BinSAFE: Extending Functions Embeddings to Entire Binaries

Gianluca Capozzi[1,*,†], Giordano Laurenzi[1,*,†], Marco Mormando[2], Carmine Gianni[3], Gianluca Marcilli[4], Leonardo Querzoni[1] and Giuseppe Di Luna[1]

[1]*Sapienza University of Rome*
[2]*Independent Researcher*
[3]*Cy4gate s.p.a.*
[4]*Innovation and Space Office - General Staff - Italian Navy*

Abstract

The increasing number of cybersecurity regulations highlights the growing importance of scrutinizing firmware in smart devices to ensure compliance and security. However, such scrutiny often involves reverse engineering—a process that is time-consuming, costly, and reliant on highly specialized skills that are in short supply. Consequently, there is a rising demand from the industrial sector for innovative tools and solutions to streamline and accelerate firmware analysis, making it more efficient and accessible. In this paper, we introduce *BinSAFE*, an integrated system for comparing binaries within a firmware against a knowledge base. *BinSAFE* supports adding new firmware, extracting its binaries, and matching them against the knowledge base for comparison. The core of *BinSAFE* is a graph-matching algorithm that leverages embedding-based solutions to identify similar functions across binaries and compute binary-level similarity. This consists of a greedy strategy to match the call graphs of two binaries, considering both library and user-defined functions. We evaluated *BinSAFE* on a multi-architecture dataset comprising binaries compiled with different compilers and optimization levels. The results demonstrate that *BinSAFE* outperforms a simple baseline, highlighting that combining intra-procedural information from functions with inter-procedural one from call graphs enhances the understanding of binaries' semantics.

Keywords

Binary analysis, neural networks, software security

1. Introduction

Reverse engineering binaries is a critical activity essential for identifying vulnerable and malicious functions when the source code is unavailable. However, this process demands a significant amount of human effort.

Recent research has focused on aiding reverse engineers by developing tools based on Deep Neural Networks (DNNs) designed to reduce the effort required to analyze unknown binaries. Many of these works make extensive use of DNNs for various analysis tasks (e.g., [1, 2, 3, 4, 5, 6, 7]).

Function Similarity. Among the numerous challenges in binary analysis, Function Similarity (FS) Detection [8, 9, 10, 11] is of particular importance to the security community. FS involves determining whether two binary code functions originate from the same source code. This is crucial for security-sensitive scenarios, such as identifying known malware functionalities, detecting vulnerabilities in firmware, and uncovering cases of copyright infringement in compiled binaries [1, 5, 12]. However, this task is non-trivial due to factors like the variety of compiler toolchains, the impact of different optimization levels, and the obfuscation techniques that can drastically change a function's appearance while preserving its original semantics.

Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), February 03-8, 2025, Bologna, IT
[*]Corresponding author.
[†]These authors contributed equally.

capozzi@diag.uniroma1.it (G. Capozzi); laurenzi@diag.uniroma1.it (G. Laurenzi); marco.mormando@gmail.com (M. Mormando); carmine.gianni@cy4gate.com (C. Gianni); gianlucam.marcilli@marina.difesa.it (G. Marcilli); querzoni@diag.com (L. Querzoni); diluna@diag.com (G. D. Luna)

Existing solutions for FS can be broadly divided into two categories: those that employ DNNs [1, 3, 4, 5, 6, 13, 14] and those that do not [10, 15, 16]. Traditional (non-DNN) approaches often rely on symbolic execution or dynamic analysis to capture the semantics of the input. While these methods are generally robust against code obfuscation and compiler optimization, they usually demand considerable analysis effort and long execution times. In contrast, DNN-based solutions are typically more computationally efficient and offer state-of-the-art performance. These approaches leverage embeddings, which are vectors learned by the DNN through the input function's semantics. The similarity between two functions is then computed as the distance between their embeddings in the vector space. This method offers two key advantages over traditional approaches: first, embeddings can be precomputed and stored in a compact format within a knowledge base, enabling much faster similarity computations. Second, unlike hash-based methods, embedding techniques can detect functions with similar semantics even when their binary code differs.

Binary Code Similarity. Despite these advancements, almost all existing works focus on computing similarity between individual functions. In practice, it is essential to develop solutions that operate at the level of entire binaries (e.g. programs or libraries), thus enabling Binary Code Similarity (BCS) Detection. A Binary Code Similarity Detection algorithm would take as input two entire binaries, e.g. two ELF files, and give as output a similarity score between 0 and 1.

Such an algorithm would allow an analyst to build a knowledge base of interesting binaries (such as common libraries or known malicious code) and, when presented with a new firmware image, quickly identify matches of interest between the binaries it contains and those already known.

***BinSAFE* System.** In this paper, we propose an integrated system named *BinSAFE*, which enables the creation of a knowledge base of binaries. New firmware can then be fed into the system, which automatically extracts the binaries within the firmware and matches them against the knowledge base using the BCS algorithm.

The core of *BinSAFE* is a general methodology that extends embedding-based FS solutions to BCS. Our *BinSAFE* BCS Algorithm approach achieves this by matching the call graphs of the target binaries using a greedy strategy that operates in two main phases. First, the system identifies "hotspots" by selecting pairs of similar functions across the two binaries based on a predetermined similarity threshold. Next, it attempts to match the neighbors of these hotspots in the binaries' call graphs, continuing this process until no more viable pairs remain or all neighbors have been examined.

Throughout this process, special care is given to known library functions, incorporating their rarity as a weighting factor. While the use of a common `libc` function (e.g., `printf`) provides limited insight, the presence of an extremely uncommon function could strongly suggest a deeper similarity between the binaries.

In this paper, we propose the following contributions:

- We introduce the architecture of *BinSAFE*, a system that adapts FS solutions for BCS. *BinSAFE* provides all the functionalities required to build a knowledge base of binaries and test new binaries against it.
- We present the *BinSAFE* BCS algorithm, a novel approach that calculates a similarity score between 0 and 1 for two entire binaries, using any embedding-based technique as its building block.
- We provide an experimental evaluation of *BinSAFE* on a dataset of 2118 `amd64` binaries and 1568 `aarch64` binaries, showing that in terms of nDCG it outperforms a naive baseline by 67.56% in the case of `amd64` and 66.03% in the case of `aarch64`. Our evaluation employs *SAFE* [5] as the FS component.

2. The Architecture of *BinSAFE*

The architecture of *BinSAFE* is shown in Figure 1. The system consists of a web GUI that allows an operator to interact with the system, communicating with the back-end via REST APIs. This setup enables the operator to add new binaries to the knowledge base or submit firmware for analysis.

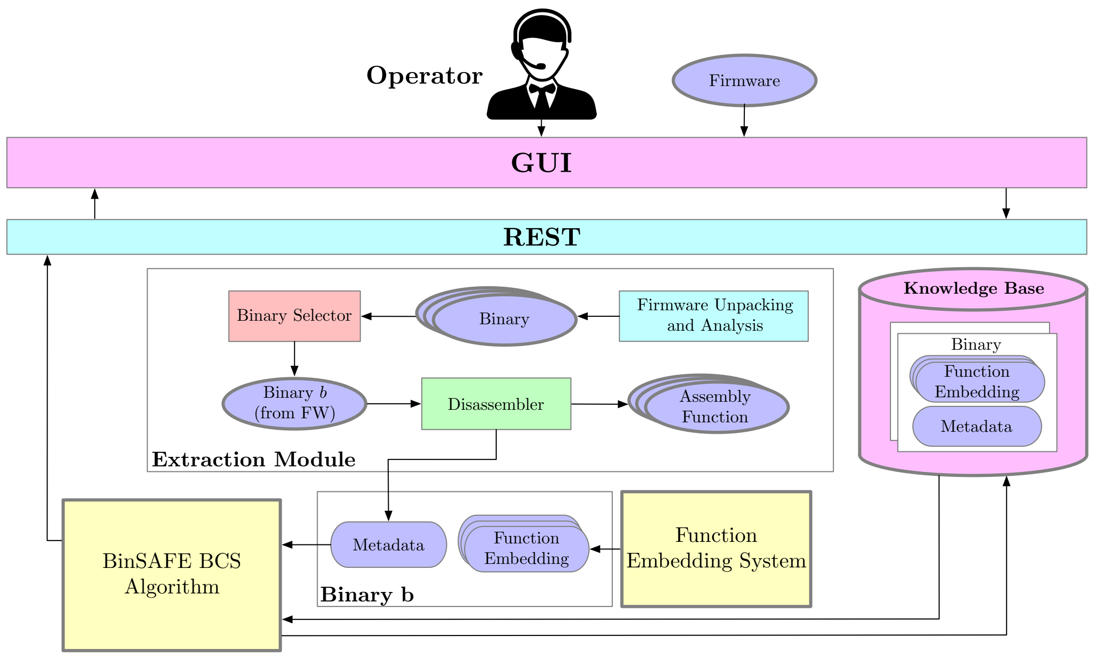

Figure 1: The architecture of *BinSAFE*.

Figure 1 illustrates the execution flow when an analyst submits firmware for analysis. The firmware is unpacked and analyzed to identify executables, which is performed using `binwalk` [17]. Once the binaries are extracted, a binary selector determines which binaries need to be processed by the *BinSAFE* BCS Algorithm. For example, some binaries may be immediately recognized using common hashing techniques. To accommodate this, the system also stores traditional hashes in the knowledge base, allowing *BinSAFE* to instantly recognize binaries previously analyzed.

The *BinSAFE* BCS algorithm processes the binaries to analyze through the following steps: each binary is disassembled to extract the assembly code of its functions and its call graph. The assembly functions are then transformed into embedding vectors using the FS system. While our architecture employs *SAFE* [5] as the embedding system, alternative techniques can also be integrated.

The call graph and function embeddings together form a representation of the binary, referred to as binary b, which is then used by the *BinSAFE* BCS algorithm. The algorithm retrieves relevant binaries from the knowledge base for comparison with binary b. These binaries are stored in the knowledge base in the same representation format. A detailed discussion of the inner workings of the *BinSAFE* BCS Algorithm is provided in the next section.

Finally, the algorithm ranks the relevant binaries in the knowledge base based on their similarity to binary b and returns these results to the operator via the web GUI.

The process of adding a new binary to the knowledge base follows a similar flow up to the computation of the binary's representation. This representation is then directly stored in the knowledge base for future use.

3. BinSAFE BCS Algorithm

This section describes the core component of *BinSAFE*, the graph-matching algorithm that extends embedding-based FS solutions to compute similarity at the binary level.

We represent a binary through its call graph, a pair (F, E), where F is the set of functions in the binary, and E is the set of edges denoting calls between them. The algorithm follows an iterative process to match similar functions across call graphs of different binaries. Specifically, given two call graphs, $b_1 = (F_1, E_1)$ and $b_2 = (F_2, E_2)$, the procedure begins by identifying the pair of functions with the highest similarity score, computed using the FS system. To identify this pair, we generate embeddings for each $f \in F_1 \cup F_2$ using the FS system and then calculate the similarity between the embeddings of F_1 and F_2. In our case, this boils down to multiplying the vectors to compute the cosine similarity.

During each iteration, the algorithm examines the neighbors of the current hotspot in the corresponding call graphs. Each pair of neighbors from b_1 and b_2 is matched, with the pair showing the highest similarity becoming the hotspot for the next iteration. The algorithm then recursively examines and matches the neighbors of the newly matched pairs, extending this process to subsequent neighbors along the two call graphs. This continues until no more matchable pairs remain. At that point, the process is repeated for the next iteration considering the new hotspot.

The procedure terminates when no more functions in b_1 and b_2 can be matched or there are no more hotspot candidates. This process is controlled by a threshold τ. If the maximum similarity falls below this threshold, the process stops, even if unmatched functions remain in the two binaries. The rationale is to avoid matching functions with very low similarity scores. In Appendix B, we discuss the experimental approach used to determine a viable value of τ. The similarity between the two binaries is finally calculated as:

$$sim(b_1, b_2) = avg(sims) * \frac{coverage(b_1) + coverage(b_2)}{2} \tag{1}$$

where: $avg(sims)$ is the average similarity between the matched functions; $coverage(b)$ calculates the percentage of matched functions across the binary b.

3.0.1. Incorporate Library Functions in Binary Matching.

The solution described in the previous section does not explicitly account for library functions, i.e., functions that are imported and linked at runtime. For dynamic linking to work correctly, the symbols of such functions cannot be stripped from the binary. Practically speaking, if an ELF file imports `libc` to use `printf`, the ELF file, even after stripping, retains a symbolic call to `printf`. This allows us to retrieve the name of each imported function and use this information in our matching algorithm. In this approach, imported functions are represented as nodes in the call graphs and are matched when their symbols are identical.

It is widely recognized that common library functions provide limited semantic insight, as they are generic and reused across many programs. In contrast, rare library functions can offer valuable clues about whether binaries implement similar functionalities. To address this, we introduce a mechanism that prioritizes rare library functions over common ones during the initial phase of the algorithm. This mechanism scales the similarity score of library functions using two approaches: Additive Frequency (**AF**), where the similarity of two library functions with the same name is computed as $sim = 1 + (1 - \delta)^2$ and Multiplicative Frequency (**MF**), where the similarity is computed as $sim = 1 \cdot (1 - \delta)^2$. Here, δ is a weight assigned to each function based on its popularity. Additional details on how the δ factor is determined can be found in Section 4.2.

A detailed description of the algorithm together with the pseudocode can be found in Appendix A.

4. Dataset and Implementation

In this section, we describe the evaluation dataset and explain the process for identifying library functions of interest, as well as how to assign them weights based on their popularity.

4.1. Dataset

We evaluated our approaches on a dataset of roughly 2800 binaries generated from 16 open-source C programs and libraries: busybox, coreutils, dnsmasq, dropbear, libboost, libcurl, libgcrypt, liblzma, libpng, libtiff, libzip, lighttpd, nettle, readline, wolfssl, and zlib. These projects were compiled for `amd64` and `aarch64` architectures using four compilers (`clang` 11, `clang` 13, `gcc` 8.3, and `gcc` 10.2) and four optimization levels (O0, O1, O2, O3).

We use ghidra [18] to disassemble binaries and extract their call graphs. After filtering duplicates and handling disassembler errors, the dataset includes 2118 `amd64` binaries and 1568 `aarch64` binaries. Each `amd64` binary has an average of 12 similar binaries (ranging from 1 to 25), while `aarch64` binaries have an average of 12 similar binaries, ranging from 1 to 27.

4.2. Symbols and Frequency of Library Functions

The procedure described in Section 3 involves matching library functions based on their names and requires a frequency measure for each name to adjust the similarity score. Therefore, it is necessary to create a list of symbols extracted from libraries referenced by common binaries. To do so, we analyzed 70,961 `amd64` POSIX-based binaries, focusing on their import tables. Libraries referenced by more than 3% of the analyzed binaries were considered relevant. Through this process, we identified 29 relevant libraries and extracted their symbols. For each identified library, we extracted the function names, resulting in a set of 25,713 functions of interest. We then measured the frequency of each function considering an additional dataset of 4,168 `amd64` POSIX-based binaries. Specifically, for each function name, we counted whether it was called within an analyzed binary. If a function was called multiple times within the same binary, its frequency was still counted as one.

5. Evaluation

In our evaluation, we answer the following research questions:

RQ1: *What is the impact of including libraries on the overall performance of the matching algorithm?*
RQ2: *How does the neighbor-based matching strategy affect the overall performance?*

In the following, we refer to the basic matching strategy described in Section 3.0.1 as **BS-LF**. The main matching strategy based on library names is denoted as **BS-LF-A** when using the **AF** scaling strategy and as **BS-LF-M** when using the **MF** scaling strategy. We test these algorithms against two baselines: **BS-UO**, which runs the algorithm described in Section 3 exclusively on the user-defined functions without considering imported libraries (so that we can measure the impact of considering libraries); **BS-AVG** where the similarity between two binaries is simply computed as the average similarity of the pairwise similarities between all functions in their respective call graphs (in this way we measure the impact of considering topology information).

5.0.1. Setup and Evaluation Metrics.

We test our system in a retrieval scenario. We sample a query binary from the dataset described in Section 4.1 and compute the similarity between the query and all binaries in the dataset. The results are ordered by their similarity scores, and we compute the performance metrics on the top-K results. As performance metrics, we use standard measures for information retrieval systems: nDCG [19], precision, and recall.

In our experiments, for each architecture, we randomly sampled 160 binaries from the dataset described in Section 4.1 to serve as queries (10 for each compiler-optimizer pair) and used the entire set of binaries as the knowledge base.

5.1. RQ1: Impact of Libraries on Matching Algorithm Performance

Here, we evaluate the impact of library functions on the overall performance of our matching algorithm. In particular, we compare **BS-UO**, where library functions are excluded from matching candidates, with **BS-LF**, **BS-LF-A**, and **BS-LF-M**.

5.1.1. Results on amd64.

Looking at the results in Figure 2**(a)** and Table 1, including library functions in the matching strategy produces a noticeable increase in algorithm performance. In particular, when considering the results in average for the search depth value $K \in \{1, 200\}$, **BS-LF** demonstrates a noticeable advantage over **BS-UO**, as it shows a 1.90% higher nDCG, 2.53% higher precision, and a 2.43% higher recall. These gaps slightly increase when considering frequency information of library functions, with **BS-LF-A** outperforming **BS-LF** by 0.47% in nDCG, and 0.43% in recall, while achieving comparable precision. Interestingly, the performance of the matching algorithm varies based on how the similarity of library functions is scaled. Specifically, using frequency as an additive factor leads to better results than applying it as a multiplicative factor, with **BS-LF-A** outperforming **BS-LF-M** by 0.63% in nDCG, 0.63% in precision, and 0.57% in recall on average.

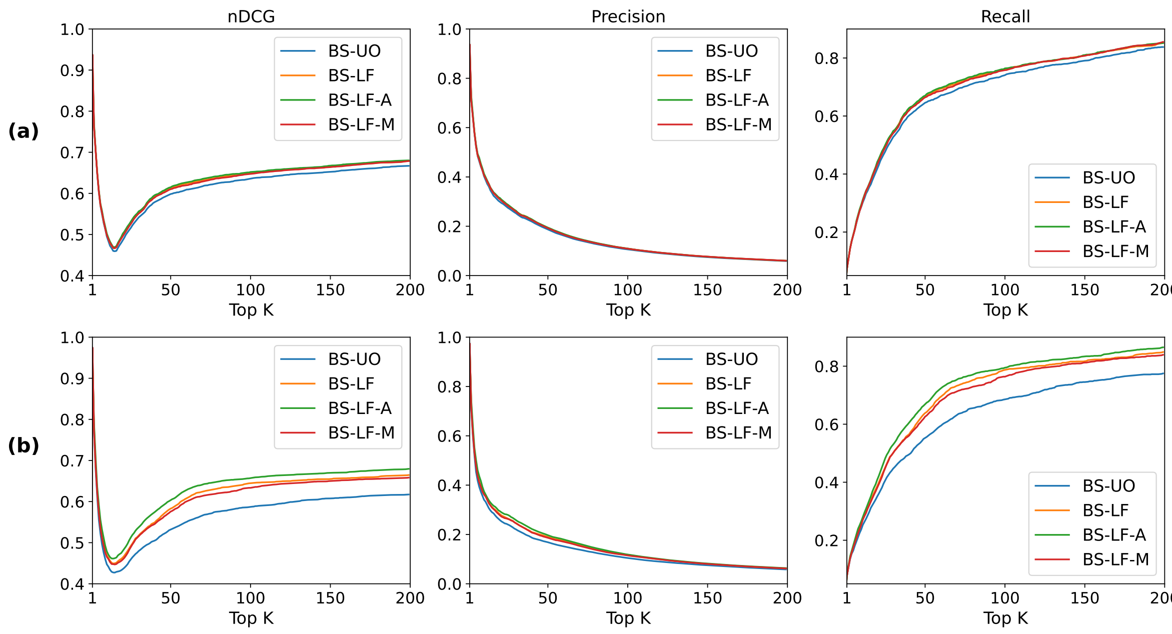

Figure 2: Comparison for architectures `amd64` **(a)** and `aarch64` **(b)** between **BS-UO**, **BS-LF**, **BS-LF-A**, and **BS-LF-M** according to the three considered metrics while varying the search depth $K \in \{1, 200\}$.

The results in Figure 2**(a)** and Table 1 reveal an interesting trend. For low search depth values, **BS-UO** outperforms all other matching algorithms that include library functions across all evaluated metrics. Specifically, when considering the @1 results for the three metrics, while **BS-UO**, **BS-LF**, and **BS-LF-M** perform almost similarly (with **BS-UO** improving by 0.32% **BS-LF-M** nDCG), **BS-UO** demonstrates a clear advantage over **BS-LF-A**, as it achieves a 2.67% higher nDCG, a 2.74% higher precision, and 2.78% higher recall.

This indicates that for low search depth values, considering library functions introduces noise into the matching process. The previous trend is subverted when increasing the search depth, with **BS-LF-A** being the most effective technique, as confirmed by the @10 and @100 results reported in Table 1.

			BS-AVG	BS-UO	BS-LF	BS-LF-A	BS-LF-M
amd64	nDCG	@1	0.068	**0.938**	**0.938**	0.913	0.935
		@10	0.080	0.495	0.499	**0.503**	0.498
		@100	0.223	0.636	0.650	**0.652**	0.647
		avg	0.206	0.620	0.632	**0.635**	0.631
	precision	@1	0.069	**0.938**	**0.938**	0.913	**0.938**
		@10	0.082	0.403	0.408	**0.413**	0.407
		@100	0.057	0.106	0.109	**0.110**	0.109
		avg	0.062	0.154	**0.158**	**0.158**	0.157
	recall	@1	0.004	**0.072**	**0.072**	0.070	**0.072**
		@10	0.054	0.287	0.289	**0.294**	0.289
		@100	0.383	0.741	0.762	**0.764**	0.758
		avg	0.350	0.682	0.699	**0.702**	0.698
aarch64	nDCG	@1	0.143	**0.975**	0.969	0.950	**0.975**
		@10	0.115	0.447	0.469	**0.479**	0.466
		@100	0.230	0.587	0.645	**0.657**	0.633
		avg	0.215	0.570	0.617	**0.633**	0.610
	precision	@1	0.144	**0.975**	0.969	0.950	**0.975**
		@10	0.111	0.341	0.365	**0.374**	0.362
		@100	0.057	0.104	0.118	**0.119**	0.115
		avg	0.064	0.143	0.156	**0.161**	0.154
	recall	@1	0.009	**0.077**	0.076	0.074	**0.077**
		@10	0.067	0.237	0.256	**0.265**	0.254
		@100	0.366	0.686	0.787	**0.795**	0.764
		avg	0.334	0.623	0.698	**0.716**	0.687

Table 1
Comparison between the **BS-AVG** baseline with **BS-UO**, **BS-LF**, **BS-LF-A**, and **BS-LF-M** matching algorithms according to nDCG, precision, and recall metrics, considering 1, 10, and 100 as search depth values.

5.1.2. Results on `aarch64`.

Figure 2**(b)** presents the results for the `aarch64` architecture. These reflect the performance observed for `amd64`, with **BS-LF-A** consistently outperforming all other approaches across all metrics. Specifically, for nDCG, it surpasses **BS-UO** by 9.95%, **BS-LF** by 2.53%, and **BS-LF-M** by 3.63%. In terms of precision, **BS-LF-A** improves upon **BS-UO** by 11.18%, **BS-LF** by 3.11%, and **BS-LF-M** by 4.35%. Finally, for recall, **BS-LF-A** outperforms **BS-UO** by 12.99%, **BS-LF** by 2.51%, and **BS-LF-M** by 4.05%.

The results in Table 1 confirm the trend observed for `amd64`, with **BS-UO** outperforming the other approaches for low search depth values, whereas **BS-LF-A** becomes the most effective method as the search depth increases. Interestingly, the performance gap between our approaches widens significantly when `aarch64` binaries are used as queries.

5.2. RQ2: Impact of Graph Topology

In this section, we examine how inter-procedural information derived from call graph topology contributes to identifying similar binaries. Based on the results of the previous section, we focus exclusively on **BS-LF-A** and its performance against the **BS-AVG** baseline.

Results in Table 1 demonstrate that call-graph information is fundamental in determining whether two binaries are similar. When considering `amd64` binaries, **BS-LF-A** consistently outperforms **BS-AVG**, with an average improvement of 67.56% in nDCG, 60.76% in precision, and 50.14% in recall. Moving to `aarch64`, we observe similar results, with **BS-LF-A** improving **BS-AVG** by 66.03% in nDCG, 60.25% in precision, and 53.35% in recall.

6. Related Works

Existing works on Binary Code Similarity detection can be divided into traditional and learning-based solutions.

Traditional BCS detection methods leverage manually crafted features derived from static or dynamic analysis, combining them using various approaches to compute similarity. Solutions like BinDiff [15] and Genius [20] use semantic features to represent CFG nodes and apply graph-matching algorithms to compute similarity. TEDEM [16] measures function similarity by calculating the edit distance between CFG nodes represented as expression trees. Some methods avoid graph matching; for instance, Tracelet [21] calculates function similarity through the edit distance between instruction sequences, while [9] uses a program verifier to assess the similarity between basic block slices (strands) before lifting the results to functions.

Learning-based approaches harness recent advancements in NLP and Graph Neural Networks to generate low-dimensional representations (i.e., embeddings) of the input that capture the semantics of code snippets. Similarity is then measured by calculating the distance between these vectors. During training, the parameters of the DNN model are adjusted so that embeddings of semantically similar snippets are positioned close in the vector space. *SAFE* [5] proposes an RNN that generates function embeddings starting from the linear disassembly, treating instructions as tokens. More recent approaches, like jTrans [6], TREX [13], and BinBert[14] propose a Transformer-based architecture to learn function semantics from instruction sequences that explicitly represent the execution trace. Gemini [1] consists of a GNN for learning the attributed control-flow graph of binary functions, where basic blocks are encoded using manually-selected features. Finally, DeepBinDiff [4], combines an NLP-based methodology for learning function semantics with a graph-matching algorithm for computing similarity at the binary level.

7. Limitations

In this paper, we have demonstrated how combining function embedding techniques, particularly *SAFE*, with information from the call graph can yield valuable results in measuring program-level similarity.

Our *BinSAFE* system is designed for use in security-sensitive scenarios. In this context, it is crucial to understand its behavior when analyzing obfuscated code or when facing adversaries capable of crafting binaries specifically to mislead the *BinSAFE* system (i.e., adversarial examples). The robustness of our system is closely tied to the underlying FS system, which is responsible for modeling the semantics of binary functions within the analyzed binary. Since *SAFE* was not trained to handle obfuscated code, this presents a clear limitation for the entire *BinSAFE* system, as well as its vulnerability to adversarial examples, as demonstrated in [22, 23].

8. Conclusions and Future Works

In this paper, we presented our *BinSAFE* system. Our main contribution is a technique to extend FS similarity systems to BCS. In this paper, we tested our system using *SAFE* [5] as the FS model. However, our approach can be used as it is with other FS embedding-based models. Future work would include a more extensive evaluation of *BinSAFE* using different FS models to assess their impact.

Acknowledgment. This work has been supported by the *"Piano Nazionale della Ricerca Militare"* (PNRM) through the SAFE-ES project which aims to protect the supply chain of electronic devices in the ICS environment, within which *BinSAFE* was developed.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT, Grammarly in order to: Grammar and spelling check, Paraphrase and reword. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] 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 24th ACM SIGSAC Conference on Computer and Communications Security (CCS '17), 2017, pp. 363–376.

[2] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, G. Stringhini, Mamadroid: Detecting android malware by building markov chains of behavioral models, in: Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS '17), The Internet Society, 2017.

[3] Y. Li, C. Gu, T. Dullien, O. Vinyals, P. Kohli, Graph matching networks for learning the similarity of graph structured objects, in: Proceedings of the 36th International Conference on Machine Learning (ICML '19), 2019, pp. 3835–3845.

[4] Y. Duan, X. Li, J. Wang, H. Yin, Deepbindiff: Learning program-wide code representations for binary diffing, in: Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS '20), The Internet Society, 2020.

[5] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, R. Baldoni, Function Representations for Binary Similarity, IEEE Transactions on Dependable and Secure Computing 19 (2022) 2259–2273.

[6] 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 (ISSTA'22), ACM, 2022, pp. 1–13.

[7] K. Pei, W. Li, Q. Jin, S. Liu, S. Geng, L. Cavallaro, J. Yang, S. Jana, Exploiting code symmetries for learning program semantics, in: Proocedings of the 41st International Conference on Machine Learning (ICML '24), 2024.

[8] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code, Digital Investigation 12 (2015) S61–S71.

[9] Y. David, N. Partush, E. Yahav, Statistical similarity of binaries, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16), 2016, pp. 266–280.

[10] T. Dullien, R. Rolles, Graph-based comparison of executable objects (English version), in: Proceedings of the Symposium sur la sécurité des technologies de l'information et des communications (SSTIC '05), 2005, p. 3.

[11] W. M. Khoo, A. Mycroft, R. Anderson, Rendezvous: A search engine for binary code, in: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13), 2013, pp. 329–338.

[12] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, D. Balzarotti, How Machine Learning Is Solving the Binary Function Similarity Problem, in: Proceedings of the 31st USENIX Security Symposium (SEC '22), USENIX Association, 2022, pp. 2099–2116.

[13] K. Pei, Z. Xuan, J. Yang, S. Jana, B. Ray, Learning approximate execution semantics from traces for binary function similarity, IEEE Transactions on Software Engineering 49 (2023) 2776–2790.

[14] F. Artuso, M. Mormando, G. A. Di Luna, L. Querzoni, Binbert: Binary code understanding with a fine-tunable and execution-aware transformer, IEEE Transactions on Dependable and Secure Computing (2024) 1–18.

[15] Zynamics, Zynamics BinDiff, 2021. URL: https://www.zynamics.com/software.html.

[16] 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 (ACSAC '14), 2014, pp. 406–415.

[17] ReFirmLabs, Binwalk, https://github.com/ReFirmLabs/binwalk, 2024.
[18] National Security Agency (NSA), Ghidra Software Reverse Engineering Framework, https://ghidra-sre.org, 2024. Last Accessed: 2024-11-29.
[19] Y. Wang, L. Wang, Y. Li, D. He, T. Liu, A theoretical analysis of NDCG type ranking measures, in: Proceedings of the 26th Annual Conference on Learning Theory (COLT '13), volume 30, 2013, pp. 25–54.
[20] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, H. Yin, Scalable graph-based bug search for firmware images, in: Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS '16), ACM, 2016, pp. 480–491.
[21] Y. David, E. Yahav, Tracelet-based code search in executables, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14), 2014, pp. 349–360.
[22] G. Capozzi, D. C. D'Elia, G. A. D. Luna, L. Querzoni, Adversarial attacks against binary similarity systems, IEEE Access 12 (2024) 161247–161269.
[23] G. Capozzi, T. Tang, J. Wan, Z. Yang, D. C. D'Elia, G. A. D. Luna, L. Cavallaro, L. Querzoni, On the lack of robustness of binary function similarity systems, CoRR abs/2412.04163 (2024). arXiv:2412.04163.

Appendices

A. *BinSAFE* BCS Algorithm

We present our matching procedure in Algorithm 1. Here, we begin by computing the similarity matrix (line 2), calculating the pairwise similarity between the functions in F_1 and F_2 using *SAFE*. Then, we match by name library functions, setting their similarity to 1 or 0 and scaling it by frequency. At this point, every pair of matched library functions is considered a hotspot, so the procedure tries to recursively match all the matchable neighbors of the pairs identified before (lines 3-8). After this first phase, we now start matching the user-defined functions (line 9) by iteratively determining the next hotspot pair and matching the subsequent pairs along the two graphs (lines 11-16). Finally, we identify the percentage of matched functions in b_1 and b_2 (line 17) and calculate the similarity (line 18) using Equation 1.

B. Threshold Tuning

An important aspect of our approach is deciding which function pairs from the two call graphs to match. The strategies introduced in Section 3 pair only those functions from the two binaries with a similarity score above a specified threshold τ, which is determined through experimental evaluation.

In these experiments, we perform 181 queries on a knowledge base containing 5,256 binaries and libraries. Specifically, the knowledge base includes binaries from the following projects: busybox, coreutils, dnsmasq, dropbear, libBoost, lighttpd, libzip, readline, and zlib. These binaries were compiled for the `amd64` architecture using `gcc` 8.3, `gcc` 10.2, `clang` 11, and `clang` 13.

The results of these experiments are presented in Figure 3, where we evaluate the performance of **BS-LF** using three threshold values for τ: 0, 0.50, and 0.75. All metrics indicate a significant difference between $\tau = 0$ and $\tau \in \{0, 50, 0.75\}$. Specifically, considering nDCG, there is an average increase of 1.49% from 0 to 0.50 and 1.78% from 0 to 0.75, with a smaller improvement of 0.3% between 0.50 and 0.75. A similar trend is observed for precision and recall: precision improves by 1.61% from 0 to 0.50 and 2.24% from 0 to 0.75, while recall increases by 2.3% from 0 to 0.50 and 3.41% from 0 to 0.75.

For our experiments, we choose the value $\tau = 0.75$.

Algorithm 1 Matching algorithm for computing similarity at the binary level

Input:

- Call graphs: $b_1 = (F_1, E_1)$ and $b_2 = (F_2, E_2)$.
- Library functions: $L_1 \subset F_1$ and $L_2 \subset F_2$.
- Similarity threshold: τ.

Output: Similarity value between b_1 and b_2.

Definitions:

- getSimilarities(F_1, L_1, F_2, L_2): Compute the pairwise similarity between functions in F_1 and F_2 using SAFE. Match library functions by name in L_1 and L_2, setting similarity to 1 or 0, then scale by frequency. Set similarity between library and user-defined functions to 0. Return matched library function pairs and the similarity matrix.
- getNeighbors(b, f): Get neighbors of f in the call graph b.
- matchNeighbors$(n_1, n_2,$ m, sim_matrix$)$: Match recursively functions in n_1 and n_2 if their similarity is above τ and if they have not already been matched in m.
- getUnmatched(F, m): Get the functions in F that are not part of any match in m.
- getHotspot(u_1, u_2, sim_matrix, m): Get the pair $\langle f, g\rangle \in u_1 \times u_2$ with highest similarity value in sim_matrix among those not in m.
- getCoverage(b_1, b_2, matches): Get the percentage of functions across b_1 and b_2 that are in m.
- computeSim(sim_matrix, m_1, m_2): Compute the similarity according to Equation 1.

```
1: matches ← set()
2: match_libs, sim_matrix ← getSimilarities($F_1$, $L_1$, $F_2$, $L_2$)
3: for $\langle f, g\rangle \in$ match_libs do
4:     sim_score ← sim_matrix[$f$][$g$]
5:     matches.add($f$, $g$, sim_score)
6:     $n_1, n_2$ ← getNeighbors($b_1$, $f$), getNeighbors($b_2$, $g$)
7:     if sim_score $\geq \tau$ AND $\langle f, g\rangle \notin$ matches then
8:         matches ← matchNeighbors($n_1$, $n_2$, matches)
9: $u_1, u_2$ ← getUnmatched($F_1$, matches), getUnmatched($F_2$, matches)
10: while $u_1 \neq \emptyset$ AND $u_2 \neq \emptyset$ AND $\exists \langle f, g\rangle \in u_1 \times u_2$ s.t. sim_matrix[$f$][$g$] $\geq \tau$ do
11:     $f$, $g$, sim_score ← getHotspot($u_1$, $u_2$, sim_matrix, matches)
12:     matches.add($f$, $g$, sim_score)
13:     $n_1, n_2$ ← getNeighbors($b_1$, $f_1$), getNeighbors($b_2$, $f_2$)
14:     matches ← matchNeighbors($n_1$, $n_2$, matches)
15:     $u_1$.remove($f_1$)
16:     $u_2$.remove($f_2$)
17: $m_1, m_2$ ← getCoverage($b_1$, $b_2$, matches)
18: return computeSim(sim_matrix, $m_1$, $m_2$)
```

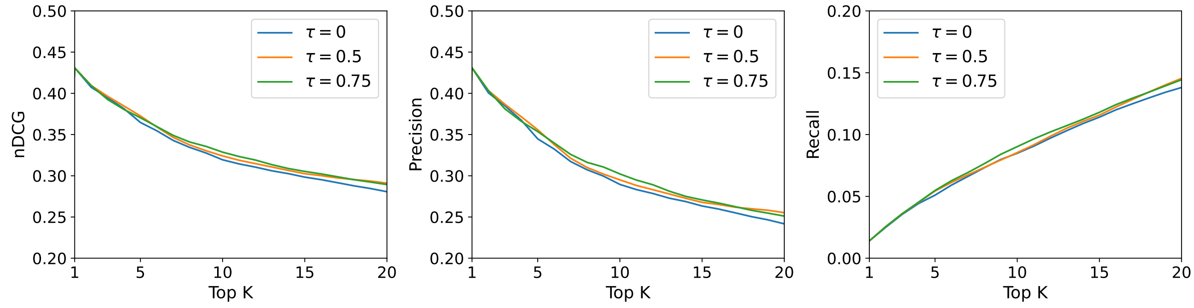

Figure 3: Evaluation of ***SAFE*-LF** considering $\tau \in \{0, 0.50, 0.75\}$. The metrics are evaluated considering the search depth parameter $K \in \{1, 20\}$.