

ReGraph: A Tool for Binary Similarity Identification

Li Zhou
KAUST
KSA
li.zhou@kaust.edu.sa

Marc Dacier
KAUST
KSA
marc.dacier@kaust.edu.sa

Charalambos Konstantinou
KAUST
KSA
charalambos.konstantinou@kaust.edu.sa

Abstract

Binary Code Similarity Detection (BCSD) is not only essential for security tasks such as vulnerability identification but also for code copying detection, yet it remains challenging due to binary stripping and diverse compilation environments. Existing methods tend to adopt increasingly complex neural networks for better accuracy performance. The computation time increases with the complexity. Even with powerful GPUs, the treatment of large-scale software becomes time-consuming. To address these issues, we present a framework called ReGraph to efficiently compare binary code functions across architectures and optimization levels. Our evaluation with public datasets highlights that ReGraph exhibits a significant speed advantage, performing 700 times faster than Natural Language Processing (NLP)-based methods while maintaining comparable accuracy results with respect to the state-of-the-art models.

CCS Concepts

• **Security and privacy** → *Malware and its mitigation; Embedded systems security; Software reverse engineering.*

Keywords

Binary Code Similarity Detection, Code Property Graph, Graph Neural Network, Code Lifting, Binary Code Re-Optimization.

ACM Reference Format:

Li Zhou, Marc Dacier, and Charalambos Konstantinou. 2025. ReGraph: A Tool for Binary Similarity Identification. In *34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA Companion '25)*, June 25–28, 2025, Trondheim, Norway. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3713081.3731728>

1 Introduction

Binary Code Similarity Detection (BCSD) plays a critical role in various downstream applications such as vulnerability identification, firmware security analysis, and software reuse detection [4, 15, 20, 27, 37]. Despite significant progress, it remains an active research area. A key difficulty lies in that, even when compiled from the same source code, binaries can exhibit substantial differences across various compilation options in both structure and semantics. Furthermore, to minimize file size, binary stripping is commonly employed [22], eliminating symbol tables and debug information [11], thereby exacerbating the challenge of binary-level code analysis [29].

Manually reverse engineering binary code functions is a solution but a very time-consuming one, especially when analyzing millions of stripped binary functions [32]. Hence, research directions focus on automatic methods. At first, methods have compared binary file elements such as operator counts, jumps, and Control Flow Graphs (CFGs) [8, 30], but different compilation options significantly alter CFGs and other extracted features [23], limiting their performance in cross-optimization and cross-architecture scenarios [22, 25]. More recently, new methods have been proposed to leverage neural networks and have outperformed earlier methods [31, 35, 36]. Despite that, the differences caused by different compilation options require an increasing amount of neural network parameters [5, 14, 31], as well as the creation of large models [9, 21, 36]. These models require powerful and expensive GPUs and render the computation time impractical when analyzing large amounts of functions.

To address these challenges, we propose ReGraph, a function-level cross-architecture and cross-compilation binary code matching framework. ReGraph efficiently identifies known functions within a given software by leveraging code lifting, re-optimization, Code Property Graphs (CPGs), and Graph Neural Networks (GNNs) to compute function similarity. This enables analysts to further examine the most likely similar functions within a binary executable. We evaluate ReGraph on public datasets [22]. The results demonstrate that our framework exhibits high accuracy and fast matching speed with much lower resource consumption than state-of-the-art solutions. Our contributions can be summarized as follows:

- We propose mitigating binary discrepancies by offloading them to the lifter and re-optimizer. Our experiments show that these techniques improve similarity scores by an average of 72.8% with respect to those reported by BinDiff [10]. Our results demonstrate that these techniques can serve as a generalizable meta-method for other binary code similarity detection frameworks.
- We integrate CPGs and GNNs with lifting and re-optimization to develop ReGraph. ReGraph achieves a 700× speedup over NLP-based methods while maintaining comparable performance to state-of-the-art approaches.
- With ReGraph, our proposed command-line tool, analysts can focus on the top-K most similar functions rather than analyzing the entire binary file, significantly reducing their workload.

This paper is structured as follows: Section II presents the usage of our tool, and Section III provides a detailed explanation of each component of ReGraph. We evaluate ReGraph in Section IV. Section V concludes the paper and discusses potential directions for future research.

ISSTA Companion '25, June 25–28, 2025, Trondheim, Norway

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA Companion '25)*, June 25–28, 2025, Trondheim, Norway, <https://doi.org/10.1145/3713081.3731728>.

2 Usage

In this section, we demonstrate the usage of ReGraph, A Python-based command-line tool that detects similar binary code functions. It operates in two phases: the training phase, which processes newly encountered binary files, and the inference phase, which computes similarity results based on the trained model.

2.1 Requirements

ReGraph is built in Python with additional dependencies. It requires RetDec [17] for binary lifting, LLVM [18] for re-optimization, and Joern [33] for extracting CPGs. These dependencies should be installed before using ReGraph.

2.2 Training

Preprocessing: Preprocessing includes binary lifting, re-optimization, CPG extraction, and CPG vectorization. We have integrated the first three functionalities into a Python script, allowing them to be executed with a single command. By default, we assume the directory structure follows the format `project/architecture/optimization-level`. If not, users can specify custom file paths through the extra-provided Python programming interface. Once CPGs are extracted, a separate command-line script vectorize the CPGs into datasets for training and inference. Additionally, an auxiliary file, the `op_file`, will be generated to store statistics about operators extracted from binaries to assist in encoding the CPGs.

Training: To achieve optimal performance on new binary files, we recommend training the model on these files. All hyperparameters governing the training process are defined within the configuration file called `train_config.yaml`. Subsequent to data preprocessing, this file requires modification to specify the dataset directory path and finalize the parameters for training execution. Then, running the model training script will initiate the training process and generate the trained model file in the specified output folder.

2.3 Inference

With the trained model file and the corresponding `op_file`, inference can be performed. Given a target function in a binary file (target binary file), our tool identifies the top K's most similar functions, in which the user pre-defines the K, from another binary file (candidate binary file). To minimize user interaction, our tool outputs all functions from the target binary file along with their top-K matches from the candidate binary file, including the corresponding similarity scores, rather than requiring users to select a target function manually. This functionality is encapsulated in a script, where users specify the pre-trained model path, the `op_file`, and the two binaries via the command line to generate the results in the Excel spreadsheet.

2.4 Illustrative Example

In this section, we present an example using two files from OpenPLC (a collection of open-source programs for Programmable Logic Controllers, PLCs) [3], compiled under different environments. The target binary is compiled for the x86 architecture with `-O0` optimization, while the candidate binary is compiled for the ARM architecture with `-O3` optimization. We use the function `__time_sub` as

a case study. Despite originating from the same source code, their corresponding assembly code demonstrates significant differences, as shown in Figure 1.

MOV R12, R0	push ebp
SUB SP, SP, #8	mov ebp, esp
PUSH {R4, LR}	sub esp, 28h
...	...
ADD SP, SP, #8	leave
BX LR	ret 4

Figure 1: The assembly code of example functions compiled in ARM -O3 (left) and X86 -O0 (right).

After training ReGraph and utilizing the pre-trained model, we set $K = 5$. Table 1 presents the example output of functions similar to `__time_sub`. In stripped binaries, function names are typically removed. In such cases, security analysts can inspect the top-K functions to identify the target function, rather than searching through the entire binary, which significantly reduces the workload and improves efficiency in binary analysis. To aid in understanding the results, we provide the function names in the last column, labeled *Real Name*. Due to minor operator differences between `__time_sub` and `__time_add`, their similarity scores are close. In contrast, `INTEGRAL_body__` exhibits more structural differences from `__time_sub`, resulting in a lower similarity score.

Top K	Score	Stripped Name	Real Name
1	0.921	function_154	<code>__time_sub</code>
2	0.917	function_104	<code>__time_add</code>
3	0.809	function_2cc8	<code>INTEGRAL_body__</code>
4	0.806	function_35b8	<code>PROG0_body__</code>
5	0.797	function_74	<code>__normalize_timespec</code>

Table 1: Top K functions with corresponding scores and addresses.

3 ReGraph

In this section, we detail each component of ReGraph, whose structure is depicted in Figure 2. Given a target binary code function, our objective is to measure its similarity score with each function of a test set via their corresponding CPGs. ReGraph first uses a lifter to lift the binary code function to LLVM Intermediate Representation (IR) [18]. Then, the optimizer further re-optimizes it to the `-O3` level, representing the highest optimization level commonly used in compilation.

However, when the lifter and optimizer attempt to reorganize the binary code, in some cases, differences persist. Hence, we employ CPG and GNN to tolerate these differences. Once we obtain the optimized LLVM IR, we decompile it into pseudo-C code and convert it into a CPG using Joern [33]. We then encode the CPG into the GNN to obtain its corresponding function vector that represents the whole function. The similarity score is finally computed using the *Pearson coefficient* [24] applied to their respective function vectors. A higher coefficient score indicates a greater similarity between the two functions. In the rest of this section, we detail the design choices for each component.

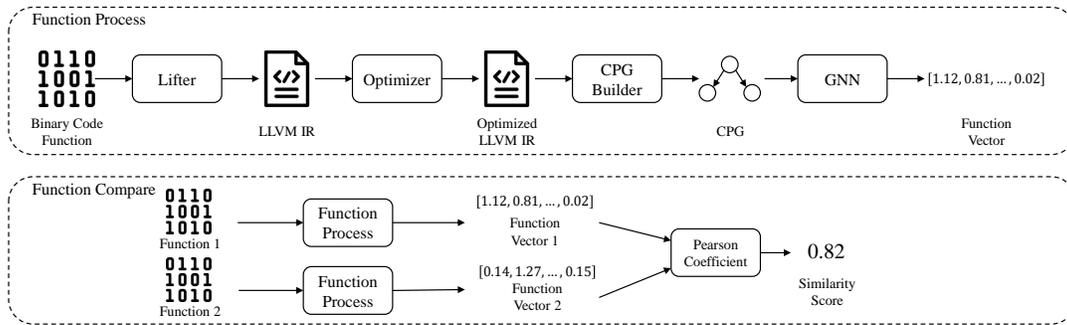


Figure 2: The structure of ReGraph.

3.1 Lifter

Decompilers such as IDA Pro [13] and Ghidra [1] are widely used in reverse engineering. These decompilers maintain their own IR systems as fundamental platforms supporting their decompilation analysis frameworks, which lift all binary files to intermediate levels where subsequent optimizations and analyses are performed. The IR system decouples program logic analysis from specific CPU architectures, eliminating the need to implement program analysis and optimization methods for each architecture. However, the IR generated by decompilers is primarily intended to produce more readable pseudo-C code rather than implement the optimizations typically found in compilers. Therefore, we choose to use another lifter, RecDec [17], that lifts binary codes from diverse architectures into standardized LLVM IR, thereby reducing differences caused by varying architectures.

3.2 Re-optimization

Different software vendors employ different optimization options to meet their specific requirements. According to the documentation of GCC [12] and Clang [6], the relationship between different optimization levels is hierarchical. For instance, -O2 includes all optimization options from -O1 and adds its own additional optimizations. -O3 encompasses all optimization options from -O2 and represents the highest optimization level. Applying -O3 to code that has already been optimized at lower levels optimizes the code to its maximum potential. Conversely, applying lower optimization levels to code undergoing -O3 optimization does not yield additional changes, as -O3 has already implemented all available optimizations. Therefore, if we can re-optimize the program to the highest optimization level, the difference arising from different optimization options would be mitigated [7]. We utilize the built-in toolkit of LLVM [18] to re-optimize the LLVM IR obtained after lifting the binary.

3.3 CPG

Compilers generate a code’s Abstract Syntax Tree (AST) by parsing the programming language, followed by semantic validation. The AST serves as the fundamental structure for representing code at a low level and captures more semantic information than CFGs [34]. CPG is an enhanced version of the AST. It explicitly incorporates information such as data dependencies and program dependencies,

providing a richer semantic representation than the AST alone. With the CPG in hand, we can infer the corresponding code’s syntax and semantic information in a uniform graph representation. After decompiling the re-optimized LLVM IR into pseudo-C code, we use Joern [33] to transform C code into CPG [2].

3.4 Graph Neural Network

The process of lifting, code re-optimization, and CPG reconstruction does not always produce identical graphs for similar functions. Some structural differences can still exist, and some information remains unrecoverable, leading to variations in the control flow of the codes. These differences are reflected in the graphs as structural discrepancies. Hence, we need a graph-matching algorithm that tolerates partial structural differences based on semantic relationships, enabling the determination of graph similarity between two semantically similar functions even when there are differences in their graph structures. As GNNs comprehensively consider not only the graph topology but also global information based on the features of nodes and edges [19, 28], we incorporate semantic information into the node features of CPGs and train a GNN based framework capable of evaluating the similarity between two CPGs.

4 Evaluation

In this section, we evaluate ReGraph on open-source software [3] and public datasets [22], we aim to answer the following evaluation questions (EQs):

- (1) **EQ 1:** Can lifters and optimizers reduce differences between binary code functions from different compilation environments?
- (2) **EQ 2:** Compared with state-of-the-art methods, how much speedup can ReGraph achieve while keeping similar performance?

4.1 BinDiff Enhancement

To answer **EQ 1**, we choose to apply lifting and re-optimization on binary files to be compared among each other with the BinDiff [10] tool, a widely used open-source framework that computes structural similarity scores between functions in binary executables. This enables us to show substantial improvement in the results. To do this, we carry out the following experiment using the OpenPLC dataset [3]. We take the default example ladder logic file from

Table 2: The similarity score reported by BinDiff [10] before and after applying lifting, re-optimization and re-compilation

		O3																	
		ARM			PowerPC			MIPS			X86			AVG					
		Before	After	Inc	Before	After	Inc	Before	After	Inc	Before	After	Inc	Before	After	Inc			
O0	ARM	0.239	0.676	183%	0.415	0.657	58%	0.346	0.672	94%	0.386	0.628	63%	0.347	0.658	90%			
	PowerPC	0.233	0.573	146%	0.373	0.622	67%	0.348	0.578	66%	0.376	0.555	48%	0.333	0.582	75%			
	MIPS	0.258	0.591	129%	0.435	0.601	38%	0.532	0.672	26%	0.462	0.596	29%	0.422	0.615	46%			
	X86	0.233	0.626	169%	0.415	0.641	54%	0.347	0.697	101%	0.400	0.632	58%	0.349	0.649	86%			
	AVG	0.241	0.617	157%	0.410	0.630	54%	0.393	0.655	72%	0.406	0.603	49%	0.362	0.626	74%			

OpenPLC Editor [16] as the input and compile it into binary files for four different architectures: MIPS, X86, PowerPC, and ARM. We use two extreme optimization levels: -O0 for no optimization and -O3 for the highest optimization. We then use the BinDiff [10] and IDA Pro [13] to have the similarity score for the same function but from binaries with different compilation environments. In the BinDiff, the similarity score ranges from 0 to 1.

To simulate conditions involving cross-architecture and cross-optimization option scenarios, a function that originates from -O0 of a certain architecture is compared with the same name functions from all architectures using the -O3. Then, we extract the corresponding similarity scores from BinDiff, and calculate the average similarity score from all the functions from that binary as the total result for one compilation environment versus another compilation environment. Next, we lift, re-optimize, and recompile the binary files using the x86-O3 setup. Subsequently, we repeat the same process to observe changes in the similarity scores.

The results are displayed in Table 2 to provide a clear visual representation. The percentage increase in the similarity score for each comparison is presented in the increase (Inc.) column. Furthermore, the average similarity scores for each column and row are calculated and presented in the last row and column, respectively. From the results, we conclude our answer to the EQ 1.

Answer to EQ 1: The similarity scores reported by BinDiff show a significant increase across all comparisons by 74% on average, highlighting the effectiveness of our method in mitigating differences between optimization levels and distinct architectures.

Since BinDiff relies on graph matching and some functions cannot be fully re-optimized, some differences remain between graphs. Despite major improvement, those differences prevent the similarity scores from reaching 1 with BinDiff. Moreover, we noticed that BinDiff performs very poorly when comparing stripped binaries with non-stripped ones.

4.2 Public Dataset Evaluation

We evaluate ReGraph with the state-of-the-art models using the identical dataset proposed by [22] to answer EQ 2. Our experimental platform is a general-purpose PC with an Intel i7-12700F CPU and an RTX 2080 Ti GPU. We align the exact GPU specifications and other experiment conditions as defined in [22]. We separately train and evaluate our model on two distinct, non-overlapping datasets

	Zeek on GPU	Gemini on GPU	Asm2Vec on GPU	GMN on GPU	ReGraph on CPU	ReGraph on GPU
Recall@1	0.21	0.33	0.18	0.54	0.65	0.65
Time secs/100 functions	0.091	0.135	7.17	0.876	0.283	0.011

Table 3: Perf. comparison with SotA methods from [22].

from [22] to simulate scenarios where the model encounters unseen binary files. From [22], Recall@1 is used to evaluate the accuracy of a model’s first prediction, while secs/100 functions measures its inference speed. We run ReGraph on CPU and GPU separately. Table 3 shows our result compared to other models in [22]. Given that preprocessing, lifting, and re-optimization utilize external tools outside the scope of our control, we restrict our evaluation to model inference time, which is consistent with that used in [22]. From the results, we conclude our answer to the EQ 2.

Answer to EQ 2: While maintaining high accuracy as measured by the Recall@1 metric, ReGraph achieves a speedup of over 700× compared to the NLP-based model Asm2Vec [9] and 9× compared to Zeek [26]. Moreover, it performs efficiently even when running solely on a CPU.

5 Conclusion and Future Work

In this paper, we propose ReGraph, a BCSD framework that effectively handles variations introduced by different compilation environments. ReGraph mitigates discrepancies caused by different compilation options by leveraging lifters and re-optimizers. With a lightweight GNN model, our experiments empirically demonstrate that ReGraph can yield promising results compared with those state-of-the-art solutions but with considerably less resource consumption. Furthermore, our experiment demonstrates that lifting and re-optimization can serve as a meta-method to enhance the performance of existing BCSD tools. We hope that our tool will inspire new perspectives in binary code analysis. In the future, we plan to extend our framework to the snippet level to enhance robustness.

Tool Availability

We open-source ReGraph and pre-trained models on GitHub: <https://github.com/damao000/ReGraph>. Additionally, our demo video can be seen on YouTube: <https://youtu.be/5CSJkZh89hs>.

References

- [1] NationalSecurity Agency. 2019. NationalSecurityAgency/Ghidra: Ghidra is a software reverse engineering (SRE) framework. <https://github.com/NationalSecurityAgency/ghidra>
- [2] Fabian Yamaguchi Alex Denisov. 2021. LLVM meets code property graphs. <https://blog.llvm.org/posts/2021-02-23-llvm-meets-code-property-graphs/>
- [3] Thiago Rodrigues Alves, Mario Buratto, Flavio Mauricio De Souza, and Thelma Virginia Rodrigues. 2014. OpenPLC: An open source alternative to automation. In *IEEE Global Humanitarian Technology Conference (GHTC 2014)*. IEEE, 585–589.
- [4] Hamid Abdul Basit and Stan Jarzabek. 2005. Detecting higher-level similarity patterns in programs. *ACM Sigsoft Software engineering notes* 30, 5 (2005), 156–165.
- [5] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 678–689.
- [6] TI Clang. 2024. 1.3.7. optimization options. https://software-dl.ti.com/codegen/docs/tiarmclang/compiler_tools_user_guide/compiler_manual/using_compiler/compiler_options/optimization_options.html
- [7] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. 79–94.
- [8] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360.
- [9] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. 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, 472–489.
- [10] GmbH. 2011. <https://www.zynamics.com/bindiff.html>
- [11] GNU. [n. d.]. https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_node/binutils_10.html
- [12] GCC GNU. 2023. Options That Control Optimization. <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>
- [13] Rays Hex. 1996. Ida Pro. <https://hex-rays.com/ida-pro/>
- [14] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. Cross-architecture binary semantics understanding via similar code comparison. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 57–67.
- [15] He Huang, Amr M Youssef, and Mourad Debbabi. 2017. Binsequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. 155–166.
- [16] Página inicial. 2022. <https://openplcproject.com/docs/3-1-openplc-editor-overview/>
- [17] Jakub Křoustek, Peter Matula, and P Zemek. 2017. Retdec: An open-source machine-code decompiler. In *July 2018*.
- [18] Chris Arthur Lattner. 2002. LLVM: An infrastructure for multi-stage optimization. (2002).
- [19] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*. PMLR, 3835–3845.
- [20] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 389–400.
- [21] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search. In *NDSS*.
- [22] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*. 2099–2116.
- [23] Elie Mengin and Fabrice Rossi. 2021. Binary diffing as a network alignment problem via belief propagation. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 967–978.
- [24] Iain Pardoe, Laura Simon, and Derek Young. 2018. 2.6 - (Pearson) Correlation Coefficient r. <https://online.stat.psu.edu/stat462/node/96/> Accessed: 2025-03-11.
- [25] Jannik Powny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 709–724.
- [26] Noam Shalev and Nimrod Partush. 2018. Binary similarity detection using machine learning. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*. 42–47.
- [27] Paria Shirani, Leo Collard, Basile L Agba, Bernard Lebel, Mourad Debbabi, Lingyu Wang, and Aiman Hanna. 2018. B in a rm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, June 28–29, 2018, Proceedings 15*. Springer, 114–138.
- [28] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).
- [29] Yongpan Wang, Hong Li, Xiaojie Zhu, Siyuan Li, Chaopeng Dong, Shouguo Yang, and Kangyuan Qin. 2024. BinEnhance: An Enhancement Framework Based on External Environment Semantics for Binary Code Search. *arXiv preprint arXiv:2411.01102* (2024).
- [30] xorpd. [n. d.]. FCATALOG. <https://www.xorpd.net/pages/fcatalog.html>
- [31] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. 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*. 363–376.
- [32] Zimu Xu, Maria H Gonzalez-Serrano, Rocco Porreca, and Paul Jones. 2021. Innovative sports-embedded gambling promotion: A study of spectators' enjoyment and gambling intention during XFL games. *Journal of Business Research* 131 (2021), 206–216.
- [33] Yamaguchi. 2014. The bug hunter's workbench. <https://joern.io/>
- [34] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. 2021. 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, 224–236.
- [35] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 1145–1152.
- [36] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. Codemr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems* 33 (2020), 3872–3883.
- [37] Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, et al. 2022. A large-scale empirical analysis of the vulnerabilities introduced by third-party components in IoT firmware. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 442–454.