A Large Scale Study of AI-based Binary Function Similarity Detection Techniques for Security Researchers and Practitioners

Jingyi Shi[1,2 †‡], Yufeng Chen[1,2 †‡], Yang Xiao[1,2 *†‡], Yuekang Li[3], Zhengzi Xu[4], Sihao Qiu[1,2 †‡], Chi Zhang[1,2 †‡], Keyu Qi[1,2 †‡], Yeting Li[1,2 †‡], Xingchu Chen[1,2 †‡], Yanyan Zou[1,2 †‡], Yang Liu[5], Wei Huo[1,2 *†‡]
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3]University of New South Wales, Sydney, Australia
[4]Imperial College London, Imperial Global Singapore, Singapore
[5]Nanyang Technological University, Singapore
{shijingyi, chenyufeng, xiaoyang, qiusihao, zhangchi2024, qikeyu, liyeting, chenxingchu, zouyanyan, huowei}@iie.ac.cn
yuekang.li@unsw.edu.au, z.xu@imperial.ac.uk, yangliu@ntu.edu.sg

***Abstract*—Binary Function Similarity Detection (BFSD) is a foundational technique in software security, underpinning a wide range of applications including vulnerability detection, malware analysis. Recent advances in AI-based BFSD tools have led to significant performance improvements. However, existing evaluations of these tools suffer from three key limitations: a lack of in-depth analysis of performance-influencing factors, an absence of realistic application analysis, and reliance on small-scale or low-quality datasets.**

In this paper, we present the first large-scale empirical study of AI-based BFSD tools to address these gaps. We construct two high-quality and diverse datasets: BINATLAS, comprising 12,453 binaries and over 7 million functions for capability evaluation; and BINARES, containing 12,291 binaries and 54 real-world 1-day vulnerabilities for evaluating vulnerability detection performance in practical IoT firmware settings. Using these datasets, we evaluate nine representative BFSD tools, analyze the challenges and limitations of existing BFSD tools, and investigate the consistency among BFSD tools. We also propose an actionable strategy for combining BFSD tools to enhance overall performance (an improvement of 13.4%). Our study not only advances the practical adoption of BFSD tools but also provides valuable resources and insights to guide future research in scalable and automated binary similarity detection.

***Index Terms*—Binary Function Similarity Detection, Artificial Intelligence, In-practice Strategy, Dataset and Evaluation**

I. INTRODUCTION

Binary Function Similarity Detection (BFSD) aims to quantify the similarity between binary functions and has diverse applications in software security, including vulnerability detection [1]–[11], malware identification [12]–[14], software composition analysis [15]–[17], and software plagiarism detection [18], [19]. BFSD serves as a foundational technology in these domains. For instance, in vulnerability detection scenarios, users employ BFSD tools to compare functions within target binary programs against known vulnerable functions, thereby identifying 1-day vulnerabilities. Given its extensive usage, studying and improving BFSD techniques is of significant importance.

Recently, AI-based BFSD tools [4]–[6], [20]–[23] have demonstrated superior performance by leveraging various models to analyze different representations of binary functions, outperforming traditional BFSD methods [1], [2], [24]–[26].

To systematically evaluate the performance of AI-based BFSD tools, several studies have been conducted [10], [27]. However, they exhibit three key limitations. First, they lack fine-grained analyses of factors that influence tool performance. In real-world scenarios, key factors such as function inlining and function pool size can significantly affect effectiveness. Although previous studies evaluated BFSD tools under several conditions, yet the extent and nature of these impacts remain unclear. A deeper analysis is essential to reveal tool-specific limitations and common challenges. Second, these studies often overlook practical usage scenarios. In practice, users may combine multiple tools to enhance performance, but the potential and rationale of such combinations have not been explored. Finally, prior evaluations rely on small-scale datasets with quality issues—including biased project selection, improper compilation settings, and flawed labeling—which introduce systematic bias and limit the generalizability of their conclusions.

In this paper, we address the aforementioned gaps through a large-scale empirical study. To ensure comprehensiveness, we construct two extensive datasets: BINATLAS and BINARES. BINATLAS is designed to evaluate BFSD tools under diverse

*Corresponding author.
†Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing, China
‡Beijing Key Laboratory of Network Security and Protection Technology, Beijing, China

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00093

and realistic conditions. It captures broad real-world variability, comprising 12,453 binaries and 7,339,256 functions compiled from popular projects spanning six categories across 320 distinct compilation configurations. We ensure its correctness through metadata verification and labeling based on debug information. BINARES is tailored for vulnerability detection experiments. It includes 12,291 binaries and 3,676,923 functions extracted from 58 IoT firmware images. The dataset features 54 known 1-day vulnerabilities as queries and 1,442 manually identified homologous functions as ground truths. Leveraging these large-scale datasets, our study aims to uncover the limitations and challenges of existing BFSD tools, and to **gain insights into practical strategies that remain effective as new tools continue to emerge**.

Specifically, we aim to address the following research questions (RQs):

- ***RQ1:*** *How do different factors impact BFSD tools?*
- ***RQ2:*** *Can BFSD tools be combined to improve overall results?*
- ***RQ3:*** *How do BFSD tools and the combination strategy perform in large-scale real-world vulnerability detection?*

By addressing the proposed research questions, we gain valuable insights into the inconsistencies among BFSD tools and derive an actionable strategy to enhance their practical applicability. Specifically, we find that tools based on different representations exhibit distinct failure patterns. Building on this observation, we propose a combination strategy that achieves a 13.4% improvement over the best-performing individual tool in a large-scale real-world vulnerability detection task.

Beyond these practical insights, our study also highlights promising directions for future research. First, our findings suggest that integrating multiple representations within a single tool can further improve overall performance. Second, our evaluation identifies key challenges faced by current BFSD tools, particularly in handling inconsistencies introduced by function inlining and mitigating performance degradation in large-scale settings. Lastly, reducing the manual effort required to verify ranked candidate functions remains an open problem, pointing to the need for more automated or reliable verification mechanisms.

In summary, our contributions are as follows:

- **Datasets:** We present two high-quality and diverse datasets: BINATLAS, designed for comprehensive capability evaluation, and BINARES, a large-scale dataset tailored for real-world vulnerability detection. Significant human effort was devoted to compilation, verification, and labeling to ensure the accuracy and reliability of both datasets. We publicly release these datasets to support and advance future research in the BFSD community.
- **Large-scale Experiments:** We conduct comprehensive evaluations of nine BFSD tools and present the first in-depth investigation into their effectiveness in realistic usage scenarios.
- **Practical Strategy:** We propose an actionable strategy to improve the effectiveness of BFSD tools by combining BFSD tools, leading to a 13.4% improvement in a large-scale real-world vulnerability detection task.
- **Future Directions:** We identify key limitations of current BFSD techniques and outline promising future research directions to address the challenges.

To facilitate future research, we open-source our source code and dataset here: https://sites.google.com/view/bfsd-study.

II. BACKGROUND

A. Workflow of BFSD Tools

AI-based function-level BFSD tools transform binary functions into vector representations, casting similarity detection as a vector comparison task. A typical use case is searching for functions similar to a query within a large-scale pool. As shown in Figure 1, this process involves three stages: embedding, ranking, and result verification.

❶ **Embedding**: The goal of this stage is to convert each input function into a vector representation. BFSD tools extract features from various representations of binary functions and feed them into an AI model to generate embeddings.

❷ **Ranking**: In this stage, the similarity between the query function's embedding and each function in the pool is computed to generate a ranked candidate queue. Common similarity metrics include cosine similarity [4], [21], [28], [29] and Euclidean distance [30].

❸ **Result Verification**: In the result verification stage, ranked candidates are manually compared to the query function to identify true targets, as the list includes only similarity scores without labels or match guarantees.

B. BFSD Application-oriented Evaluation

Common applications of BFSD tools include vulnerability detection [1]–[7], [30]–[34], malware identification [35], [36], software composition analysis [15]–[17], program comprehension [37] and software plagiarism detection [18], [19]. These applications share the general workflow illustrated in Figure 1. For example, in the widely studied task of vulnerability detection, the common approach is to use a known vulnerable function as the query and search for similar functions in a large pool of binary functions extracted from numerous programs. The goal is to identify vulnerabilities that arise from code reuse or that exist in semantically similar functions.

These real-world scenarios share two main characteristics: **Partial knowledge of compilation configurations:** While certain attributes such as architecture and bitness are typically known, other important details—like the specific compiler or optimization level used—are often unavailable [38], which complicates tool performance estimation under varying settings. **Large-scale function pools:** In practical applications, function pools often contain millions of functions [39]. For example, detecting third-party library vulnerabilities in firmware extracted from hundreds of IoT devices involves analyzing a vast number of binary functions.

In practical settings, evaluating BFSD tools and identifying actionable strategies are essential for maximizing their utility. First, analyzing performance across diverse configurations

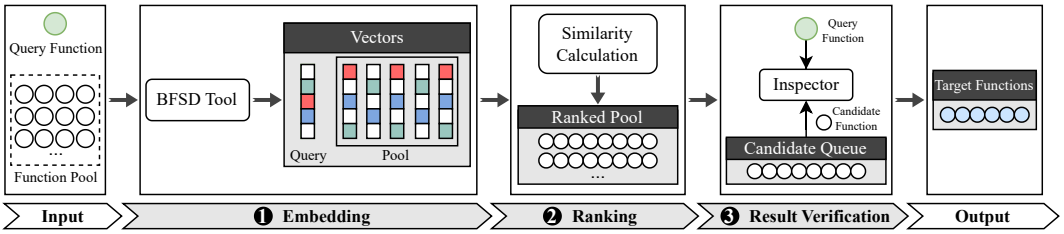

Fig. 1: Workflow of a typical BFSD application.

reveals tool-specific strengths and challenges. Second, large-scale evaluations set realistic expectations for real-world use. Finally, exploring external strategies, such as tool combinations, can uncover further performance gains. Empirical insights into these aspects are key to developing effective and practical solutions.

While previous studies [10], [27] have partially evaluated BFSD tools, to the best of our knowledge, none have systematically assessed their performance in large-scale, real-world scenarios or explored effective usage strategies. This paper fills this gap by answering three RQs, offering a comprehensive evaluation and proposing actionable combination strategies to improve real-world effectiveness.

III. Overview

A. Research Questions

In this paper, we investigate three main RQs (as shown in Figure 2):

- ***RQ1:*** *How do different factors impact BFSD tools?*
- ***RQ2:*** *Can BFSD tools be combined to improve overall results?*
- ***RQ3:*** *How do BFSD tools and the combination strategy perform in large-scale real-world vulnerability detection?*

To begin with, in RQ1, we conduct a systematic evaluation of BFSD tools under diverse real-world settings. This allows us to derive results that closely approximate actual performance, and to compare and analyze how different tools perform under various settings, hence identifying the impact of different factors. We also investigate how the key factors influence BFSD tools, both in terms of their nature and the degree of impact.

Building on the findings of RQ1, RQ2 investigates the consistency among different BFSD tools, along with the underlying causes. Motivated by the observed inconsistencies in their failure patterns, we further explore whether combining tools can lead to improved overall performance.

Finally, RQ3 evaluates BFSD tools in a large-scale, real-world vulnerability detection task, with applying the combination strategy from RQ2 to assess its practical effectiveness. The focus is on identifying homologous and vulnerable functions.

Overall, this study aims to systematically uncover the applicability and limitations of BFSD tools. Based on the insights from the RQs, we propose an effective tool combination strategy, and conclude by outlining practical guidance, key challenges, and future directions in the BFSD domain.

B. BFSD Papers and Tool Selection

BFSD methods can be broadly categorized into dynamic and static approaches. Dynamic methods [19], [34], [40]–[44] compare the execution semantics of functions through dynamic emulation or execution. These methods tend to be accurate but often lack scalability. Static methods can be further divided into two subcategories: fuzzy hashing-based methods and AI-based methods. Fuzzy hashing-based methods [2] map binary functions into fuzzy hashes and compute the similarity between hashes. AI-based approaches extract features from binary functions using raw bytes [33], assembly code [5]–[7], [20], [23], [28], [45]–[48], decompiled code [8], or attribute graphs [3], [4], [21], [22], [29], [30], [49]–[57] constructed based on dependencies and manually crafted features. These features are then encoded using graph-based or language models to generate vector representations, transforming the function similarity problem into a vector similarity computation, thereby improving efficiency. Additionally, some techniques enhance the effectiveness of BFSD tools by applying pre-processing to address inconsistencies caused by compilers [58], [59], incorporating extra context information [9], [60], re-ranking results [39], [61], [62] or adversarial training [63], [64].

TABLE I: Overview of selected BFSD tools. (Rep: Representation, CA: Cross-Architecture, #Cite: citation count, #Star: Github stars, #BL: frequency of use as baselines. A: Assembly, G: Graph, D:Decompiled code.)

Tool	Avenue	Year	Rep	Model	CA	#Cite	#Star	#BL
Gemini [4]	CCS	2017	G	Structure2Vec [65]	✓	785	135	15
GMN [22]	ICML	2019	G	GMN [22]	✓	736	304	7
Asm2Vec [5]	S&P	2019	A	PV-DM [66]	✓	487	624	13
PalmTree [23]	CCS	2021	A	BERT [67]	✗	174	141	7
SAFE [6]	TDSC	2022	A	Word2Vec [68], SANN [69]	✓	241	175	16
jTrans [28]	ISSTA	2022	A	BERT [67]	✗	97	153	6
CLAP [20]	ISSTA	2024	A	RoBERTa [70]	✗	11	54	0
HermesSim [21]	SEC	2024	G	GGNN [71]	✓	16	64	0
DeJina	-	-	D	BERT [67]	✓	-	-	-

In this paper, we limit our scope in AI-based static BFSD approaches at the function level. We exclude dynamic methods due to their limited scalability in large-scale scenarios. Furthermore, prior work [27] has demonstrated that recent AI-based BFSD tools outperform earlier fuzzy hashing techniques. Enhancement techniques that can be applied to arbitrary

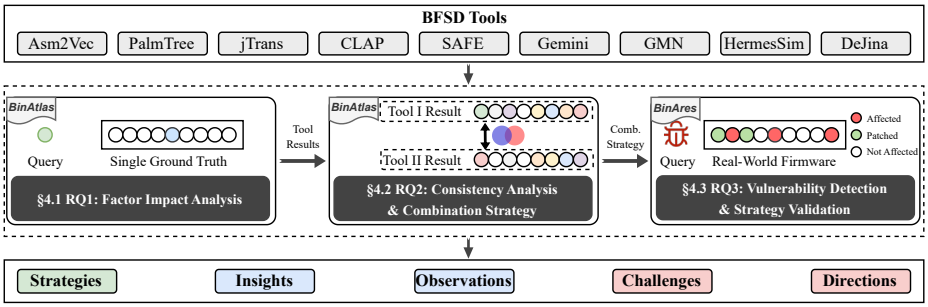

Fig. 2: Research questions in this study.

standalone BFSD tools are also excluded from our scope, as they are orthogonal to our focus.

To select BFSD tools for evaluation, we conducted a systematic literature review (SLR). We began by searching leading venues in security, software engineering, AI, and programming languages (e.g., S&P, CCS, ICSE, FSE, ICML, OOPSLA) for papers published between 2022 and 2024 using keywords such as "binary similarity" and "binary search," yielding 28 papers. After excluding 13 out-of-scope studies through manual screening, we recursively examined the references of the remaining 15 papers. This process resulted in a final set of 34 BFSD papers. We further selected *representative*, *popular*, and *diverse* BFSD tools for evaluation. First, we excluded ten papers with unavailable tools. Next, we identified six tools published before 2023, based on citation count, GitHub stars, and their use as baselines in prior work. Finally, we included two state-of-the-art (SOTA) tools from 2023 and 2024, based on their GitHub stars. The final set of tools is listed in Table I. Our selected tools encompass assembly code-based approaches and graph-based approaches, covering mainstream BFSD methodologies and ensuring diversity.

The fourth column in Table I presents the function representations used by the selected BFSD tools, all of which rely on either graph-based or assembly-level representations. With advancements in source code embedding models, these models can potentially be adapted for BFSD by fine-tuning them for decompiled code similarity detection tasks. Therefore, we fine-tune a source code embedding model, jina-embeddings-v2-base-code [72], which is derived from Jina [73], and include it as an additional baseline (DEJINA).

C. Dataset Construction

To address the three RQs posed in this study, we require two datasets: (1) a representative dataset with diverse real-world configurations to support tool evaluation and consistency analysis in RQ1 and RQ2; and (2) a large-scale, real-world vulnerability detection dataset to assess BFSD tool effectiveness in RQ3 under unknown compilation settings and to validate the strategy proposed in RQ2.

To the best of our knowledge, existing datasets [10], [27], [28], [74] do not satisfy our requirements. For the first dataset, existing public datasets suffer from limited diversity, incomplete compilation configurations, and quality issues, making them inadequate for evaluating BFSD tools across varied real-world settings and get reliable results. For the second dataset, prior vulnerability detection datasets are limited by their small scale, both in terms of the number of vulnerable query functions and the size of the candidate function pool. In addition, their labeling is often incomplete, as annotations typically cover only the top-10 results. These issues hinder reliable and generalizable evaluation. Thus, we constructed two datasets for better evaluation: BINATLAS and BINARES.

1) BINATLAS*:* BINATLAS[1] is designed to represent diverse real-world scenarios, guided by three principles: compilation variability, compositional diversity, and correctness. It includes 12,453 binaries compiled from popular projects across six categories using 320 configurations, covering five optimization levels, two compilers (two versions each), four architectures (32/64-bit), and two inlining options (enable function inlining or not). This yields 27.8 million functions, filtered to 7.3 million by excluding short functions that lack meaningful content. Specifically, we exclude functions with fewer than five basic blocks, following the settings of previous papers [24], [27], [30]. The dataset is split into training, validation, and test sets without project overlap (as shown in Table II). The projects are selected based on functionality, popularity, and project size to minimize bias and better reflect real-world scenarios. To ensure correctness, we use debug information to label functions with identical names and source positions as positive pairs, avoiding mislabeling from compiler-induced renaming. We also verify binary architecture and optimization levels using the `file` and `strings` commands to ensure compilation correctness. All binaries are stripped after extracting the necessary metadata.

2) BINARES*:* BINARES[2] is a large-scale dataset built from 58 real-world firmware images (from 13 vendors including ASUS, Cisco, and Tenda) and 54 known vulnerable functions from nine widely-used libraries (cJSON, Libexpat, LibPNG,

[1]BINATLAS is named after Atlas, a figure from Greek mythology who symbolizes strength and foundation—qualities reflected in this dataset's comprehensiveness and robustness for BFSD research.

[2]Named after Ares, the Greek god of war, symbolizing strength and challenge in vulnerability detection.

TABLE II: Projects used for training, validation, and testing in BINATLAS. Projects marked with * are written in C++, otherwise in C.

	Compression	Network	Text	Database	Image	Other
Training	XZ	Nmap*, Openldap, Curl	Xerces-c*	SQLite	ImageMagick	Fmt*
Validation	Zlib	Libnet	yaml_cpp*	-	OpenJPEG	-
Testing	UnRAR	Openssl, ZeroMQ	JSON*	LevelDB*	Libwebp, Libtiff	PuTTy*

Libxml2, Lighttpd1.4, Nginx, OpenSSL, SQLite and Zlib). Unlike BINATLAS, its binaries have unknown compilation settings, reflecting realistic deployment scenarios. BINARES comprises 12,291 binaries and 3,676,923 functions across architectures such as 32-bit/64-bit MIPS and 32-bit ARM. The 54 vulnerable functions, compiled under default x86-64 settings, serve as queries. Ground truths were established by manually reviewing functions with the same name and the top-100 results from each tool. Three security experts with at least three years of experience conducted independent annotations, with discrepancies resolved through consensus discussion. This process took over 300 hours and involved inspecting more than 10,000 functions, ultimately identifying 1,442 homologous functions (1–72 per query, median: 23, average: 27).

Details regarding the limitations of prior datasets, as well as comprehensive information on BINATLAS and BINARES, are available on our website.

D. Tool Implementation

All tools, except DEJINA, have publicly available implementations and require only modifications for adapting to our datasets. For DEJINA, we fine-tuned jina-embeddings-v2-base-code [72], one of the SOTA source code embedding models available in early 2024, with a 1024-token limit for efficiency. CLAP was used in its original zero-shot settings [20]. DEJINA and other tools were trained with default configurations on the non-inlined subset of BINATLAS, using a balanced 1:1 ratio of positive and negative function pairs, totaling approximately four million pairs.

IV. EVALUATION

In this section, we present each research question along with the corresponding experimental setup, results, and the resulting **Observations** and **Insights**. Here, **Observations** refer to factual and objective findings directly derived from the results, while **Insights** represent deeper interpretations that can inform the practical application and further development of BFSD tools.

A. Factor Impact Analysis

RQ1: How do different factors impact BFSD tools?

This RQ explores the performance of diverse BFSD tools across varying compilation settings, aiming to determine their respective applicability and the impact of individual factors. Specifically, we evaluate each tool across following tasks: XO (cross-optimization levels), XC (cross-compiler and compiler version), XB (cross-bitness), XBCO (a combination of cross-bitness, compiler, compiler version and optimization levels), XA (cross-architecture), XAB (cross-architecture and bitness), XM (cross all compilation configurations), XM^{RW} (a variant of XM that excludes $O0$ and $O1$ optimization levels, as they are rarely used in practice), and XM-100K (an XM variant with an enlarged function pool of 100,000).

Experimental Setup. For each task, we randomly selected a corresponding positive pair from the test set, designating one function as the query and the other as the ground truth. To construct the function pool, multiple negative samples were randomly chosen for each query, with the ground truth function included. Each tool was then used to compute the similarity scores between the query and all functions in the pool, and to rank the ground truth accordingly. Across all tasks, we randomly selected 1,000 query functions for evaluation. The pool size for all tasks, except XM-100K, is set to 10,000, following the standard setup in recent studies [20], [21], [27]. For tasks that do not involve cross-architecture comparisons, functions are selected only from the x86 architecture, as Asm2Vec, PalmTree, jTrans and CLAP all support x86. We adopt Recall@1 (R1), Recall@10 (R10), and mean reciprocal rank (MRR) as evaluation metrics, consistent with prior work [20], [21], [27].

Results. Table III and Table IV present the performance of each tool on non-inlined binaries and inlined binaries, respectively. The results indicate that HermesSim and DEJINA consistently achieved the highest performance, outperforming the next best tool by at least 10 percentage points. In the XO and XC tasks, CLAP performed slightly below HermesSim and DEJINA but significantly outperformed all other tools, with GMN ranking next. Conversely, in the XB and XBCO tasks, GMN surpassed Gemini, CLAP, jTrans, PalmTree, SAFE, and Asm2Vec, indicating that both CLAP and GMN exhibit task-specific strengths. Among assembly code-based approaches, CLAP outperformed all others. In contrast, among graph-based methods, HermesSim achieved the best overall results.

Observation 1: HermesSim and DEJINA consistently delivered strong performance across all configuration scenarios. CLAP slightly trailed them in the XO and XC tasks, while GMN followed closely in the XB, XA, and XAB tasks.

We further examined the performance decline of SAFE, PalmTree, jTrans, and CLAP in the XB and XBCO tasks. These tools primarily focus on addressing the impact of optimization levels on BFSD, with CLAP pre-trained exclusively on binaries differing in optimization levels and compilers. However, cross-bitness configurations introduce substantial changes in register naming conventions and addressing modes, which significantly alter the assembly code structure and degrade the effectiveness of these tools. This highlights a key limitation of assembly-based representations in handling architectural and bitness diversity.

TABLE III: Performance of BFSD tools across different tasks in **non-inlined** binaries of BINATLAS. (R1:Recall@1, R10:Recall@10)

Tool	XO	XC	XB	XBCO	XA	XAB	XM	XM^{RW}	XM-100k
	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR
Asm2Vec	0.1 / 0.7 / 0.4	0.0 / 0.4 / 0.3	0.0 / 0.6 / 0.2	0.1 / 0.9 / 0.4	- / - / -	- / - / -	- / - / -	- / - / -	- / - / -
PalmTree	42.8 / 50.3 / 45.4	41.7 / 52.9 / 45.3	2.3 / 6.8 / 3.8	24.7 / 29.5 / 26.6	- / - / -	- / - / -	- / - / -	- / - / -	- / - / -
jTrans	56.9 / 65.7 / 60.1	56.8 / 72.2 / 62.3	5.8 / 14.3 / 8.9	34.6 / 45.9 / 38.3	- / - / -	- / - / -	- / - / -	- / - / -	- / - / -
CLAP	83.8 / 91.0 / 86.2	81.6 / 89.0 / 84.3	25.4 / 42.1 / 31.2	53.8 / 67.2 / 58.3	- / - / -	- / - / -	- / - / -	- / - / -	- / - / -
SAFE	17.7 / 20.9 / 19.0	14.3 / 21.1 / 16.9	0.4 / 1.9 / 1.2	9.7 / 14.5 / 11.6	0.2 / 0.8 / 0.8	0.5 / 2.4 / 1.3	5.5 / 8.8 / 7.1	10.0 / 13.3 / 11.5	5.0 / 5.9 / 5.3
Gemini	42.1 / 52.5 / 45.9	48.3 / 63.7 / 53.4	30.3 / 50.5 / 37.2	30.7 / 42.7 / 34.9	4.6 / 14.8 / 8.1	11.3 / 24.1 / 15.7	15.4 / 25.8 / 19.2	24.5 / 37.3 / 28.9	12.6 / 16.8 / 14.1
GMN	64.7 / 81.6 / 70.8	75.3 / 90.0 / 80.5	76.8 / 95.2 / 83.3	59.2 / 78.7 / 65.9	61.3 / 87.7 / 70.6	63.0 / 87.0 / 71.4	45.0 / 73.9 / 54.9	59.7 / 81.3 / 67.1	32.6 / 51.1 / 39.0
HermesSim	94.6 / 98.4 / 96.0	**95.9** / **99.0** / **97.1**	**97.7** / **99.7** / **98.4**	94.0 / 97.7 / 95.3	93.6 / 98.0 / 95.5	93.7 / **98.5** / **95.4**	88.5 / **95.4** / 91.0	89.3 / 94.9 / 91.3	83.0 / 92.1 / 86.2
DEJINA	**95.8** / **98.8** / **96.8**	95.8 / 98.6 / 96.8	96.1 / 98.9 / 97.1	**95.0** / **98.4** / **96.2**	**94.4** / **98.5** / **95.9**	**93.9** / 97.7 / 95.3	**90.0** / **95.4** / **92.1**	**91.7** / **96.3** / **93.3**	**84.4** / **92.6** / **87.0**

TABLE IV: Performance of BFSD tools across different tasks in **inlined** binaries of BINATLAS. (R1:Recall@1, R10:Recall@10)

Tool	XO	XC	XB	XBCO	XA	XAB	XM	XM^{RW}	XM-100k
	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR	R1 / R10 / MRR
Asm2Vec	0.2 / 0.5 / 0.4	0.0 / 1.1 / 0.4	0.0 / 0.4 / 0.2	0.0 / 0.3 / 0.2	- / - / -	- / - / -	- / - / -	- / - / -	- / - / -
PalmTree	37.3 / 45.3 / 40.1	37.1 / 49.8 / 41.5	4.6 / 7.8 / 5.7	25.2 / 31.9 / 27.8	- / - / -	- / - / -	- / - / -	- / - / -	- / - / -
jTrans	50.3 / 61.0 / 53.7	50.9 / 65.7 / 56.1	4.2 / 11.4 / 7.0	31.1 / 41.3 / 34.7	- / - / -	- / - / -	- / - / -	- / - / -	- / - / -
CLAP	75.0 / 83.3 / 77.9	76.5 / 85.2 / 79.5	25.4 / 43.4 / 31.2	48.5 / 61.3 / 52.8	- / - / -	- / - / -	- / - / -	- / - / -	- / - / -
SAFE	16.2 / 20.4 / 18.0	13.3 / 19.5 / 15.5	0.4 / 2.2 / 1.3	6.3 / 10.8 / 7.9	0.0 / 0.7 / 0.6	0.2 / 1.5 / 1.0	3.7 / 6.7 / 5.1	5.2 / 7.8 / 6.4	2.5 / 3.6 / 3.0
Gemini	39.2 / 49.8 / 43.0	42.0 / 56.2 / 47.2	30.1 / 53.1 / 37.7	28.7 / 41.2 / 33.0	5.1 / 15.9 / 8.9	12.3 / 26.3 / 17.0	12.3 / 22.3 / 16.3	19.2 / 30.0 / 23.2	9.7 / 14.7 / 11.3
GMN	58.2 / 73.6 / 63.3	65.7 / 83.9 / 71.9	79.7 / 94.6 / 85.2	52.9 / 72.1 / 59.5	66.0 / 87.2 / 73.6	66.1 / 88.0 / 73.7	42.2 / 63.4 / 49.4	49.7 / 69.7 / 56.7	28.6 / 44.2 / 34.3
HermesSim	88.3 / 93.4 / 90.2	**94.6** / **97.3** / **95.6**	96.5 / **99.3** / 97.6	84.3 / 90.9 / 86.7	94.7 / **98.3** / **96.2**	**94.8** / **98.2** / **96.0**	81.4 / 89.7 / 84.3	85.8 / 93.3 / 88.3	75.0 / 83.6 / 77.9
DEJINA	**89.1** / **94.5** / **91.0**	93.6 / 97.1 / 94.8	**97.0** / 98.9 / **97.7**	**86.9** / **92.2** / **88.8**	**94.8** / 97.3 / 95.8	94.4 / 98.1 / 95.8	**84.7** / **91.4** / **87.2**	**87.2** / **93.7** / **89.4**	**77.8** / **85.2** / **80.5**

Insight 1: BFSD tools that use assembly code as representation tend to underperform in the XB and XA tasks, due to limited robustness against syntactic variations introduced by changes in bitness or architecture.

When comparing tasks involving cross-single compilation settings across non-inlined dataset and inlined dataset, the XO and XC tasks show noticeable performance degradation, whereas the XB and XA tasks are less affected. This aligns with the understanding that different optimization levels apply varying inlining strategies, which also differ across compilers and evolve with compiler versions. To assess the impact of function inlining on BFSD tool performance, we conduct an in-depth analysis of the XO task, where inlining-induced degradation is most evident. In the following analysis, a failure case is defined as one in which the ground truth is not ranked in the top-10 predictions.

We first compare the proportion of inlined functions among all queries versus failure cases for each tool (see "Prop. Inline" in Table V). Notably, high-performing tools such as CLAP, GMN, HermesSim, and DEJINA exhibit a significantly higher proportion of inlined functions in their failure cases, suggesting inlining as a key factor. Lower-performing tools do not exhibit such features, as they are more sensitive to diverse compilation variations, with inlining contributing secondarily to their errors.

To further elucidate the impact of inlining, we distinguish between identical inlining, where both functions in a positive pair inline the same callees, and differential inlining, where the inlined callees differ between the pair. As shown in the "Prop. Inline-Id." and "Prop. Inline-Diff." rows of Table V, failure cases for HermesSim and DEJINA exhibit 0% identical inlining but over 80% differential inlining. This indicates that performance degradation in these tools is primarily driven by differential inlining, which introduces semantic inconsistencies in positive function pairs. To quantify this semantic divergence caused by inlining, we compute the Diff. Ratio, defined as the proportion of instructions originating from non-overlapping inlined callees between a pair of functions. For instance, given a positive function pair F and F', if F inlines callees G and H while F' inlines G' and I', the Diff. Ratio is computed as $\frac{|H|+|I'|}{\text{average}(|F|,|F'|)}$, where F denotes the size of F. Here, G and G' are homologous and thus excluded from the calculation. The last row in Table V reports the average Diff. Ratio for failure cases involving differential inlining for each tool. Across all queries, the average Diff. Ratio is 24.0%. However, in failure cases, this value rises to 72.9% for HermesSim and 78.4% for DEJINA, indicating that these tools are robust to minor inlining-induced variations but fail when semantic shifts become substantial. In contrast, lower-performing tools exhibit smaller Diff. Ratios in their failure cases, suggesting they are more vulnerable to even slight semantic inconsistencies introduced by inlining.

Insight 2: Function inlining remains a major challenge for BFSD tools. Performance degradation primarily occurs when positive pairs undergo asymmetric inlining. Higher-performing tools, such as HermesSim and DEJINA, are more robust to function inlining, failing when inlining introduces substantial semantic differences (averaging over 70%).

A comparison of the XM and XM-100K columns in Table III and Table IV reveals that as the function pool size increases from 10,000 to 100,000, the overall MRR of all tools declines, with absolute drops ranging from 2.1% to 15.1%. However, the extent of degradation varies significantly across tools. HermesSim and DEJINA demonstrate stronger

TABLE V: Function inlining impact analysis across the failure cases of different tools and queries.

Metric	Asm2Vec	PalmTree	jTrans	CLAP	SAFE	Gemini	GMN	HermesSim	DEJINA	Queries
Prop. Inline	51.6%	48.5%	49.5%	65.3%	51.6%	52.0%	62.9%	81.8%	83.6%	51.5%
Prop. Inline-Id.	18.6%	5.4%	3.3%	4.8%	13.4%	5.6%	3.0%	0.0%	0.0%	18.7%
Prop. Inline-Diff.	33.0%	43.1%	46.2%	60.5%	38.2%	46.4%	59.8%	81.8%	83.6%	32.8%
Diff. Ratio	24.1%	35.4%	38.8%	45.9%	28.8%	35.9%	46.9%	72.9%	78.4%	24.0%

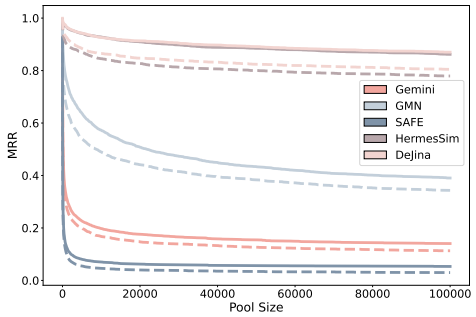

Fig. 3: MRR by the size of function pool. (Solid lines: non-inlined dataset; dashed lines: inlined dataset.)

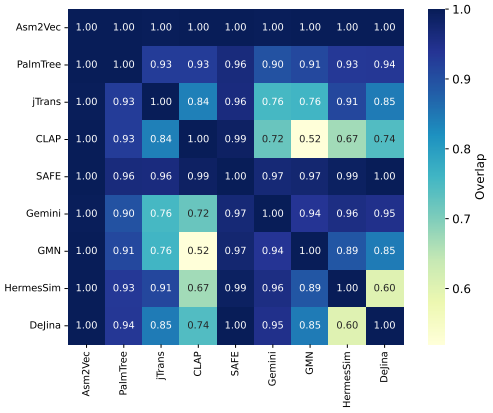

Fig. 4: The overlap coefficient between tool pairs.

robustness, with MRR reductions of less than 7%, whereas Gemini, GMN, and SAFE suffer more pronounced performance drops, each exceeding 25%, relatively. Figure 3 shows the MRR performance of Gemini, GMN, SAFE, HermesSim, and DEJINA across increasing function pool sizes. Other tools are excluded due to incompatibility with cross-architecture settings. As observed, MRR drops sharply when the pool size increases from 0 to 20,000, revealing a key limitation of current BFSD evaluations—namely, the lack of assessment under large-scale function pools. Most existing studies cap the pool size at 10,000, which may not reflect real-world conditions. Although the decline slows as the pool grows, performance continues to degrade. In practical scenarios, where function pools can reach millions, this degradation becomes much more substantial compared to smaller-scale experimental settings.

Observation 2: The size of the function pool affects the performance of BFSD tools. As the pool size increases, their MRR tends to decline rapidly at first and then levels off.

B. Consistency Analysis and Combination Strategy

RQ2: Can BFSD tools be combined to improve overall results?

The results of RQ1 reveal that the performance of different tools varies across tasks. These disparities raise the question of whether the tools can complement one another. To explore this possibility, this RQ first analyzes the consistency among these tools based on the results from RQ1. Subsequently, we investigate various combination strategies and evaluate their effectiveness in enhancing overall performance.

Experimental Setup. In this RQ, a query is considered a failure case if its ground truth is ranked outside the top 10. We analyze the consistency of failure cases between two tools using overlap coefficient [75]:

$$overlap(A, B) = \frac{|A \cap B|}{min(|A|, |B|)}$$

where A and B denote the sets of failure cases for each tool. The overlap coefficient quantifies the extent to which the failure cases of the better-performing tool overlap with those of another, with higher values indicating similar failure patterns and reduced potential for complementary gains.

Results. We analyze the consistency of failure cases across tools on the XBCO task using the inlined dataset. Specifically, we compute the overlap coefficient for all tool pairs, as presented in Figure 4. All pairs exhibit coefficients above 0.5, indicating that at least half of one tool's failure cases are shared with the other. But some of them exibit relatively lower value than other tool pairs, e.g. GMN and CLAP (0.52), HermesSim and DEJINA (0.6).

Observation 3: Some tool pairs exhibit relatively low overlap, indicating potential for performance improvement through complementary use.

To investigate the reason behind their relatively lower inconsistency, we manually analyzed all common (47) and unique ones in HermesSim (44) and DEJINA (31), as they are the best-performing tools. The reasons are concluded in Table VI.

In the table, ❶ Function Inlining refers to cases where aggressive inlining by the compiler leads to substantial difference between function pairs, thereby complicating similarity detection. ❷ Flow Changes denote instances where aggressive

TABLE VI: Reasons for failure cases in HermesSim and DEJINA.

Tool	Function Inlining	Flow Changes	Indistinctive Functions	Distinct Instruction Sets	Decompiled Code Issues
Common	42 (89.4%)	5 (10.6%)	0 (0%)	0 (0%)	0 (0%)
HermesSim	20 (45.5%)	17 (38.6%)	7 (15.9%)	0 (0%)	0 (0%)
DEJINA	15 (48.4%)	6 (19.4%)	2 (6.5%)	4 (12.9%)	4 (12.9%)

compiler optimizations significantly modify the control flow structure, resulting in pronounced structural variations in the decompiled code. ❸ Indistinctive Functions capture scenarios where the target functions lack distinctive characteristics, or where retrieved candidates exhibit high similarity to the query function, making precise differentiation challenging. ❹ Distinct Instruction Sets describe cases where compilers, driven by performance or size constraints, selectively utilize SIMD instructions instead of conventional ones, leading to significant discrepancies in the decompiled representation. Finally, ❺ Decompiled Code Issues encompass failures caused by decompiler errors that fail to generate valid decompiled code.

Though HermesSim and DEJINA share similar underlying causes for their respective unique failures, such as function inlining and flow changes, our analysis reveals that the primary reasons one tool may fail while the other succeeds are rooted in their distinct design philosophies. HermesSim constructs graphs based on intra-function dependency relationships and relies on a graph-based model for feature extraction. Its sensitivity to flow variations allows it to effectively distinguish between functions with similar functionality but different structural patterns. However, this same sensitivity makes it more prone to failure when the flow undergoes substantial transformations, such as those introduced by aggressive compiler optimizations or inlining. In contrast, DEJINA leverages language models to generate embeddings from decompiled code, enabling it to extract rich semantic features from symbol names and string literals. This enables it to distinguish functions with similar structural patterns but different symbol names or string literals. However, this reliance on semantic content renders it significantly more vulnerable to the quality and consistency of the decompiler's output. For example, decompilation errors or inconsistent recovery of symbolic information and strings across function pairs can lead to failures in DEJINA's predictions. Additionally, the insensitivity to dependency relationships makes it fail to distinguish similar functions. A case study for further explaining the inconsistency of HermesSim and DEJINA is provided on our website.

These observations suggest that BFSD tools built on different representations (graph, assembly code, and decompiled code, etc.) may exhibit complementary strengths and vulnerabilities.

Insight 3: BFSD tools based on different representations exhibit varying robustness to function pair discrepancies, resulting in inconsistent failure patterns. This variability highlights the potential for performance improvements through integrating complementary approaches.

Experimental Setup. Motivated by the above observations and insights, we conduct a preliminary investigation into combining BFSD tools to enhance overall performance. The central hypothesis is that tool pairs built on different representations and with comparable performance levels are more likely to be complementary. Tools based on different representations exhibit distinct failure patterns, thereby increasing the likelihood of complementary strengths. At the same time, combining tools with similar performance levels helps mitigate the risk of performance degradation caused by noise from a substantially weaker tool. To evaluate this hypothesis, we assess all pairwise combinations on the XBCO task and examine the resulting performance improvements.

Combination strategy. To integrate multiple BFSD tools, we propose a voting strategy, which ranks candidate functions based on consensus across tools. Functions retrieved by all tools are prioritized and ranked by their average position, followed recursively by those retrieved by fewer tools. This strategy is practical, as it requires no parameter tuning and is compatible with any BFSD tool. For example, if Tool I returns A, B, C, D and Tool II returns D, A, E, C, the overlapping functions A, C, and D are ranked first by average rank: A (1.5), D (2.5), C (3). The remaining functions B and E are then ranked as B (2), E (3), resulting in a final order of A, D, C, B, E.

TABLE VII: The results of tool combinations ranked by MRR improvement. (δMRR: The MRR gap between the tools in a pair.)

Combination	δMRR	Combination Results (R1 / R10 / MRR)
GMN+CLAP	6.7	66.2 (↑13.3) / 83.5 (↑11.4) / 72.3 (↑12.8)
jTrans+Gemini	1.7	37.0 (↑5.9) / 52.8 (↑11.5) / 43.2 (↑8.5)
CLAP+Gemini	19.8	50.8 (↑2.3) / 69.7 (↑8.4) / 58.0 (↑5.2)
PalmTree+Gemini	5.2	30.6 (↑1.9) / 44.1 (↑2.9) / 35.9 (↑2.9)
HermesSim+DEJINA	2.1	88.7 (↑1.8) / 94.7 (↑2.5) / 90.8 (↑2.0)
CLAP+jTrans	18.1	47.5 (↓1.0) / 64.8 (↑3.5) / 53.9 (↑1.1)
PalmTree+jTrans+SAFE	26.8	26.9 (↓4.2) / 42.0 (↓0.7) / 32.8 (↓2.1)
CLAP+jTrans+Asm2Vec	52.6	44.6 (↓3.9) / 63.3 (↓1.5) / 51.6 (↓2.3)
DeJina+HermesSim+GMN	29.3	85.1 (↓3.6) / 94.2 (↓0.5) / 88.4 (↓2.4)

Results. We list the combinations of two tools that with more than 1% improvement in MRR in the first part of Table VII, the results of all combinations can be found in our website. Notably, combining GMN and CLAP yields the most consistent improvements across metrics under both strategies.It improves R1, R10, and MRR by 13.3%, 11.4%, and 12.8%, respectively. The combination of DEJINA and HermesSim achieves the highest overall performance: 88.7%, 94.7%, and 90.8%.

Observation 4: The proposed combination strategy can yield performance improvements of up to 12.8%.

Moreover, all tool pairs yielding MRR improvements are based on different representations, except CLAP and jTrans, which also show the smallest gain. Effective combinations typically exhibit small performance gaps between paired tools. Notably, all tool pairs with distinct representations and low performance gaps demonstrate improved performance after combination. Conversely, tool pairs with large performance gaps consistently suffer degradation. For example, combining HermesSim and CLAP results in a 6.0% MRR drop due to their large performance gap (33.9% δMRR). These findings empirically validate our hypothesis and establish clear criteria on performance gap for effective tool complementarity.

Insight 4: Effective tool combination requires two criteria: ❶ tools must be based on different binary function representations, leading to complementary failure patterns, and ❷ tools must exhibit a small performance gap, as large disparities introduce noise from the weaker tool.

Our results show that when the δMRR exceeds 20%, combinations fail to improve performance. All tool pairs satisfying both criteria achieve MRR improvements (up to 12.8%), demonstrating the importance of balancing complementarity with comparable capability.

To further explore potential gains from incorporating additional tools, we iteratively introduce a third tool into the pairwise combinations that previously demonstrated performance improvement. The best-performing three-tool combinations are presented in the second part of Table VII. However, all such combinations result in performance degradation. This outcome likely stems from a violation of the criteria, as each three-tool combination includes either two tools based on the same representation or at least with a δMRR exceeding 20%.

C. RQ3: Vulnerability Detection & Strategy Validation

RQ3: How do BFSD tools and the combination strategy perform in large-scale real-world vulnerability detection?

A typical usage of BFSD is to use a known vulnerable function as the query and search for similar functions within a large binary function pool. This enables the discovery of vulnerabilities that arise from code reuse or that exist in semantically similar functions. Since the function pool often comprises thousands of binaries from diverse projects, it is common that there are either multiple target functions or none at all, making the number of target functions inherently unknown.

In this RQ, we evaluate the practical effectiveness of the combination strategy proposed in RQ2 on BINARES, a large-scale, real-world dataset. We also highlight key challenges and insights in applying BFSD tools to real-world vulnerability detection. Given BINARES 's multi-architecture nature, only cross-architecture tools are included. Due to the dataset's scale and high evaluation cost, we focus on the top three tools—DEJINA, HermesSim, and GMN—as well as the best-performing combination from RQ2 (HermesSim + DEJINA).

TABLE VIII: Results of homologous function detection on BINARES. The content in each cell is Precision / Recall / F1 score. (T: Threshold of similarity.)

Baseline	Top-10	Top-25	Top-50	Best Threshold
GMN	26.9 / 9.2 / 13.1	19.6 / 16.0 / 16.6	14.3 / 22.5 / 16.4	19.4 / 11.9 / 12.3 (T=0.98)
HermesSim	71.3 / 45.4 / 42.7	55.6 / 67.8 / 51.6	36.7 / 78.5 / 43.8	45.9 / 43.4 / 40.2 (T=0.49)
DEJINA	70.2 / 43.3 / 40.5	55.9 / 65.7 / 50.1	38.9 / 80.7 / 45.7	57.1 / 44.0 / 44.6 (T=0.79)
DEJINA + HermesSim	77.4 / 48.7 / 46.2	63.6 / 75.9 / 58.5	45.0 / 91.9 / 53.4	-

Experimental Setup. We use 54 vulnerable functions as queries against a pool of 3,676,923 functions in BINARES, and evaluate BFSD tools across three tasks: ❶ Homologous function detection: We treat functions derived from the same source code as ground truth. The ground truth was established manually as described in § III-C. ❷ Vulnerable Homologous Function Detection: To evaluate the tools' effectiveness in identifying vulnerable instances among homologous functions, we manually inspected 500 randomly sampled homologous candidates retrieved by the top-performing tool. This step accounts for scenarios where some homologous functions may have been patched or originate from versions preceding the vulnerability. ❸ Vulnerable Non-Homologous Function Detection: To assess the ability of tools to identify vulnerabilities beyond strict source-level similarity, we also examined 500 top-ranked non-homologous functions. Each function was manually inspected to determine whether it contained a similar vulnerability, capturing semantically similar but structurally divergent cases.

The metrics used in previous RQs (R1, R10, and MRR) are not suitable here, as they assume a single ground truth per query. In contrast, this RQ addresses scenarios with multiple ground truths, necessitating a more comprehensive evaluation. Therefore, we adopt precision, recall, and F1 score, which are more appropriate for assessing performance in multi-ground truth settings.

Results of homologous function detection. Table VIII presents the homologous function detection results on the BINARES dataset. We report the performance of each tool in the top-10, top-25, and top-50 retrieved results. In addition, we apply a threshold-based approach to distinguish positive and negative samples, and report the threshold at which each tool achieves its highest F1 score in the last column. The results show that the proposed combination method consistently outperforms all individual tools across all evaluation settings, achieving an F1 score of 58.5% in the top-25 results, representing a 13.4% improvement over the best-performing individual tool, HermesSim (51.6%).

Observation 5: The tool combination strategies demonstrate strong applicability and effectiveness (13.4% improvement) in large-scale, real-world vulnerability detection scenarios.

The F1 score at the top-25 cutoff consistently outperforms those at top-10, top-50, and the best-threshold settings. This can be attributed to the distribution of homologous functions

per query, which has a median value of 23 and an average value of 27 that are both close to 25. Furthermore, the relatively poor performance under the best-threshold setting suggests that threshold-based classification is impractical in this context, as identifying the optimal threshold for each tool requires additional effort and tuning, limiting its applicability in real-world scenarios.

Results of vulnerable homologous function detection. Manual analysis of 500 confirmed homologous functions revealed that only 56% (280) were actually vulnerable. This indicates that while BFSD tools effectively identify homologous functions, they struggle to distinguish between vulnerable and non-vulnerable ones. This limitation arises because BFSD tools emphasize overall functional similarity, whereas vulnerability detection often hinges on subtle differences in specific code fragments. Additionally, manual verification is labor-intensive, requiring the identification of vulnerable locations from patches and matching them to corresponding code segments in target functions—an effort that varies with each vulnerability. This highlights the need for fine-grained techniques capable of matching code fragments within functions to approximate potential vulnerability locations. Such methods could substantially ease the verification burden and represent a promising avenue for future research.

Observation 6: BFSD tools cannot differentiate between vulnerable and non-vulnerable homologous functions. Effective vulnerability detection thus requires integration with fine-grained localization and validation techniques.

Results of vulnerable non-homologous function detection. We further analyzed 500 top-ranked non-homologous functions retrieved by the combination of HermesSim and DEJINA. Surprisingly, only one function exhibited the similar vulnerability to the queried ones-specifically, an incorrect return value check for a particular API. While some non-homologous functions shared functional similarities (e.g., image processing), this highlights a key limitation of BFSD tools: they can retrieve functionally similar but rarely vulnerability-equivalent functions. Addressing this challenge will likely require fundamentally new methodologies tailored to vulnerability semantics rather than general functional similarity.

Insight 5: BFSD tools operate at the function level and often overlook fine-grained vulnerability semantics, limiting their effectiveness in detecting similar vulnerabilities that reside in small code regions.

V. DISCUSSION

A. Practical Implications for the Community

Our findings provide actionable strategies, practical guidance, and resources across multiple stakeholder groups.

For practitioners and security analysts, we recommend selecting and combining complementary BFSD tools based on different representations with comparable performance levels. HermesSim and DEJINA represent favorable choices given their consistently strong performance (§ IV-A). When constraints exist, such as lightweight model requirements or evaluating new tools, practitioners should identify target scenario characteristics (e.g., cross-architecture, cross-bitness), construct test datasets with BINATLAS under similar conditions, and apply the selection criteria established in this work.

For researchers and tool developers, our study identifies critical directions (§ V-B). The insights on representation-level abstractions and tool complementarity also provide concrete guidance for designing next-generation BFSD techniques.

For educators and the community, we provide open-source datasets (BINATLAS and BINARES) and ready-to-use implementations, lowering barriers to reproducible research and accelerating progress in the BFSD field.

B. Future Directions for BFSD

Addressing function inlining. As shown in § IV-A, inconsistent inlining between homologous functions significantly disrupts semantics and leads to detection failures. Although prior work [5], [59], [76] attempts to mitigate this, complex inlining remains challenging. Future work could explore partial matching or inclusion-based methods to better handle inlining-induced variability.

Unifying multiple representations. Our findings in § IV-B and § IV-C show that tools based on distinct representations exhibit complementary strengths. Combining such tools yields notable performance gains, highlighting the promise of a unified BFSD framework that integrates multiple representations for enhanced robustness and accuracy.

Scaling to large search spaces. Tool performance degrades with larger candidate pools (§ IV-A, § IV-C), limiting scalability. Effective filtering is essential. Techniques like software composition analysis (SCA) [77], [78] can help narrow the search space. Application-specific filtering strategies, possibly integrating recent advances [9], [39], offer a promising path forward.

Reducing manual verification effort. Current BFSD tools typically return similarity scores, requiring costly manual inspection to check if they are the searching targets. In § IV-C, ground-truth labeling demanded substantial human effort. Two directions may help with this: (1) fine-grained matching to localize relevant code fragments, and (2) leveraging LLMs for automated result validation. Our preliminary experiments suggest that LLMs can effectively validate both homologous and vulnerable function matches. We provide the results of the preliminary experiment on our website.

Towards finer-grained search. BFSD tools focus on global similarity and struggle to pinpoint specific vulnerabilities. This limits their effectiveness in tasks like vulnerability detection, where identifying precise vulnerable code regions is critical. Future research should explore searching at a finer granularity—e.g., matching vulnerable code fragments or semantic patterns within functions—beyond coarse-grained function similarity.

Leveraging higher-level representations. Our evaluation reveals that top-performing tools, DEJINA and HermesSim,

share a key design principle: lifting binary functions to unified, higher-level representations. DEJINA employs decompiled code, while HermesSim utilizes intermediate representation and constructs semantic-oriented graphs through static analysis. These abstractions effectively eliminate architectural differences and reduce inconsistencies, unlike other tools operating directly on assembly code or dependency graphs. This suggests that future BFSD research should prioritize architecture-agnostic, higher-level representations to achieve robust cross-platform and cross-optimization detection.

C. Threats to Validity

External validity. Our findings are derived from experiments on two datasets, BINATLAS and BINARES. However, their generalizability to other settings, such as obfuscated code [5], [19], [79] or proprietary binaries, remains unverified. Broader validation on larger and more diverse datasets is needed. While our evaluation covers a subset of available tools, the tool combination strategies, and the limitations of BFSD tools are tool-agnostic and applicable across BFSD frameworks. Moreover, our tools are trained on the non-inlined dataset. While training on the inlined dataset may yield performance improvements, such improvements may not generalize to all inlining patterns, as these patterns are content-dependent and vary across different functions.

Internal validity. Our experiments face two primary threats to internal validity. First, in § IV-A and § IV-B, query functions were selected randomly, which may introduce variability. To address this, we used 1,000 query functions and reported averaged results. Second, in § IV-C, results required manual verification, which may involve subjectivity or errors. To reduce this risk, we adopted a two-person verification protocol: one performed the analysis, and a second independently reviewed and confirmed the results.

VI. RELATED WORKS

A. Evaluation Studies of BFSD Tools

Recent studies have evaluated BFSD tools in different settings. Marcelli et al. [27] compared fuzzy hashing and machine learning approaches, revealing that machine learning methods performs better. Fu et al. [10] assessed AI-based BFSD methods and their downstream applications, providing a structural analysis of common neural networks-based approaches. These prior works exhibit three key limitations: **Limited evaluation scope.** Existing studies often overlook critical real-world factors, such as function inlining and varying pool sizes, thereby limiting the depth and generalizability of their findings. **Lack of analysis on practical usage.** Prior evaluations focus primarily on assessing the effectiveness of individual tools, missing the opportunity to explore practical usage strategies, such as combining tools or adapting them to specific contexts, that could improve real-world applicability. **Non-representative and biased datasets.** Many evaluations rely on small-scale datasets, typically involving fewer than 10 vulnerabilities or firmware images, with limited architectural and project diversity. Furthermore, some datasets suffer from issues such as mislabeling (e.g., misclassified inlined functions) and heavy project bias (e.g., 57.6% of functions sourced from Z3), undermining the reliability and generalizability of the results.

In contrast, our evaluation offers more reliable and representative insights, enabled by higher-quality and more diverse datasets. Particularly, we propose an actionable strategy for tool combination, which shows practical effectiveness.

B. Literature Reviews of BFSD Approaches

Several surveys have reviewed BFSD methods over time [80]–[82]. Haq et al. [80] provided a broad overview of BFSD techniques, including their characteristics, implementations, and applications. Alrabaee et al. [81] focused on binary code fingerprinting, categorizing methods by similarity levels, features, and detection strategies. Ruan et al. [82] offered a multidimensional comparison, analyzing the strengths and limitations in addressing the diverse characteristics of binary code. However, these surveys primarily offer qualitative analyses and lack quantitative evaluations of tool performance.

VII. CONCLUSION

In this paper, we present the first large-scale empirical study of AI-based BFSD tools on two high-quality and diverse datasets. Based on three research questions, we propose an actionable tool combination strategy, that demonstrates strong effectiveness in large-scale, real-world vulnerability detection tasks, with an improvement of 13.4%. Furthermore, we identify key limitations of current BFSD tools and offer insights for future research, including the promising potential of LLMs in enhancing BFSD capabilities.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful feedback on an earlier version of this paper. This work is partly supported by National Key R&D Program of China under Grant #2022YFB3103901, Chinese National Natural Science Foundation (Grants #62202462, #62302500, #62032010). This research is supported by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG4-GC-2023-008-1B); by the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN); and by the Prime Minister's Office, Singapore under the Campus for Research Excellence and Technological Enterprise (CREATE) Programme. Any opinions, findings and conclusions, or recommendations expressed in these materials are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, Cyber Security Agency of Singapore, Singapore.

REFERENCES

[1] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *Acm sigplan notices*, vol. 51, no. 6, pp. 266–280, 2016.

[2] ——, "Similarity of binaries through re-optimization," in *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, 2017, pp. 79–94.

[3] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 896–899.

[4] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 363–376.

[5] S. H. Ding, B. C. Fung, and 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.

[6] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 2019, pp. 309–329.

[7] J. Yang, C. Fu, X.-Y. Liu, H. Yin, and P. Zhou, "Codee: A tensor embedding scheme for binary code search," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2224–2244, 2021.

[8] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and 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.

[9] S. Yang, C. Dong, Y. Xiao, Y. Cheng, Z. Shi, Z. Li, and L. Sun, "Asteria-pro: Enhancing deep learning-based binary code similarity detection by incorporating domain knowledge," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–40, 2023.

[10] L. Fu, P. Liu, W. Meng, K. Lu, S. Zhou, X. Zhang, W. Chen, and S. Ji, "Understanding the ai-powered binary code similarity detection," *arXiv preprint arXiv:2410.07537*, 2024.

[11] I. Ahmad and L. Luo, "Unsupervised binary code translation with application to code clone detection and vulnerability discovery," in *Conference on Empirical Methods in Natural Language Processing*, 2023.

[12] R. Mirzazadeh, M. H. Moattar, and M. V. Jahan, "Metamorphic malware detection using linear discriminant analysis and graph similarity," in *2015 5th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 2015, pp. 61–66.

[13] J. Jang, D. Brumley, and 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.

[14] J. Wang, M. Sharp, C. Wu, Q. Zeng, and L. Luo, "Can a deep learning model for one architecture be used for others? retargeted-architecture binary code analysis," in *USENIX Security Symposium*, 2023.

[15] S. Li, Y. Wang, C. Dong, S. Yang, H. Li, H. Sun, Z. Lang, Z. Chen, W. Wang, H. Zhu, and L. Sun, "Libam: An area matching framework for detecting third-party libraries in binaries," *ACM Transactions on Software Engineering and Methodology*, vol. 33, pp. 1 – 35, 2023.

[16] L. Jiang, J. An, H. Huang, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "Binaryai: Binary software composition analysis via intelligent binary source code matching," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 2771–2783, 2024.

[17] C. Dong, S. Li, S. Yang, Y. Xiao, Y. Wang, H. Li, Z. Li, and L. Sun, "Libvdiff: Library version difference guided oss version identification in binaries," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 791–802, 2024.

[18] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *2014 IEEE 25th international symposium on software reliability engineering*. IEEE, 2014, pp. 66–77.

[19] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[20] H. Wang, Z. Gao, C. Zhang, Z. Sha, M. Sun, Y. Zhou, W. Zhu, W. Sun, H. Qiu, and X. Xiao, "Clap: Learning transferable binary code representations with natural language supervision," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 503–515.

[21] H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue, "Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection," in *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA*, 2024.

[22] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*. PMLR, 2019, pp. 3835–3845.

[23] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236–3251.

[24] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *Network and Distributed System Security Symposium*, 2016.

[25] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and 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.

[26] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary function clustering using semantic hashes," in *2012 11th International Conference on Machine Learning and Applications*, vol. 1. IEEE, 2012, pp. 386–391.

[27] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2099–2116.

[28] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "jtrans: Jump-aware transformer for binary code similarity," *arXiv preprint arXiv:2205.12713*, 2022.

[29] S. Jiang, C. Fu, S. He, J. Lv, L. Han, and H. Hu, "Bincola: Diversity-sensitive contrastive learning for binary code similarity detection," *IEEE Transactions on Software Engineering*, vol. 50, pp. 2485–2497, 2024.

[30] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and 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.

[31] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[32] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," *it - Information Technology*, vol. 59, pp. 83 – 91, 2015.

[33] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and 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.

[34] X. Xu, Z. Xuan, S. Feng, S. Cheng, Y. Ye, Q. Shi, G. Tao, L. Yu, Z. Zhang, and X. Zhang, "Pem: Representing binary program semantics for similarity analysis via a probabilistic execution model," *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[35] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "Mutantx-s: Scalable malware clustering based on static features," in *USENIX Annual Technical Conference*, 2013.

[36] S. Cesare, Y. Xiang, and W. Zhou, "Control flow-based malware variant-detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, pp. 307–317, 2014.

[37] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 57–67, 2016.

[38] L. Chen, Z. He, H. Wu, F. Xu, Y. Qian, and B. Mao, "Dicomp: Lightweight data-driven inference of binary compiler provenance with high accuracy," *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 112–122, 2022.

[39] H. Wang, Z. Gao, C. Zhang, M. Sun, Y. Zhou, H. Qiu, and X. Xiao, "Cebin: A cost-effective framework for large-scale binary code similarity detection," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 149–161.

[40] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *USENIX Security Symposium*, 2014.

[41] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 319–330, 2017.

[42] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu, "Accurate and scalable cross-architecture cross-os binary code search with emulation," *IEEE Transactions on Software Engineering*, vol. 45, pp. 1125–1149, 2019.

[43] Y. Hu, H. Wang, Y. Zhang, B. Li, and D. Gu, "A semantics-based hybrid approach on binary code similarity comparison," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1241–1258, 2019.

[44] A. Zhou, Y. Hu, X. Xu, and C. Zhang, "Arcturus: Full coverage binary similarity analysis with reachability-guided emulation," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–31, 2024.

[45] Z. Fei, L. Xiaopeng, Y. Patrick, L. Lannan, Z. Qiang, and Z. Zhexin, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[46] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *arXiv preprint arXiv:2012.08680*, 2020.

[47] S. VenkataKeerthy, S. Banerjee, S. Dey, Y. Andaluri, R. PS, S. Kalyanasundaram, F. M. Q. Pereira, and R. Upadrasta, "Vexir2vec: An architecture-neutral embedding framework for binary similarity," *arXiv preprint arXiv:2312.00507*, 2023.

[48] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with bert-based transferable similarity learning," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 361–374.

[49] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, R. Baldoni *et al.*, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, 2019, pp. 1–11.

[50] H. Wang, P. Ma, S. Wang, Q. Tang, S. Nie, and S. Wu, "sem2vec: Semantics-aware assembly tracelet embedding," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–34, 2023.

[51] W. Li, J. Lu, R. Xiao, P. Shao, and S. Jin, "Rcfg2vec: Considering long-distance dependency for binary code similarity detection," *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 770–782, 2024.

[52] K. He, Y. Hu, X. Li, Y. Song, Y. Zhao, and D. Gu, "Strtune: Data dependence-based code slicing for binary similarity detection with fine-tuned representation," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 10 233–10 245, 2024.

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

[54] G. Kim, S. Hong, M. Franz, and D. Song, "Improving cross-platform binary analysis using representation learning via graph alignment," *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[55] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.

[56] Y. Guo, P. Li, Y. Luo, X. Wang, and Z. Wang, "Exploring gnn based program embedding technologies for binary related tasks," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 366–377.

[57] O. Vinyals, S. Bengio, and M. Kudlur, "Order matters: Sequence to sequence for sets," *CoRR*, vol. abs/1511.06391, 2015.

[58] X. Xu, S. Feng, Y. Ye, G. Shen, Z. Su, S. Cheng, G. Tao, Q. Shi, Z. Zhang, and X. Zhang, "Improving binary code similarity transformer models by semantics-driven instruction deemphasis," *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[59] L. Jia, C. Wu, P. Zhang, and Z. Wang, "Codeextract: Enhancing binary code similarity detection with code extraction techniques," in *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2024, pp. 143–154.

[60] Y. Wang, H. Li, X. Zhu, S. Li, C. Dong, S. Yang, and K. Qin, "Binenhance: A enhancement framework based on external environment semantics for binary code search," *arXiv preprint arXiv:2411.01102*, 2024.

[61] W. K. Wong, H. Wang, Z. Li, and S. Wang, "Binaug: Enhancing binary similarity analysis with low-cost input repairing," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 51–63, 2024.

[62] H. Wang, P. Ma, Y. Yuan, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Enhancing dnn-based binary code function search with low-cost equivalence checking," *IEEE Transactions on Software Engineering*, vol. 49, pp. 226–250, 2023.

[63] J. Wang, C. Zhang, L. Chen, Y. Rong, Y. Wu, H. Wang, W. Tan, Q. Li, and Z. Li, "Improving ml-based binary function similarity detection by assessing and deprioritizing control flow graph features," in *USENIX Security Symposium*, 2024.

[64] L. Jia, C. Wu, B. Tang, P. Zhang, Z. Jiang, Y. Yang, N. Liu, J. Zhang, and Z. Wang, "Enhancing learning-based binary code similarity detection model through adversarial training with multiple function variants," in *Conference on Empirical Methods in Natural Language Processing*, 2024.

[65] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International conference on machine learning*. PMLR, 2016, pp. 2702–2711.

[66] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[67] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[68] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[69] Z. Lin, M. Feng, C. N. d. Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A structured self-attentive sentence embedding," *arXiv preprint arXiv:1703.03130*, 2017.

[70] Y. Liu, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, vol. 364, 2019.

[71] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[72] "jinaai/jina-embeddings-v2-base-code · Hugging Face — huggingface.co," https://huggingface.co/jinaai/jina-embeddings-v2-base-code, [Accessed 19-03-2025].

[73] M. Günther, J. Ong, I. Mohr, A. Abdessalem, T. Abel, M. K. Akram, S. Guzman, G. Mastrapas, S. Sturua, B. Wang *et al.*, "Jina embeddings 2: 8192-token general-purpose text embeddings for long documents," *arXiv preprint arXiv:2310.19923*, 2023.

[74] F. Zuo, C. Tompkins, Q. Zeng, L. Luo, Y. R. Choe, and J. Rhee, "Binsimdb: Benchmark dataset construction for fine-grained binary code similarity analysis," *ArXiv*, vol. abs/2410.10163, 2024.

[75] M. McGill, "An evaluation of factors affecting document ranking by information retrieval systems." 1979.

[76] A. Jia, M. Fan, X. Xu, W. Jin, H. Wang, and T. Liu, "Cross-inlining binary function similarity detection," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 2758–2770, 2024.

[77] M. Feng, Z. Yuan, F. Li, G. Ban, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu, A. Piao, J. Xue, and W. Huo, "B2sfinder: Detecting open-source software reuse in cots software," *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1038–1049, 2019.

[78] B. Zhao, S. Ji, X. Zhang, Y. Tian, Q. Wang, Y. Pu, C. Lyu, and R. A. Beyah, "Uvscan: Detecting third-party component usage violations in iot firmware," in *USENIX Security Symposium*, 2023.

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

[80] I. U. Haq and J. Caballero, "A survey of binary code similarity," *Acm computing surveys (csur)*, vol. 54, no. 3, pp. 1–38, 2021.

[81] S. Alrabaee, M. Debbabi, and L. Wang, "A survey of binary code fingerprinting approaches: taxonomy, methodologies, and features," *ACM Computing Surveys (CSUR)*, vol. 55, no. 1, pp. 1–41, 2022.

[82] L. Ruan, Q. Xu, S. Zhu, X. Huang, and X. Lin, "A survey of binary code similarity detection techniques," *Electronics*, vol. 13, no. 9, p. 1715, 2024.