, it is crucial to establish a standardized metric for similarity calculation. From our observation, these disparities come from the online searching stage (introduced in Section 2.1). Specifically, after the feature is extracted from the binary file and the distance is calculated, different tools use diversity methods to explain the accuracy. To this end, we abandon the different metrics of different BCSD tools and use a consistent measurement for the accuracy — *precision@k*, which is also commonly used in several BCSD works [12,28,29,31,56].

In the BCSD scenario, the search results are presented as a ranked list. Consider two binary files P and Q that are compiled from the same source code but with different options. Each of them has N functions, where $P = \{p_1, p_2, \dots, p_i, \dots, p_N\}$ and $Q = \{q_1, q_2, \dots, q_i, \dots, q_N\}$. The ground truth is p_i matches q_i, where $p_i \in P$ and $q_i \in Q$. For each function $p_i \in P$, the BCSD tools identify the top-k functions in Q that are most similar to p_i. These functions are ordered by a similarity score, indicating their rank $Rank_{q_i}$ in the list. Utilizing the definitions of precision, the precision ratio $precision@k$ is determined using the following metric:

$$precision@k = \frac{1}{N}\sum_{p_i \in P}(Rank_{q_i} \le k) \times 100\% \quad (1)$$

For example, if the $precision@10$ of a specific BCSD tool exceeds 80%, it means over 80% functions in P are matched correctly in the top-10 candidate functions from Q. By normalizing the metrics of different BCSD tools and mapping the results to a common representation space, the dataset can facilitate fair and consistent evaluation.

Multi-level workloads. As shown in Table 2, BinCodex contains 4 different workloads. The *BIN-Default* workload aims to evaluate the adaptability to default compiler options, which is commonly used in real-world programs. The *BIN-Nondefault* workload is specifically designed for the emerging use of non-default options, which are confirmed by security analysts that these options can make reverse engineering analysis complicated [31,69]. Obfuscation often introduces challenges to existing BCSD tools, given that they extract syntactic- or graph-level information, which does not necessarily reflect the real functionality, thus the *BIN-Obfuscated* workload is used to evaluate if these tools are vulnerable to assembly codes with similar functionality but the differing appearance. Considering the diverse application scenarios of BCSD, BinCodex also adopts 69 CVEs to form the *BIN-Vulnerable* workload for the specific vulnerable code searching scenario and normalizes the measurement using *precision@k*.

The workloads are designed in increasing order of difficulty for BCSD tools, with the first three of them representing a progression from easier to harder transformations. For instance, most BCSD tools can handle transformations between O0 and O3, but only a few consider non-default options, and they all face challenges when dealing with inter-procedural obfuscation [39]. The *BIN-Obfuscated* workload presents similar difficulties to BCSD tools as the *BIN-Vulnerable* workload, as the vulnerable codes in the latter are also obfuscated. To precisely measure the vulnerability search result, it uses more comprehensive measurements — *precision@1/10/50*.

Algorithm 1: Dataset generation algorithm.

Input: Program source code set $progSet$, N_{def}, N_{nondef}, N_r, N_{obf}
Output: Binary variants dataset $binSet$

```
binSet ← {}
for each program prog ∈ progSet do
    for i = 1 to N_def do
        def ← getDefaultOption(i);
        bin_def ← Compile prog with option def;
        binSet[prog].append(bin_def);
    end
    for i = 1 to N_nondef do
        def ← getDefaultOption(i);
        bin_def ← Compile prog with option def;
        max_diff ← 0;
        bin_max ← bin_def;
        for i = 1 to N_r do
            bin_nondef ← BinTuner(bin_def);
            diff ← BinDiff(bin_def, bin_nondef);
            if max_diff < diff then
                max_diff ← diff;
                bin_max ← bin_nondef;
            end
        end
        binSet[prog].append(bin_max);
    end
    for i = 1 to N_obf do
        obf ← getObfOption(i);
        bin_obf ← Compile prog with option obf;
        binSet[prog].append(bin_obf)
    end
end
return binSet;
```

3.3. Workflow of BinCodex

For the dataset listed in Table 2, algorithm 1 outlines the workflow of BinCodex to generate their binary variants. For each program $prog$ in the $progSet$, three types of variants, namely default compiler options, non-default compiler options, and code obfuscation, are applied to $prog$ to produce a set of binary variants, which are then added to $binSet$. N_{def}, N_{nondef}, and N_{obf} denote the number of default compiler options, non-default compiler options, and code obfuscation techniques that need to be generated for each $prog$, respectively.

Firstly, when generating the binary variants with default compiler options, we choose all default compiler options (2 compilers, 10 options in all) in Table 2 to apply. Secondly, to ensure the diversity of the generated samples under non-default compiler options, we perform N_r rounds of generating binary variants by the BinTuner [31] tool and use

Table 3
Summary of the chosen diffing works.

BCSD works	Diffing granularity	Symbol relying	Time/memory consuming	Call-graph lacking
Asm2Vec [12]	function	N	N	Y
SAFE [41]	function	N	N	Y
DeepBinDiff [10]	basic block	N	Y	N
jTrans-0 [28]	function	N	Y	Y
jTrans [28]	function	N	Y	Y
BinDiff [42]	all	Y	N	N
DiEmph [29]	function	N	Y	Y
VulSeeker [14]	function	N	Y	Y

the BinDiff [42] to select the most different variant bin_{max}. Thirdly, for each selection of the obfuscation technique, we generate all obfuscated variants from them under the same baseline. After the above process, the BinCodex contains 68,930,974 binary functions (3,627,946 functions in the source code, with 19 variants for each function), which is the largest dataset to our best knowledge.

4. Implementation and evaluation

The BinCodex is implemented under the Linux operating system. The evaluation is conducted on Ubuntu 22.04 (Kernel v5.15.0) which runs on the x86_64 platform (Intel Xeon Gold 6148 CPU with 160 cores and 1.5TB memory) since all current BCSD tools only have common implementation in the x86 platform. This section aims to answer the following questions:

- (Q1) How do the state-of-the-art BCSD works perform on BinCodex?
- (Q2) What is the impact of the three levels of code transformation techniques, namely default compiler options, non-default compiler options, and code obfuscation, on the effectiveness of BCSD works?
- (Q3) What kinds of code transformation have the greatest impact on the BCSD works?

Corresponding to the four workloads in BinCodex, the evaluation consists of four parts, including the adaptability to the default compiler options using GCC and Clang (Section 4.1), the non-default compiler options using GCC (Section 4.2), the code obfuscation using Clang (Section 4.3), and a specific application scenario of BCSD — vulnerable function searching (Section 4.4).

Confrontation targets. We leverage 8 state-of-the-art BCSD tools to evaluate BinCodex. Their characteristics are summarized in Table 3. All learning-based tools among them are retrained on BinCodex. The column "*symbol relying*" means whether the un-stripped binaries have side-effects or not, for example, BinDiff [42] uses function names to reduce the searching space. The column "*time/memory consuming*" means the diffing process takes a long time (e.g., over one 1 month) or requires a lot of memory (e.g., more than 1 TB). The column "*call-graph lacking*" means whether the call-graph is used as the feature. Their detailed techniques are as follows:

- **Asm2Vec** [12] employs random walks on the function CFG to sample instruction sequences and then uses the PV-DM model to learn function and instruction token embedding jointly.
- **SAFE** [41] utilizes a word2vec model to generate instruction embeddings and proposes a recurrent neural network for function embedding generation.
- **DeepBinDiff** [10] is a learning-based work for diffing the semantic similarity in basic block granularity.
- **jTrans-0** [28] incorporates control flow information from binary code into transformer-based language models for function embedding. **jTrans** [28] fine-tunes the pre-trained model to generate function embedding for the supervised learning task of binary diffing.
- **BinDiff** [42] is an industry-standard binary diffing tool, which diffs the semantic similarity in different granularity (e.g., instruction, basic block, function, call graph).
- **DiEmph** [29] detects undesirable instruction distribution biases caused by specific compiler conventions and repairs them by removing them from the dataset and fine-tuning the models.
- **OPTango** [40] is a transformer-based multi-central representation learning approach, which purely explores the solution to build a compiler optimization-agnostic tool.
- **VulSeeker** [14] is a vulnerability seeker that integrates function semantic emulation with semantic learning.

The selection of the eight BCSD tools for evaluation was carefully considered to cover a diverse range of techniques and approaches in the field. The rationale behind their selection can be summarized as follows:

1) *Representation of Different Techniques.* The chosen tools represent a variety of techniques employed, which ensures a comprehensive evaluation of BinCodex's performance across different methodologies, allowing us to analyze its effectiveness in various scenarios.
2) *Learning-Based Approaches.* Given almost all state-of-the-art diffing tools are learning-based, we included 6 learning-based tools. By evaluating BinCodex with these tools, we can assess its compatibility with learning-based approaches and compare its performance against state-of-the-art models.
3) *Industry-Standard Tool.* BinDiff, known as an industry-standard binary diffing tool, is included to provide a benchmark for comparison. Its comprehensive analysis capabilities, including instruction-level, basic block-level, function-level, and call graph-level comparisons, make it a valuable tool for evaluating BinCodex.
4) *Compiler Optimization-Agnostic Tools.* Tools like OPTango were selected to evaluate BinCodex's robustness against different compiler optimization variants. These tools focus on building optimization-agnostic models to overcome the challenges posed by variations in compiler optimization levels.
5) *Vulnerability Detection.* VulSeeker, a vulnerability seeker tool, is included to assess BinCodex's effectiveness in detecting vulnerabilities in binary code. This tool incorporates semantic emulation and learning techniques to identify potential vulnerabilities, providing a specific use case for evaluation.

Overall, the selection of these eight BCSD tools ensures a comprehensive assessment of BinCodex. By including tools with diverse techniques, learning-based approaches, industry-standard benchmarks, optimization-agnostic models, and vulnerability detection capabilities, we can thoroughly evaluate BinCodex's performance, compatibility, and effectiveness across different dimensions of binary code similarity analysis.

Each BCSD tool has its own specific application scenario and capabilities. In the evaluation process, the tools were selected and used based on their suitability for the respective tasks. For instance, DeepBinDiff [10] focuses on diffing binaries at the basic block level. Therefore, it was specifically evaluated in the code obfuscation part, where obfuscation methods can alter the basic blocks of the code. DiEmph [29], on the other hand, relies on jTrans [28] as its underlying tool. As a result, DiEmph was solely used in the code obfuscation part of the experiment, where jTrans was evaluated comprehensively. OPTango [40] was evaluated in the compiler-option-relevant parts of the experiment, aligning with its claim of being specifically designed for compiler options.

4.1. Adaptability to default compiler options

To evaluate the adaptability of BCSD tools under default compiler options, the experiment used the *BIN-Default* workload in the BinCodex.

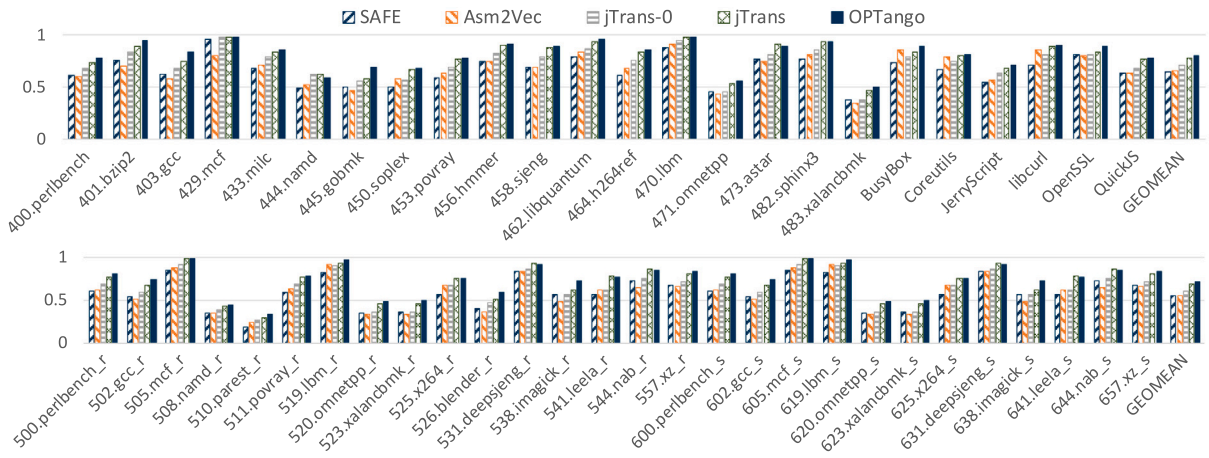

Fig. 3. The precision@1 results of chosen binary diffing works for binaries generated by default compiler options.

Space reduction. To avoid the exhaustive enumeration of different optimization levels, we defined a baseline by selecting binaries generated with the O0 and O2 options (including Os and Ofast for GCC). These baseline binaries were used for the targets of diffing, where binaries generated with the O1 and O3 options were sequentially compared. For example, the binary *400.perlbench-O1* would be compared with both *400.perlbench-O0* and *400.perlbench-O2* binaries. Similarly, the *400.perlbench-O3* binary would also be compared with them. The results of these comparisons were then averaged.

This experimental setup allowed for efficient exploration of all default compiler options. From the optimization perspective, all the binaries can be regarded as two groups: unoptimized and optimized. The baseline contains the unoptimized and optimized binaries at the same time (e.g., O0 as the unoptimized, O2 as the optimized), as well as the binaries used for querying (e.g., O1 as the unoptimized, O3 as the optimized). In this way, the number of options is reduced from 2^{10} (10 options in all) to 2 (optimized and unoptimized).

Results. The experiment's results are shown in Fig. 3. The precision@1 means the BCSD tool can match the optimized function with the unoptimized function on the first candidate in its rank list, and higher accuracy means higher adaptability. For example, the results indicate that all the BCSD tools achieved a precision rate higher than 50%, which means all the BCSD tools can match over half of the functions as the first candidate. OPTango [40] demonstrated the best adaptability, followed by jTrans/jTrans-0 [28], Asm2Vec [12], and SAFE [41].

An interesting observation was made during the experiment: *larger binary sizes tended to yield lower accuracy*. This phenomenon can be attributed to the relationship between the searching space and the number of functions. As the number of functions increases, the searching space expands, and the likelihood of finding similar functions within the same binary also grows. Consequently, the false positive ratio increases. This observation was also noted in subsequent evaluations involving non-default compiler options and code obfuscations, which is discussed further in Section 6. Additionally, the OpenMP-related programs in SPEC CPU 2017 are slightly harder for the BCSD tools to achieve a high precision compared with non-OpenMp binary variants.

4.2. Adaptability to non-default compiler options

In this subsection, the adaptability of BCSD tools under non-default compiler options is evaluated using the *BIN-Nondefault* workload. BinTuner [31] is utilized to generate binaries with non-default options effectively.

The BinTuner tool. BinTuner [31] follows a specific procedure to generate binaries. Initially, it selects a baseline binary, such as the one generated with the O0 option. Then, it iteratively searches for the target binary by combining non-default options. During the search process, BinTuner leverages the differences between the baseline binary and the target binary to guide the selection of the next combination of options. This iterative approach allows BinTuner to effectively explore the vast space of non-default compiler options and generate binaries for evaluation purposes.

However, BinTuner is not a panacea, especially when the different optimization levels are considered. For example, we first set the O0 as the baseline for BinTuner, after it generated the binaries, we utilized BinDiff [42] to calculate the difference with the baseline. As depicted in Fig. 4, we observed that a significant portion of the generated binaries exhibited similarity to the O3 variants, indicating a lack of diversity. This outcome suggests that BinTuner may not effectively explore the entire optimization space when only one baseline is considered.

Multi-baseline setting. Firstly, to enhance the diversity of the evaluation, different default optimization levels (O0, O1, O2, O3) were chosen as the baselines to generate binaries using BinTuner. This approach resulted in the creation of four groups of binaries (Ot0, Ot1, Ot2, Ot3) where each group was based on a specific default optimization level. Secondly, in addition to the individual groups, a fifth group of binaries (Ot4) was generated by setting all four default optimization levels as the baseline simultaneously. This group's binaries were distinct from those generated by any of the default options alone. Lastly, to provide a comprehensive evaluation, all five groups of binaries (Ot0, Ot1, Ot2, Ot3, Ot4) were also merged into a single group (Ot). This merged group encompasses the entire range of binary variations generated by BinTuner.

Overlap in default and non-default options. There is an overlap between default and non-default compiler options, for example, both of them have `-funroll-loop` and `-finline` options. However, the default option binds options in the specific combination (for example, O3 enables `-funroll-loop` and `-finline` at the same time), while the non-default option does not bind the combination (for example, enable `-finline` option but disable `-funroll-loop` option at the same time), which can enlarge the binary difference. The detailed options used by BinTuner are shown in Table 6.

It is important to note that each binary was created by incorporating over 100 different non-default option configurations. The details of these configurations can be found in Table 7. By merging the different groups of binaries and conducting binary similarity detection on the combined dataset, the evaluation aimed to assess the adaptability and performance of the BCSD tools across a wide range of non-default compiler option configurations.

Result. The results are shown in Fig. 5, where every group shows the same trends of accuracy. Consider the Ot2 set as an example, where the target binaries are generated by searching for the maximum difference from the O2-based binary, Asm2Vec achieves the precision@1 scores of 0.537, SAFE and jTrans-0 demonstrate similar diffing accuracy,

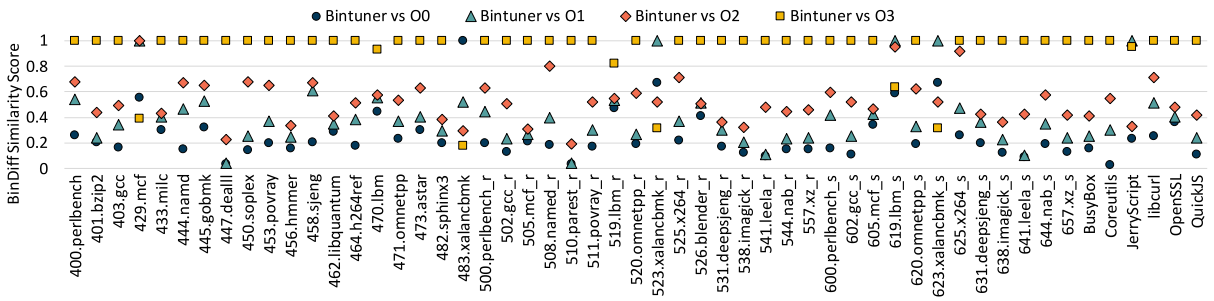

Fig. 4. The similarity score (normalized) of binaries generated by BinTuner [31] by only using the O0 as the baseline.

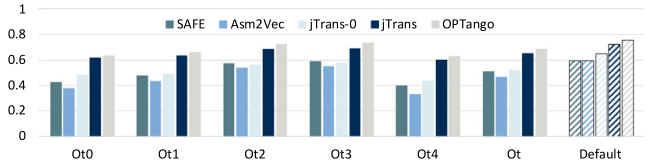

Fig. 5. The precision@1 results of chosen binary diffing works for binaries generated by non-default compiler options in GCC compiler. Higher similarity means higher adaptability.

achieving 0.571 and 0.563, respectively. jTrans outperforms jTrans-0 by 12.1% but falls 4.1% behind OPTango, which stands out with a precision of 0.726.

We also put the accuracy result of binaries generated by the default options in the rightmost group in Fig. 5(`Default`). By comparing it with the `Ot4` group, we can notice a significant decrease in accuracy under the non-default option settings compared to the default option settings for these binary diffing methods, which highlights the motivation of BinCodex.

4.3. Adaptability to code obfuscation

Obfuscators. As discussed in Section 3.2, we used the commonly used compiler option O2 as the baseline, and generated the obfuscated and un-obfuscated binaries in the *BIN-Obfuscated* workload. We keep all the binaries un-stripped to get the ground truth of paring. Besides, the Khaos [39] changes the number of functions because it is an inter-procedural obfuscation tool, thus we followed its evaluation setting to relax the requirements for *Precision@1*. The O-LLVM [35] remains unchanged since it does not change the function count. To ensure the consistency of the evaluation environment for the obfuscation tools, we upgrade the LLVM version of O-LLVM [35] to 9.0.1, which is the same as the Khaos [39]. All the existing BCSD tools adopted the old version of LLVM, which loses the obfuscation effect when facing a high optimization level (e.g., O3).

Histogram of Opcodes. We collected some internal information on the dataset to reveal the details of BinCodex. We used the *objdump* tool to disassemble all the binaries in BinCodex and calculated the histogram of opcodes. By comparing the vectors of opcodes between the original and obfuscated binaries, we can calculate the vector similarity. Since different programs may have varying numbers of instructions, we normalized the distances using the maximum similarity among all obfuscated programs. As depicted in Fig. 6, the distribution of opcodes varies in different obfuscation methods, in which `Fission` generates the binaries with the longest opcode distance and SUB generates the shortest. It demonstrates that BinCodex contains a diverse range of opcodes, and thus can fully cover the obfuscated binary scenario.

Results. After the binary is generated, we evaluated the accuracy of the selected BCSD tools by comparing obfuscated and un-obfuscated binaries. As depicted in Fig. 7, the result is divided into different groups by different obfuscation methods. The precision@1 means the BCSD tool can match the obfuscated function with the unobfuscated function on the first candidate in its rank list, and higher accuracy means higher adaptability. Among these obfuscation methods, compared with inter-procedural code obfuscation (`Fission` and `Fusion`), the BCSD tools are more adaptive to intra-procedural obfuscation (e.g., SUB and BCF). Besides, while the control flow flattening (FLA) archives a strong obfuscation effect, it also brings extremely high runtime overhead (2.8x slowdown), which is undesirable in real-world scenarios.

Binary variants that are obfuscated by the instruction substitution (SUB) are the easiest to pair, mainly because it does not change the function's control flow but only replace the opcodes of instructions, which are easy to adopt since these opcodes belong to the same opcode family, e.g., ALU. Although the bogus control flow (BCF) inserts dead code in the function, it has merely interfered with the original function control flow, thus also bringing limited obfuscated effect.

Inside each obfuscation method, different BCSD tools show different adaptability. We discuss them as follows:

- VulSeeker [14] takes more than 1 day to diff two large binaries and often gets killed due to memory limit. To speed up VulSeeker, we group the related functions into small groups to manually reduce the searching space.
- SAFE [41] and Asm2Vec [12] showed their advantages on intra-procedural obfuscation by capturing the semantics of functions.
- Because DeepBinDiff [10] uses the basic block as its granularity, its searching space is much larger than others and brings the time/memory consuming issue (e.g., requiring more than 10TB memory, waiting several weeks to compare two binaries). To this end, We reduced the dataset for DeepBinDiff [10] by only using programs with less than 40k lines of code. In this setting, it achieved higher accuracy in inter-procedural obfuscation methods (e.g., `Fusion`). This is because Khaos [39] uses original functions to obfuscate each other, lacking material reduces the obfuscation effect.
- Since we retrained the model of jTrans on BinCodex, it is more accurate than the pre-trained model jTrans-0.
- After DiEmph [29] detected the undesirable instruction distribution biases, it fine-tuned the models of jTrans [28], thus its adaptability is slightly higher than jTrans [28].
- Since BinDiff [42] takes advantage of function names of the symbols, its results are a little higher than others.

4.4. The ability to search vulnerable code

We use the *BIN-Vulnerable* workload in BinCodex to evaluate the ability to search real-world vulnerable code, each program contains at least one vulnerability (detailed in Table 5). In this experiment, we used Asm2Vec [12], VulSeeker [14] and SAFE [41] to calculate the *precision@n ratio* (the rank of truly matched pair in the matched result) of vulnerable functions. The reason why other tools were not used is that they only give top-1 matched results. We calculated *precision@1/10/50 ratio* of vulnerable functions.

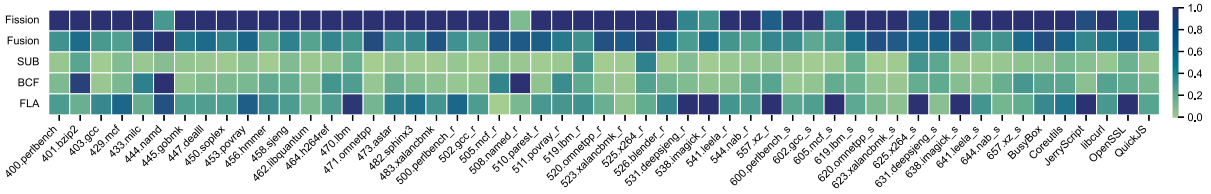

Fig. 6. The heat map of opcode histogram distance (normalized) of obfuscated binaries in BinCodex.

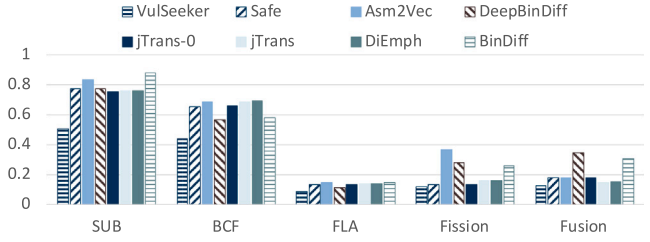

Fig. 7. Similarity results of chosen binary diffing works and tools. Diffing works use precision@1, and BinDiff uses its normalized scores. Higher similarity means a higher adaptability.

Table 4
Comparison with other BCSD datasets.

Dataset	Statistics		Transformations[a]		
	Source	#Funcs	Def	Non-Def	Obf
Esh [3]	8 CVE	1.5K	✔	✘	○
BINKIT [70]	GNU packages	36M	✔	✘	◐
BinaryCorp [28]	GNU packages	25M	✔	✘	○
BinCodex	GNU packages, libraries, SPEC CPU, firmware, 69 CVE	69M	✔	✔	●

[a] The possible transformations that the dataset considered, including default compiler options (Def), non-default compiler options (Non-Def), and code obfuscations (Obf, ○: none, ◐: only intra-procedural obfuscation methods, ●: both intra-procedural and inter-procedural obfuscation methods).

Fig. 8 gives the experimental results, which are divided into three groups as precision@1/10/50. The precision@1 means the BCSD tool can match the obfuscated function with the unobfuscated function on the first candidate in its rank list, while the precision@10 means it can match them in the top-10 candidates in its rank list, thus the accuracy is ascending in from precision@1 to precision@50. For example, the *precision@50 ratio* of `Fission` on Asm2Vec is around 0.5, which means about 50% of vulnerable functions can be found within the top-50 ranked functions using Asm2Vec.

Inside the same group, the *precision ratio* can reflect the vulnerable function searching ability of different BCSD tools. For example, for the `precision@1` group, Asm2Vec [12] is more accurate than SAFE [41], and both of them are better than VulSeeker [14]. Besides, it can also reveal the ability to hide the vulnerable function with different obfuscation methods. For example, under the same precision and binary diffing tool (e.g., `precision@50-Asm2Vec`), *Fission* and *Fusion* are better than *SUB*, *BCF*, and *FLA*.

5. Related works

Existing research often lacks transparency when it comes to disclosing their datasets. Among the few open datasets available, as shown in Table 4, Esh [3] proposed a dataset purely for vulnerability searching but with only 8 vulnerabilities. BinKit [70] claims as the largest binary dataset, however, it only consists of software from GNU packages like GNUtils and coreutils. Additionally, it only includes variants of compiler options for default optimization levels ranging from O0 to Ofast, which are fully covered in our proposed BinCodex dataset. Although BinKit considers code obfuscation, it only focuses on intra-procedural methods, which have been proven ineffective in both our evaluation and other works [39]. Another recent open-source dataset, Binarycorp [28], claims to offer diversity in terms of project size and application scenarios compared to BinKit. However, it only utilizes five compile options from O0 to Os, which are also included in our BinCodex dataset.

In contrast, our newly proposed BinCodex dataset provides a more comprehensive and realistic foundation. To the best of our knowledge, it encompasses the largest code base, including a diverse range of code sources such as GNU software packages, JS engines, firmware, and common libraries. Importantly, BinCodex incorporates 69 real-world vulnerable functions to cover more vulnerability patterns when facilitating the vulnerability searching scenario. Furthermore, it includes full-scale transformations such as non-default options and inter-procedural code obfuscation techniques, which have been neglected in other datasets but have been proven effective both in our evaluation and other works [39,40].

By incorporating these diverse factors, BinCodex offers valuable insights for BCSD techniques to learn from and evaluate the resilience of binary diffing methods against a wide range of compilation optimization variants and code obfuscation techniques. As mentioned in the introduction section, we are committed to contributing to the research community by making our dataset and trained models openly available. This will enable other researchers to utilize and build upon our work, fostering collaboration and further advancements in the field of binary code similarity and vulnerability analysis.

6. Discussion

Cross-platform consideration. As shown in Table 1, all current BCSD tools have their implementation in the x86 platform, thus the current dataset construction process primarily focuses the dataset specifically on evaluating BCSD techniques on the x86. Future work aims to incorporate cross-platform binaries to assess the accuracy of BCSD techniques across different architectures. This will require addressing the unique characteristics of each instruction set to ensure fair and meaningful comparisons. Including cross-platform binaries in the dataset is a valuable direction for future research.

Existing obfuscators. Aside from obfuscation techniques, we found that existing obfuscators have limitations in their implementation. In O-LLVM [35], In the process of implementing the SUB method, we found that the substituted instructions are usually optimized back to the original instructions, which would lose the obfuscation effect. To this end, we additional add strategies to prevent the de-obfuscation effect, including basic block splitting, adding reference or inline assemble `nop` instructions in the middle, etc. Besides, BCF and `FLA` skip the exception-relevant functions.

As for the inter-procedural obfuscation techniques, after it separates and aggregates these features, the searching difficulty increases, and the searching accuracy decreases. From our conclusion in table Table 3, the lack of call-graph consideration makes them unable to adopt inter-procedural obfuscation. We believe our study will raise awareness of inter-procedural obfuscation on binary diffing.

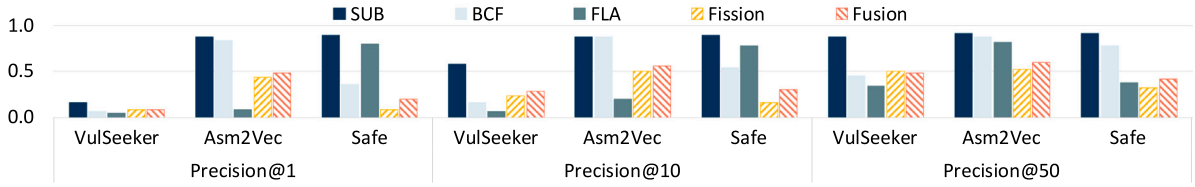

Fig. 8. Precision ratio for top@1/10/50 of vulnerable functions. Higher means stronger vulnerable function searching ability.

Table 5
Vulnerable code detail in BinCodex.

Program	CVE	Funtion	Program	CVE	funtion
JerryScript	CVE-2020–13991	opfunc_spread_arguments	libcurl	CVE-2016–5421	close_all_connections
QuickJS	CVE-2020–22876	compute_stack_size_rec		CVE-2016–7167	curl_easy_unescape
BusyBox1.33.1	CVE-2021–42378	getvar_i_int		CVE-2016–8615	get_line
	CVE-2021–42380	clrvar		CVE-2016–8616	ConnectionExists
	CVE-2021–42381	hash_init		CVE-2016–8617	Curl_base64_encode
	CVE-2021–42382	getvar_s		CVE-2016–8618	alloc_addbyter
	CVE-2021–42379	next_input_file		CVE-2016–8621	parsedate
	CVE-2021–42384	handle_special		CVE-2016–8622	unescape_word
	CVE-2021–42386	nvalloc		CVE-2016–8623	Curl_cookie_getlist
	CVE-2021–42383	evaluate		CVE-2016–8624	parseurlandfillconn
	CVE-2021–42385	evaluate		CVE-2016–8625	curl_version
OpenSSL 1.1.1	CVE-2022–0778	BN_mod_sqrt		CVE-2016–9586	dprintf_formatf
	CVE-2021–3712	EC_GROUP_new_from_ecparameters		CVE-2017–1000100	tftp_send_first
	CVE-2021–3711	sm2_plaintext_size		CVE-2017–1000254	ftp_statemach_act
	CVE-2021–3450	check_chain_extensions		CVE-2017–1000257	imap_state_fetch_resp
	CVE-2021–3449	init_sig_algs		CVE-2017–8817	setcharset
	CVE-2020–1971	GENERAL_NAME_dup		CVE-2018–1000007	Curl_http_output_auth
	CVE-2020–1967	tls1_check_sig_alg		CVE-2018–1000120	ftp_state_list
	CVE-2019–1563	cms_RecipientInfo_ktri_decrypt		CVE-2018–1000120	ftp_done
	CVE-2019–1547	EC_GROUP_set_generator		CVE-2018–1000120	ftp_parse_url_path
	CVE-2019–1543	chacha_init_key		CVE-2018–1000122	readwrite_data
	CVE-2018–0734	dsa_sign_setup		CVE-2018–1000301	Curl_http_readwrite_headers
	CVE-2018–0735	ec_scalar_mul_ladder		CVE-2019–5436	tftp_connect
libcurl	CVE-2014–0138	ConnectionExists		CVE-2019–5482	tftp_connect
	CVE-2014–3613	Curl_cookie_add		CVE-2020–8231	conn_is_conn
	CVE-2014–3620	Curl_cookie_add		CVE-2020–8231	curl_easy_duphandle
	CVE-2014–3707	FormAdd		CVE-2020–8231	curl_multi_add_handle
	CVE-2014–8150	parseurlandfillconn		CVE-2020–8231	Curl_open
	CVE-2015–3143	ConnectionExists		CVE-2020–8231	curl_multi_remove_handle
	CVE-2015–3145	sanitize_cookie_path		CVE-2020–8285	init_wc_data
	CVE-2015–3148	Curl_http_done		CVE-2021–22876	Curl_follow
	CVE-2015–3153	Curl_init_userdefined		CVE-2021–22898	suboption
	CVE-2016–0755	ConnectionExists		CVE-2021–22924	create_conn
	CVE-2016–5419	Curl_clone_ssl_config		CVE-2021–22925	suboption
	CVE-2016–5420	Curl_ssl_config_matches			

Advancing BCSD. Reducing the cost of diffing in binary code similarity detection is a challenge, particularly when dealing with smaller diffing granularity. Context information can be leveraged to narrow down the searching space and mitigate the associated costs. While previous works have predominantly focused on capturing and encoding control flow information, data flow has received less attention from the BCSD perspective. However, considering that data flow is harder to change during obfuscation, there is untapped potential in exploring data flow representation for improved BCSD techniques.

Furthermore, as observed in Section 4, larger binary sizes can lead to decreased diffing accuracy. To address this, one approach is to merge functions into groups, effectively reducing the searching space. An example of such an approach is FirmUp [61], which emphasizes optimizing the searching process rather than solely focusing on feature generation. This highlights the potential for further optimization in the searching process to enhance the accuracy of BCSD techniques.

7. Conclusion

We present a paper that addresses the challenges in the binary code similarity detection dataset and introduces the implementation and evaluation of a novel dataset called BinCodex. The primary objective of the dataset is to provide a standardized framework for evaluating BCSD techniques. To ensure the dataset's effectiveness, several factors are carefully considered during its construction, including diverse code types, a wide range of code samples, incorporating multiple aspects of binary change points, and different levels of workloads in similarity detection tasks. The implementation of BinCodex on a Linux system enables the evaluation of eight state-of-the-art BCSD tools. The results indicate that most BCSD works perform well when default compiler options or intra-procedural code obfuscation are used but face challenges with non-default options and inter-procedural obfuscation techniques. These findings provide valuable insights into the current state of BCSD works and highlight opportunities for future optimizations in the field.

CRediT authorship contribution statement

Peihua Zhang: Writing – original draft, Software, Methodology, Formal analysis, Conceptualization. **Chenggang Wu:** Supervision, Conceptualization. **Zhe Wang:** Writing – review & editing, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper

Appendix

See Tables 5–7.

Table 6
All compiler options in GCC, classified and incremental by the default optimization level. BinTuner uses all these options with all possible combinations.

-O1 (compared with O0)	-O2 (compared with -O1)	-O3 (compared with -O2)	Others
auto-inc-dec branch-count-reg compare-elim	align-functions align-jumps align-labels	gcse-after-reload	early-inlining-insns
combine-stack-adjustments cprop-registers	caller-saves code-hoisting crossjumping	loop-interchange	gcse-cost-distance-ratio
defer-pop delayed-branch forward-propagate	cse-follow-jumps cse-skip-blocks gcse-lm	loop-unroll-and-jam	iv-max-considered-uses
dce guess-branch-probability if-conversion	delete-null-pointer-checks devirtualize	predictive-commoning	reorder-blocks-algorithm=stc
if-conversion2 inline-functions-called-once	devirtualize-speculatively finite-loops	tree-partial-pre	prefetch-loop-arrays
tree-ter ipa-pure-const align-loops tree-ch ipa-reference-addressable merge-constants	gcse inline-functions indirect-inlining inline-small-functions ipa-bit-cp ipa-cp	tree-loop-distribution tree-loop-vectorize	
move-loop-invariants omit-frame-pointer reorder-blocks dse shrink-wrap-separate	optimize-strlen partial-inlining ipa-ra isolate-erroneous-paths-dereference	tree-slp-vectorize unswitch-loops	
split-wide-types ssa-backprop ssa-phiopt	ipa-icf reorder-blocks-algorithm=stc	vect-cost-model	
tree-bit-ccp tree-ccp tree-dce shrink-wrap	reorder-blocks-and-partition ipa-sra	vect-cost-model=dynamic	
tree-copy-prop tree-dominator-opts tree-dse	lra-remat rerun-cse-after-loop tree-vrp	version-loopsor-strides	
tree-forwprop tree-fre tree-phiprop tree-pta	sched-interblock sched-spec store-merging	split-loops split-paths	
tree-scev-cprop tree-sink tree-slsr tree-sra	thread-jumps tree-builtin-call-dce tree-pre	ipa-cp-clone	
ipa-profile unit-at-a-time ipa-reference tree-coalesce-vars	tree-switch-conversion tree-tail-merge hoist-adjacent-loads peephole2 ipa-vrp expensive-optimizations strict-aliasing schedule-insns2 optimize-sibling-calls reorder-functions schedule-insns	peel-loops	

Table 7
Top-5 non-default optimization settings generated from BinTuner [31].

-O0 -fauto-inc-dec -fforward-propagate -fcombine-stack-adjustments -fcompare-elim -fcprop-registers -fdce -fif-conversion2 -fno-delayed-branch -fdse -fno-defer-pop -fno-merge-constants -fno-guess-branch-probability -ftree-dominator-opts -finline-functions-called-once -fno-ipa-pure-const -fno-ipa-profile -fipa-reference -fbranch-count-reg -fno-move-loop-invariants -freorder-blocks -fno-shrink-wrap -fsplit-wide-types -ftree-copy-prop -ftree-bit-ccp -fno-tree-ter -fssa-backprop -fno-tree-coalesce-vars -fipa-cp-clone -ftree-forwprop -ftree-fre -fno-tree-sink -fno-tree-sra -fsplit-paths -ftree-pta -ftree-ccp -fno-ipa-cp -fno-unit-at-a-time -fno-omit-frame-pointer -ftree-phiprop -fno-tree-ch -ftree-slsr -fpeephole2 -fno-if-conversion -fno-ssa-phiopt -fno-shrink-wrap-separate -fthread-jumps -fno-align-functions -fno-align-labels -fno-align-loops -fstore-merging -fstrict-aliasing -fno-caller-saves -fno-crossjumping -fno-cse-follow-jumps -fcse-skip-blocks -fno-delete-null-pointer-checks -fno-devirtualize -fgcse -fno-gcse-lm -fno-devirtualize-speculatively -fno-expensive-optimizations -fno-hoist-adjacent-loads -finline-small-functions -findirect-inlining -ftree-vectorize -ftree-dce -fno-peel-loops -fno-isolate-erroneous-paths-dereference -fno-lra-remat -fno-optimize-sibling-calls -fno-optimize-strlen -fpartial-inlining -fno-ipa-icf -freorder-blocks-and-partition -fno-reorder-functions -fno-rerun-cse-after-loop -fsched-interblock -fno-sched-spec -ftree-dse -fipa-sra -fno-schedule-insns -fno-tree-partial-pre -fstrict-overflow -ftree-builtin-call-dce -ftree-switch-conversion -ftree-tail-merge -fno-tree-slp-vectorize -fno-tree-pre -fno-tree-vrp -fno-ipa-ra -freorder-blocks -fno-schedule-insns2 -fno-code-hoisting -fvect-cost-model -fno-ipa-bit-cp -fno-ipa-vrp -freorder-blocks-algorithm=simple -finline-functions -fno-unswitch-loops -fno-predictive-commoning -fno-gcse-after-reload -ftree-loop-vectorize –param early-inlining-insns=295 –param gcse-cost-distance-ratio=52 –param iv-max-considered-uses=661 -fno-tree-loop-distribute-patterns
-O0 -fno-auto-inc-dec -fbranch-count-reg -fno-combine-stack-adjustments -fcompare-elim -fcprop-registers -fdce -fno-defer-pop -ftree-bit-ccp -fipa-profile -fforward-propagate -fguess-branch-probability -fif-conversion2 -fif-conversion -fno-inline-functions-called-once -fno-ipa-pure-const -fdse -fno-ipa-reference -fmerge-constants -fno-move-loop-invariants -fno-reorder-blocks -fno-shrink-wrap -fno-split-wide-types -fdelayed-branch -ftree-ccp -fno-tree-ch -fno-tree-coalesce-vars -ftree-copy-prop -ftree-dce -ftree-dse -ftree-forwprop -fno-tree-fre -ftree-sink -ftree-slsr -ftree-sra -ftree-pta -fno-tree-ter -fno-unit-at-a-time -fno-omit-frame-pointer -fcse-skip-blocks -ftree-dominator-opts -fssa-backprop -fno-align-functions -fno-tree-phiprop -fno-ssa-phiopt -fno-shrink-wrap-separate -fno-thread-jumps -fno-align-labels -fno-align-loops -fcrossjumping -fno-ipa-sra -fno-caller-saves -fno-cse-follow-jumps -fno-delete-null-pointer-checks -fdevirtualize -fno-devirtualize-speculatively -fno-expensive-optimizations -fno-gcse -fgcse-lm -fhoist-adjacent-loads -finline-small-functions -ftree-pre -fno-ipa-cp -fipa-icf -fno-reorder-blocks-and-partition -fpeel-loops -findirect-inlining -fno-isolate-erroneous-paths-dereference -flra-remat -fno-optimize-sibling-calls -foptimize-strlen -fpartial-inlining -fsched-spec -fno-tree-tail-merg -freorder-functions -frerun-cse-after-loop -fsched-interblock -fschedule-insns -fstrict-aliasing -fno-peephole2 -fstore-merging -fipa-ra -fno-tree-builtin-call-dce -fno-tree-switch-conversion -fno-strict-overflow -fno-tree-vrp -fno-reorder-blocks -fno-schedule-insns2 -fipa-vrp -fno-code-hoisting -freorder-blocks-algorithm=simple -fno-ipa-bit-cp -fno-inline-functions -funswitch-loops -fno-gcse-after-reload -fno-split-paths -fno-tree-partial-pre -ftree-slp-vectorize -ftree-loop-vectorize -ftree-loop-distribute-patterns -ftree-vectorize -fno-vect-cost-model -fipa-cp-clone -fno-predictive-commoning –param early-inlining-insns=56 –param gcse-cost-distance-ratio=85 –param iv-max-considered-uses=874
-O0 -fno-auto-inc-dec -fno-branch-count-reg -fno-combine-stack-adjustments -fcompare-elim -fno-cprop-registers -fno-dce -fno-defer-pop -fdse -fno-delayed-branch -fno-forward-propagate -fno-guess-branch-probability -fno-move-loop-invariants -fipa-profile -finline-functions-called-once -fipa-pure-const -fif-conversion -fno-ipa-reference -fmerge-constants -fno-if-conversion2 -fno-reorder-blocks -fshrink-wrap -fno-split-wide-types -fno-align-loops -fno-ssa-phiopt -ftree-pta -funit-at-a-time -fno-omit-frame-pointer -fno-tree-phiprop -fno-tree-dominator-opts -fno-ssa-backprop -fno-devirtualize -ftree-slsr -fshrink-wrap-separate -fno-ipa-cp -falign-functions -fno-align-labels -fno-caller-saves -fno-thread-jumps -fno-ipa-sra

(*continued on next page*)

Table 7 (*continued*).

-fcrossjumping -ftree-sra -fcse-follow-jumps -fno-cse-skip-blocks -fdelete-null-pointer-checks -fno-devirtualize-speculatively -fno-gcse -fgcse-lm -fexpensive-optimizations -fno-hoist-adjacent-loads -fno-inline-small-functions -fpartial-inlining -fno-ipa-icf -fisolate-erroneous-paths-dereference -fno-optimize-sibling-calls -ftree-pre -fno-sched-spec -fno-optimize-strlen -fno-indirect-inlining -fno-peephole2 -fno-reorder-blocks-and-partition -fno-code-hoisting -ftree-ccp -ftree-ch -ftree-coalesce-vars -ftree-copy-prop -ftree-dce -ftree-dse -ftree-forwprop -ftree-fre -ftree-sink -fno-tree-ter -fstore-merging -freorder-blocks-algorithm=simple -fno-ipa-bit-cp -fno-ipa-vrp -finline-functions -fno-tree-partial-pre -fno-predictive-commoning -fno-tree-vectorize -fpeel-loops -fsplit-paths -fgcse-after-reload -fno-tree-loop-vectorize -fno-tree-loop-distribute-patterns -fno-tree-slp-vectorize -fno-lra-remat -fno-rerun-cse-after-loop -freorder-functions -fno-strict-aliasing -fno-vect-cost-model -ftree-switch-conversion -fno-strict-overflow -ftree-builtin-call-dce -fno-tree-tail-merge -ftree-vrp -fno-ipa-ra -fno-schedule-insns -fno-schedule-insns2 -fsched-interblock -fno-reorder-blocks -fno-ipa-cp-clone -funswitch-loops -fno-tree-bit-ccp –param early-inlining-insns=846 –param gcse-cost-distance-ratio=14
-O3 -fno-auto-inc-dec -fno-branch-count-reg -fno-combine-stack-adjustments -fcompare-elim -fcprop-registers -fno-dce -fno-defer-pop -fno-dse -fdelayed-branch -fforward-propagate -fguess-branch-probability -fif-conversion2 -fif-conversion -fno-inline-functions-called-once -fipa-profile -fno-merge-constants -fno-ipa-reference -fno-tree-coalesce-vars -fmove-loop-invariants -fno-tree-copy-prop -fshrink-wrap -fno-split-wide-types -fno-tree-slsr -freorder-blocks -ftree-bit-ccp -ftree-ccp -fno-tree-pta -fno-tree-dce -ftree-dse -ftree-forwprop -ftree-fre -fno-tree-sink -fno-tree-ch -falign-labels -ftree-sra -fno-tree-ter -fomit-frame-pointer -ftree-phiprop -fno-tree-dominator-opts -fssa-backprop -fno-ssa-phiopt -fthread-jumps -funit-at-a-time -fno-ipa-pure-const -fshrink-wrap-separate -falign-functions -fno-align-loops -fcaller-saves -fno-crossjumping -fcse-follow-jumps -fcse-skip-blocks -fdelete-null-pointer-checks -fdevirtualize -fno-devirtualize-speculatively -fno-expensive-optimizations -fno-gcse -fno-gcse-lm -fhoist-adjacent-loads -finline-small-functions -findirect-inlining -fno-ipa-cp -fno-ipa-icf -fno-isolate-erroneous-paths-dereference -fno-lra-remat -fipa-sra -fno-optimize-sibling-calls -foptimize-strlen -fno-partial-inlining -fschedule-insns -freorder-blocks-and-partition -fno-reorder-functions -ftree-switch-conversion -frerun-cse-after-loop -fno-sched-interblock -fsched-spec -fno-strict-aliasing -fno-strict-overflow -ftree-builtin-call-dce -fno-peephole2 -ftree-tail-merge -fno-tree-pre -fipa-bit-cp -fno-ipa-ra -freorder-blocks -fschedule-insns2 -fno-code-hoisting -fno-store-merging -ftree-vrp -freorder-blocks-algorithm=simple -fipa-vrp -fno-inline-functions -fno-predictive-commoning -fgcse-after-reload -ftree-loop-vectorize -fipa-cp-clone -ftree-loop-distribute-patterns -fno-tree-slp-vectorize -fvect-cost-model -ftree-partial-pre -fpeel-loops -ftree-vectorize -fsplit-paths -funswitch-loops –param early-inlining-insns=482 –param gcse-cost-distance-ratio=59 –param iv-max-considered-uses=105
-O0 -fno-auto-inc-dec -fno-branch-count-reg -fno-combine-stack-adjustments -fcompare-elim -fcprop-registers -ftree-forwprop -fdelayed-branch -fipa-profile -fforward-propagate -fguess-branch-probability -fno-if-conversion2 -fif-conversion -fno-inline-functions-called-once -fipa-pure-const -fno-dce -fipa-reference -fmerge-constants -fmove-loop-invariants -fno-reorder-blocks -fshrink-wrap -fno-split-wide-types -ftree-bit-ccp -ftree-ccp -fno-tree-pta -fno-tree-coalesce-vars -fno-tree-copy-prop -fno-tree-dce -ftree-dse -ftree-slsr -fcrossjumping -fcaller-saves -fno-tree-sra -ftree-sink -fdse -fno-tree-ch -fthread-jumps -fno-unit-at-a-time -fomit-frame-pointer -ftree-phiprop -ftree-dominator-opts -fno-ssa-backprop -fno-ssa-phiopt -fshrink-wrap-separate -fno-gcse -fno-align-functions -falign-labels -fthread-jumps -fno-peephole2 -fno-tree-fre -fcse-follow-jumps -fno-defer-pop -fno-align-loops -fno-hoist-adjacent-loads -fdelete-null-pointer-checks -fno-devirtualize -fdevirtualize-speculatively -fno-expensive-optimizations -fno-cse-skip-blocks -fgcse-lm -finline-small-functions -fno-indirect-inlining -fipa-cp -fno-ipa-sra -fipa-icf -fisolate-erroneous-paths-dereference -fno-lra-remat -fno-optimize-sibling-calls -foptimize-strlen -fno-partial-inlining -fschedule-insns -freorder-blocks-and-partition -fipa-vrp -ftree-ter -fno-sched-interblock -fno-rerun-cse-after-loop -fno-reorder-functions -fno-sched-spec -fno-strict-aliasing -fno-tree-vrp -fno-tree-builtin-call-dce -ftree-switch-conversion -fstrict-overflow -ftree-tail-merge -ftree-pre -fno-reorder-blocks -fschedule-insns2 -fno-code-hoisting -fno-peel-loops -fipa-cp-clone -freorder-blocks-algorithm=simple -fno-ipa-bit-cp -finline-functions -funswitch-loops -fpredictive-commoning -fno-store-merging -fno-gcse-after-reload -fno-tree-loop-vectorize -fno-tree-loop-distribute-patterns -fno-tree-slp-vectorize -fno-vect-cost-model -ftree-partial-pre -fipa-ra -fno-split-paths -ftree-vectorize –param early-inlining-insns=798 –param gcse-cost-distance-ratio=46 –param iv-max-considered-uses=617

References

[1] statista, Number of connected IoT devices worldwide, 2020, https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/.

[2] S. Cesare, Y. Xiang, W. Zhou, Control flow-based malware variant detection, IEEE Trans. Dependable Secure Comput. 11 (4) (2013) 307–317, http://dx.doi.org/10.1109/TDSC.2013.40.

[3] Y. David, N. Partush, E. Yahav, Statistical similarity of binaries, ACM SIGPLAN Not. 51 (6) (2016) 266–280, http://dx.doi.org/10.1145/2908080.2908126.

[4] Y. Hu, Y. Zhang, J. Li, D. Gu, Cross-architecture binary semantics understanding via similar code comparison, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER, vol. 1, IEEE, 2016, pp. 57–67, http://dx.doi.org/10.1109/SANER.2016.50.

[5] T. Blazytko, M. Contag, C. Aschermann, T. Holz, Syntia: Synthesizing the semantics of obfuscated code, in: 26th USENIX Security Symposium (USENIX Security 17), 2017, pp. 643–659.

[6] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C.Y. Cho, H.B.K. Tan, Bingo: Cross-architecture cross-os binary search, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 678–689, http://dx.doi.org/10.1145/2950290.2950350.

[7] Y. Hu, Y. Zhang, J. Li, D. Gu, Binary code clone detection across architectures and compiling configurations, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension, ICPC, IEEE, 2017, pp. 88–98, http://dx.doi.org/10.1109/ICPC.2017.22.

[8] Y. David, N. Partush, E. Yahav, Similarity of binaries through re-optimization, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017, pp. 79–94, http://dx.doi.org/10.1145/3062341.3062387.

[9] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, D. Song, 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, 2017, pp. 363–376, http://dx.doi.org/10.1145/3133956.3134018.

[10] Y. Duan, X. Li, J. Wang, H. Yin, Deepbindiff: Learning program-wide code representations for binary diffing, in: Network and Distributed System Security Symposium, 2020, http://dx.doi.org/10.14722/ndss.2020.24311.

[11] M. Bourquin, A. King, E. Robbins, Binslayer: accurate comparison of binary executables, in: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, 2013, pp. 1–10, http://dx.doi.org/10.1145/2430553.2430557.

[12] S.H. Ding, B.C. Fung, P. Charland, Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization, in: 2019 IEEE Symposium on Security and Privacy, SP, IEEE, 2019, pp. 472–489, http://dx.doi.org/10.1109/SP.2019.00003.

[13] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, Z. Zhang, Neural machine translation inspired binary code similarity comparison beyond function pairs, in: NDSS, The Internet Society, 2019, http://dx.doi.org/10.14722/ndss.2019.23492.

[14] J. Gao, X. Yang, Y. Fu, Y. Jiang, J. Sun, VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary, in: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2018, pp. 896–899, http://dx.doi.org/10.1145/3238147.3240480.

[15] J. Pewny, F. Schuster, L. Bernhard, T. Holz, C. Rossow, Leveraging semantic signatures for bug search in binary programs, in: Proceedings of the 30th Annual Computer Security Applications Conference, 2014, pp. 406–415, http://dx.doi.org/10.1145/2664243.2664269.

[16] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, T. Holz, Cross-architecture bug search in binary executables, in: 2015 IEEE Symposium on Security and Privacy, IEEE, 2015, pp. 709–724, http://dx.doi.org/10.1109/SP.2015.49.

[17] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, T. Liu, Patch based vulnerability matching for binary programs, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 376–387, http://dx.doi.org/10.1145/3395363.3397361.

[18] L. Zhao, Y. Zhu, J. Ming, Y. Zhang, H. Zhang, H. Yin, Patchscope: Memory object centric patch diffing, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 149–165, http://dx.doi.org/10.1145/3372297.3423342.

[19] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, F. Song, Spain: security patch analysis for binaries towards understanding the pain and pills, in: 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE, IEEE, 2017, pp. 462–472, http://dx.doi.org/10.1109/ICSE.2017.49.

[20] X. Hu, K.G. Shin, S. Bhatkar, K. Griffin, {MutantX-S}: Scalable malware clustering based on static features, in: 2013 USENIX Annual Technical Conference (USENIX ATC 13), 2013, pp. 187–198.

[21] J. Jang, D. Brumley, S. Venkataraman, Bitshred: feature hashing malware for scalable triage and semantic analysis, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, 2011, pp. 309–320, http://dx.doi.org/10.1145/2046707.2046742.

[22] D. Xu, J. Ming, D. Wu, Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping, in: 2017 IEEE Symposium on Security and Privacy, SP, IEEE, 2017, pp. 921–937, http://dx.doi.org/10.1109/SP.2017.56.

[23] A. Caliskan, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, A. Narayanan, When coding style survives compilation: De-anonymizing programmers from executable binaries, 2015, http://dx.doi.org/10.48550/arXiv.1512.08546, arXiv preprint arXiv:1512.08546.

[24] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, P. Narasimhan, Binary function clustering using semantic hashes, in: 2012 11th International Conference on Machine Learning and Applications, 1, IEEE, 2012, pp. 386–391, http://dx.doi.org/10.1109/ICMLA.2012.70.

[25] Y. Xue, Z. Xu, M. Chandramohan, Y. Liu, Accurate and scalable cross-architecture cross-os binary code search with emulation, IEEE Trans. Softw. Eng. 45 (11) (2018) 1125–1149, http://dx.doi.org/10.1109/TSE.2018.2827379.

[26] H. Wang, P. Ma, Y. Yuan, Z. Liu, S. Wang, Q. Tang, S. Nie, S. Wu, Enhancing DNN-based binary code function search with low-cost equivalence checking, IEEE Trans. Softw. Eng. (2022) http://dx.doi.org/10.1109/TSE.2022.3149240.

[27] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, W. Zou, αDiff: cross-version binary code similarity detection with dnn, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 667–678, http://dx.doi.org/10.1145/3238147.3238199.

[28] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, C. Zhang, Jtrans: Jump-aware transformer for binary code similarity detection, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, in: ISSTA 2022, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1–13, http://dx.doi.org/10.1145/3533767.3534367.

[29] X. Xu, S. Feng, Y. Ye, G. Shen, Z. Su, S. Cheng, G. Tao, Q. Shi, Z. Zhang, X. Zhang, 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, in: ISSTA 2023, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1106–1118, http://dx.doi.org/10.1145/3597926.3598121.

[30] N. Shalev, N. Partush, Binary similarity detection using machine learning, in: Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 42–47, http://dx.doi.org/10.1145/3264820.3264821.

[31] X. Ren, M. Ho, J. Ming, Y. Lei, L. Li, Unleashing the hidden power of compiler optimization on binary code difference: An empirical study, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021, pp. 142–157, http://dx.doi.org/10.1145/3453483.3454035.

[32] C. Linn, S. Debray, Obfuscation of executable code to improve resistance to static disassembly, in: Proceedings of the 10th ACM Conference on Computer and Communications Security, 2003, pp. 290–299, http://dx.doi.org/10.1145/948109.948149.

[33] C. Collberg, S. Martin, J. Myers, J. Nagra, Distributed application tamper detection via continuous software updates, in: Proceedings of the 28th Annual Computer Security Applications Conference, 2012, pp. 319–328, http://dx.doi.org/10.1145/2420950.2420997.

[34] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, A. Pretschner, Code obfuscation against symbolic execution attacks, in: Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016, pp. 189–200, http://dx.doi.org/10.1145/2991079.2991114.

[35] P. Junod, J. Rinaldini, J. Wehrli, J. Michielin, Obfuscator-LLVM–software protection for the masses, in: 2015 IEEE/ACM 1st International Workshop on Software Protection, IEEE, 2015, pp. 3–9, http://dx.doi.org/10.1109/SPRO.2015.10.

[36] H. Xu, Y. Zhou, Y. Kang, F. Tu, M. Lyu, Manufacturing resilient bi-opaque predicates against symbolic execution, in: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, IEEE, 2018, pp. 666–677, http://dx.doi.org/10.1109/DSN.2018.00073.

[37] M. Hammad, J. Garcia, S. Malek, A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 421–431, http://dx.doi.org/10.1145/3180155.3180228.

[38] H. Wang, S. Wang, D. Xu, X. Zhang, X. Liu, Generating effective software obfuscation sequences with reinforcement learning, IEEE Trans. Dependable Secure Comput. (2020) http://dx.doi.org/10.1109/TDSC.2020.3041655.

[39] P. Zhang, C. Wu, M. Peng, K. Zeng, D. Yu, Y. Lai, Y. Kang, W. Wang, Z. Wang, Khaos: The impact of inter-procedural code obfuscation on binary diffing techniques, in: Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization, 2023, pp. 55–67.

[40] H. Geng, M. Zhong, P. Zhang, F. Lv, X. Feng, OPTango: Multi-central representation learning against innumerable compiler optimization for binary diffing, in: 2023 IEEE 34th International Symposium on Software Reliability Engineering, ISSRE, IEEE, 2023, pp. 774–785.

[41] L. Massarelli, G.A.D. Luna, F. Petroni, R. Baldoni, L. Querzoni, Safe: Self-attentive function embeddings for binary similarity, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2019, pp. 309–329, http://dx.doi.org/10.1007/978-3-030-22038-9_15.

[42] zynamics GmbH and Google LLC, BinDiff manual, 2022, http://www.zynamics.com/bindiff/manual/index.html.

[43] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, H. Yin, Scalable graph-based bug search for firmware images, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 480–491, http://dx.doi.org/10.1145/2976749.2978370.

[44] L. Luo, J. Ming, D. Wu, P. Liu, S. Zhu, Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 389–400, http://dx.doi.org/10.1145/2635868.2635900.

[45] J. Ming, D. Xu, Y. Jiang, D. Wu, {BinSim}: Trace-based semantic binary diffing via system call sliced segment equivalence checking, in: 26th USENIX Security Symposium (USENIX Security 17), 2017, pp. 253–270.

[46] S. Wang, D. Wu, In-memory fuzzing for binary code similarity analysis, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2017, pp. 319–330, http://dx.doi.org/10.1109/ASE.2017.8115645.

[47] C. Nachenberg, Computer virus-antivirus coevolution, Commun. ACM 40 (1) (1997) 46–51, http://dx.doi.org/10.1145/242857.242869.

[48] K.A. Roundy, B.P. Miller, Binary-code obfuscations in prevalent packer tools, ACM Comput. Surv. 46 (1) (2013) 1–32, http://dx.doi.org/10.1145/2522968.2522972.

[49] M.F.X.J. Oberhumer, L. Molnár, J.F. Reiser, The ultimate packer for eXecutables, 2022, https://upx.github.io/.

[50] Oreans Technologies, Themida overview, 2022, https://www.oreans.com/themida.php.

[51] Z. Tang, K. Kuang, L. Wang, C. Xue, X. Gong, X. Chen, D. Fang, J. Liu, Z. Wang, SEEAD: A semantic-based approach for automatic binary code de-obfuscation, in: 2017 IEEE Trustcom/BigDataSE/ICESS, 2017, pp. 261–268, http://dx.doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.246.

[52] M. Egele, T. Scholte, E. Kirda, C. Kruegel, A survey on automated dynamic malware-analysis techniques and tools, ACM Comput. Surv. (CSUR) 44 (2) (2008) 1–42, http://dx.doi.org/10.1145/2089125.2089126.

[53] A. Dinaburg, P. Royal, M. Sharif, W. Lee, Ether: malware analysis via hardware virtualization extensions, in: Proceedings of the 15th ACM Conference on Computer and Communications Security, 2008, pp. 51–62, http://dx.doi.org/10.1145/1455770.1455779.

[54] M. Sharif, A. Lanzi, J. Giffin, W. Lee, Automatic reverse engineering of malware emulators, in: 2009 30th IEEE Symposium on Security and Privacy, IEEE, 2009, pp. 94–109, http://dx.doi.org/10.1109/SP.2009.27.

[55] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, P.G. Bringas, SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers, in: 2015 IEEE Symposium on Security and Privacy, IEEE, 2015, pp. 659–673, http://dx.doi.org/10.1109/SP.2015.46.

[56] H. Wang, P. Ma, S. Wang, Q. Tang, S. Nie, S. Wu, Sem2vec: Semantics-aware assembly tracelet embedding, ACM Trans. Softw. Eng. Methodol. 32 (4) (2023) http://dx.doi.org/10.1145/3569933.

[57] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, K. Lu, VulHawk: Cross-architecture vulnerability detection with entropy-based binary code search, in: NDSS, 2023.

[58] D. Kim, E. Kim, S.K. Cha, S. Son, Y. Kim, Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned, IEEE Trans. Softw. Eng. (2022) 1–23, http://dx.doi.org/10.1109/TSE.2022.3187689.

[59] X. Xu, Q. Zheng, Z. Yan, M. Fan, A. Jia, T. Liu, Interpretation-enabled software reuse detection based on a multi-level birthmark model, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE, IEEE, 2021, pp. 873–884, http://dx.doi.org/10.1109/ICSE43902.2021.00084.

[60] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, Z. Shi, Asteria: Deep learning-based AST-encoding for cross-platform binary code similarity detection, in: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, IEEE, 2021, pp. 224–236.

[61] Y. David, N. Partush, E. Yahav, Firmup: Precise static detection of common vulnerabilities in firmware, ACM SIGPLAN Not. 53 (2) (2018) 392–404, http://dx.doi.org/10.1145/3173162.3177157.

[62] H. Huang, A.M. Youssef, M. Debbabi, Binsequence: Fast, accurate and scalable binary code reuse detection, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, 2017, pp. 155–166, http://dx.doi.org/10.1145/3052973.3052974.

[63] M. Ollivier, S. Bardin, R. Bonichon, J.-Y. Marion, How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections), in: Proceedings of the 35th Annual Computer Security Applications Conference, 2019, pp. 177–189, http://dx.doi.org/10.1145/3359789.3359812.

[64] S. Eschweiler, K. Yakdan, E. Gerhards-Padilla, discovRE: Efficient cross-architecture identification of bugs in binary code, in: NDSS, vol. 52, 2016, pp. 58–79, http://dx.doi.org/10.14722/ndss.2016.23185.

[65] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, Fossil: a resilient and efficient system for identifying foss functions in malware binaries, ACM Trans. Priv. Secur. 21 (2) (2018) 1–34, http://dx.doi.org/10.1145/3175492.

[66] Y. David, E. Yahav, Tracelet-based code search in executables, ACM SIGPLAN Not. 49 (6) (2014) 349–360, http://dx.doi.org/10.1145/2594291.2594343.

[67] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, H. Yin, Extracting conditional formulas for cross-platform bug search, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, 2017, pp. 346–359, http://dx.doi.org/10.1145/3052973.3052995.

[68] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, D. Gu, Binmatch: A semantics-based hybrid approach on binary code clone analysis, in: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2018, pp. 104–114, http://dx.doi.org/10.1109/ICSME.2018.00019.

[69] Y. Liu, H. Wang, Tracking mirai variants, Virus Bull. (2018) 1–18.

[70] D. Kim, E. Kim, S.K. Cha, S. Son, Y. Kim, Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned, IEEE Trans. Softw. Eng. (2022).