"Understanding the AI-powered Binary Code Similarity Analysis"

We open-source the artifacts to support the research community for further research. Specifically, we provide entire datasets, benchmarks, and implementation details, including the improvement/reimplementation of existing BinSD approaches, the selection of hyper-parameters, and the generation of function embedding.

Contents

0.1 Mainstream Embedding Neural Networks

NLP-based models. After binary code disassembling, a binary code snippet can be represented using an assembly language, which to some extent shares many analogical topics with a natural language—both of them are organized by specific grammatical structures and can express certain semantics. Therefore, when embedding the obtained instruction sequences, many studies propose to treat instructions as words, functions as sentences, and the whole binaries as paragraphs in natural language. Then, researchers turn to converting the assembly language into low-dimensional vectors in the same way as NLP tasks. For instance, Asm2Vec [3] uses an improved PV-DM model [13], an unsupervised NLP learning approach, to generate code embedding for binary functions.

RNN/LSTM-based models. Recursive Neural Networks (RNN) and LSTM (a special RNN with gate mechanisms) [7] have been proven to be effective in sentiment analysis, language translation, and question answering. Thus, many studies turn to applying RNN/LSTM models to extract the semantics of binary code. For instance, Inner-Eye [27] proposes to use neural machine translation (NMT), a kind of LSTM model, to detect basic block similarity. Asteria [24] and SAFE [17] use LSTM/RNN to learn the semantic representation of a binary function.

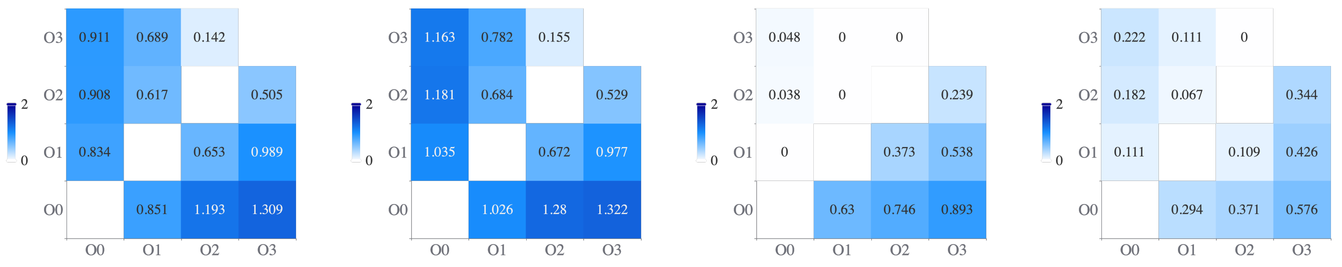
CNN-based models. CNN [1] has been widely applied to image and video recognition, classification, and processing, which is good at grasping node connectivity patterns in matrix data through kernels. Considering (1) the raw bytes contain all the semantics of binary code, and (2) the node order of the CFGs of binaries (adjacent matrices) compiled from the same source code is highly likely to remain the same, researchers proposed to learn the raw bytes pattern or the *node order information* of CFGs by CNN, which can be used as an important program feature in binary code similarity detection. For example, BinaryAI [25] uses a 3-layer CNN to construct an order-aware model to learn function embedding. αDiff [14] represents the raw bytes of binary code as a matrix and then uses a CNN to generate function embeddings.

GNN-based models. Using disassembling tools such as IDA [8] and Radare2 [18], one can obtain the graph representation (CFG or DFG) of a binary code snippet. Thus, many BinSD approaches [5, 16, 23] consider using various kinds of GNNs (a powerful class of deep learning models that yield effective representation for structured objects), including Structure2Vec [2], graph attention networks [22] and graph convolution networks (GCN) [11, 12, 15] to learn the vector representation of binary code. For instance, Gemini [23] improves Genius [4] by using Structure2Vec [2] to transform binary functions into embeddings according to the corresponding CFG and manually extracted features.

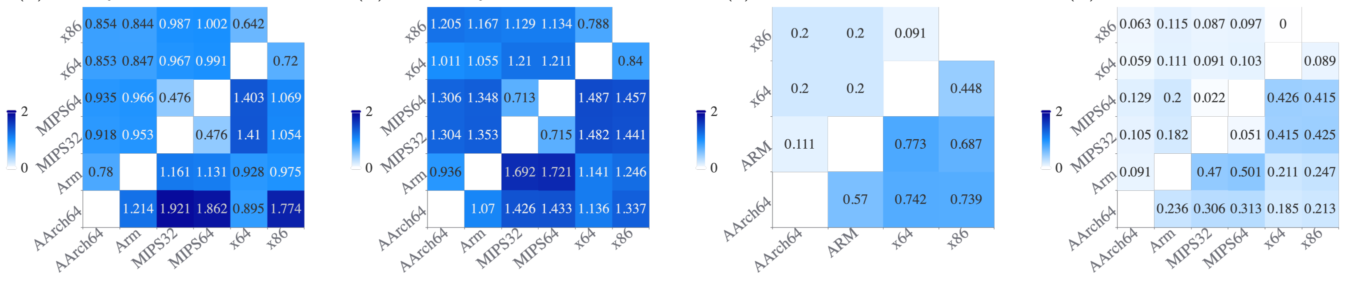
Other models. Except for the above mainstream neural networks, other neural networks such as message-passing neural networks (MPNN) [6] and proc2vec [20] are also applied to the existing AI-power BinSD tools. For instance, BinaryAI [25] also adopts MPNN, which extracts the structural information of a binary CFG to learn the representation of binary code. In this paper, we aim to comprehensively evaluate the widely used embedding neural networks, including the NLP-based, RNN/LSTM-based, CNN-based, and GNN-based models.

Table 1: Basic-dataset.

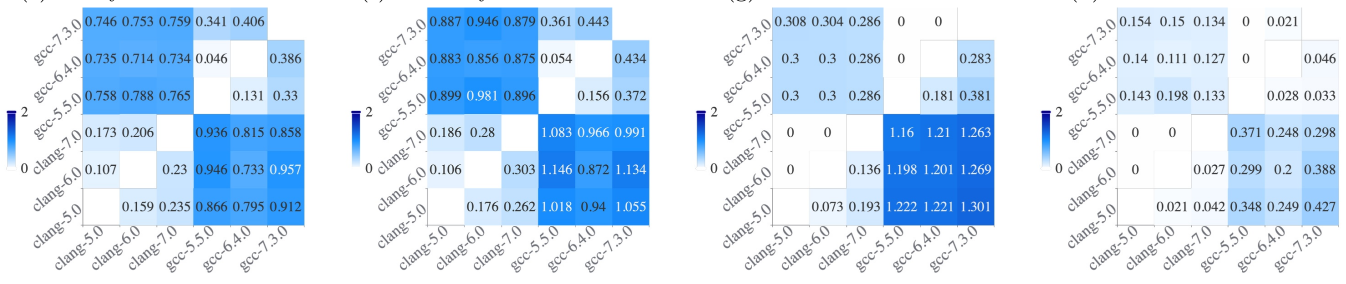
Function	**Program**	**# of ELFs**	**# of funcs under ARM**	**# of funcs under x86**	**# of funcs under x64**
Calendars Calculating	gcal-4.1	2	5,240	3,145	3,125
Data Compressing	gzip-1.9	1	3,948	3,771	3,687
	zstd-1.1.3*	1	6,055	8,220	8,064
Data Encryption	OpenSSL-1.0.1f	3	207,120	206,520	177,153
File Converting	ccd2cue-0.5	1	1,428	1,380	1,344
	enscript-1.6.6	1	1,644	1,524	1,488
File Objects Copying	xorriso-1.4.8	1	12,215	12,067	12,042
File System	direvent-5.1	1	21,256	7,875	16,713
Graph Plotting	plotutils-2.6	2	946	858	834
Language Engine	lua-5.3.5*	1	450	519	471
Math Operation	dap-3.10	1	756	750	714
	gsl-2.5	1	10,998	1,400	2,031
Network	libmicrohttpd-0.9.59	1	5,248	5,176	5,056
	ngix-1.17.0*	1	1,728	1,725	1,719
	osip-5.0.0	1	4,941	4,944	4,917
Numeric Calculation Library	gmp-6.1.2	1	8,722	8,538	8,646
OS assisting	gnudos-1.11.4	2	2,259	2,151	2,127
PostScript Converting	a2ps-4.14	2	9,344	9,030	8,980
Text Encoding Converting	libiconv-1.15	2	659	659	659
Text Processing	gawk-4.2.1	1	14,826	9,570	9,540
	sed-4.5	1	6,705	6,657	6,603
Utility	coreutils-8.29	2	6,858	6,396	6,309
	Busybox-1.27.0	1	104,364	195,528	117,054
	findutils-4.6.0	1	2,340	2,301	2,262
	inetutils-1.9.4	1	2,985	2,928	2,889
Total		33	443,045	503,632	404,427



(a) rawbyte_difference-x64 (b) assembly_difference-x64 (c) AST_difference-x64 (d) CFG_difference-x64



(e) rawbyte_difference-O2 (f) assembly_difference-O2 (g) AST_difference-O2 (h) CFG_difference-O2

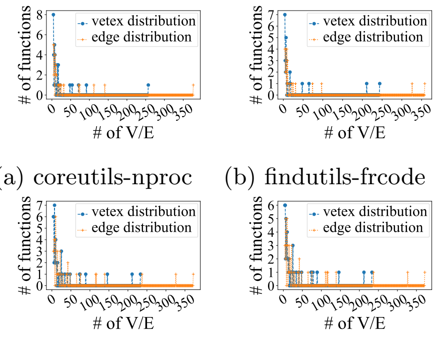


(i) rawbyte_difference-x64-O2 (j) assembly_difference-x64-O2 (k) AST_difference-x64-O2 (l) CFG_difference-x64-O2

Figure 1: The difference between binary functions compiled with different optimization levels, architectures, and compilers. (a) and (b) are the string editing distance of the raw bytes and assembly code of functions compiled with -O0 - O3 optimization levels under x64, respectively. (c) and (d) are the relative difference in the count of vertices of abstract syntax tree (AST) and CFG. The upper left and down right values of each figure are the median and average values of the difference of all function pairs, respectively.

Table 2: IoT firmware.

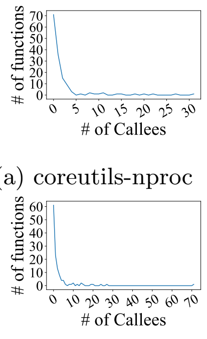
Firmware	**# of ELFs**	**# of funcs**
CAP1200v1_1.0.0_20170801-rel61314_up.bin	176	56,741
COM_T01F001_LM.1.6.18P12_sign2_TPL.TL-SC4171G.bin	97	19,573
COM_T01F001_LM.1.6.18P7_TPL.TL-SC4171G.bin	96	19,242
COVR-2600R_FW101b05_0911_txbfdisable0911190427.bin	281	19,242
COVR-2600R_FW101b05_beta01_hcr2.bin	284	75,435
COVR-3902_ROUTER_v101b05.bin	284	75,435
DAP2610-firmware-v101-rc017.bin	128	52,222
DLINK_DNR-322.2.10b022.10.0612.2014.bin	236	150,757
DLINK_DNR-322L.1.40b011.16.1219.2012.bin	225	130,155
Dap2610-firmware-v101-beta28-rc0480306165616.bin	128	52,246
Sum	1,935	651,048



(a) coreutils-nproc (b) findutils-frcode

(c) inetutils-dns* (d) inetutils-rexec

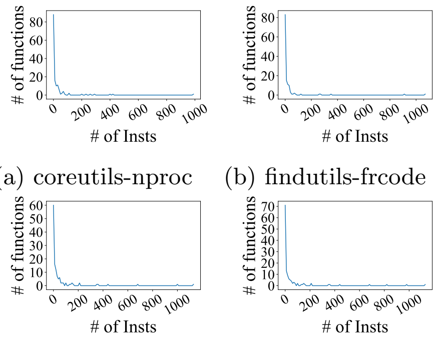
Figure 2: The distribution of basic blocks and edges.



(a) coreutils-nproc (b) findutils-frcode

(c) inetutils-dns* (d) inetutils-rexec

Figure 3: The distribution of Func-Calls.



(a) coreutils-nproc (b) findutils-frcode

(c) inetutils-dns* (d) inetutils-rexec

Figure 4: The distribution of Func-Size.

0.2 Hyper-parameters selection

0.3 Existing BinSD approaches improvement

0.4 Evaluation Dataset

As discussed in the evaluation, basic-dataset should contain representative benchmark programs with different function attributes. To construct the basic-dataset, in this paper, we first calculate the attribute distribution of BinKit[1] [10]. Then, we deduplicate the binaries with similar function attributes. Specifically, as shown in Figure 2, Figure 3, and Figure 4, the distribution of the number of basic blocks, callees, and assembly instructions are similar for the four presented binaries. Thus, we only keep one of them in the basic-dataset. Finally, as shown in Table 1, we select 25 open-source programs, including 33 ELFs and 1,351,104 functions, to construct the basic-dataset. These 25 programs are compiled with four popular optimization levels (from O0 to O3) under three architectures ARM x86 and x64. We select these three ISAs because the architectures that existing BinSD approaches support vary a lot. It is impractical to re-implement all the existing BinSD approaches to support all the ISAs. Thus, we choose the ISAs supported by over half of the evaluated BinSD approaches. In the application-dataset, the ten IoT firmware images, which include 1,935 ELFs and 651,048 functions is shown in Table 2.

0.5 Binary Comparison

We perform a comprehensive binary diffing to understand the binary change of 100,000 binary function pairs that are compiled with various compiler options. Specifically, we measure binary changes from four aspects (which can present prevalent code representations.), including the raw byte change, assembly change, and the change of the number of nodes of the AST and CFG across architectures, optimization levels, and toolchains. When performing cross-ISA and cross-compiler comparisons, we randomly set the optimization levels to -O2. When performing the cross-optimization level comparison, we randomly set the ISA to x64. As shown in Figure 1, we find that compared to raw byte, assembly code, and AST, the graph representation of binary code is more suitable for the BinSD problem since CFG is more stable across-architectures, optimization levels, and compilers, i.e., the median and average values of the difference of all function is smallest in the CFG comparison, which benefits the embedding networks to generate similar code embeddings for functions compiled from the same source code.

0.6 The influence of the K value in ranking metrics.

The value of K influences ranking metrics, including precision, recall, and MAP [21]. For instance, precision@K represents the fraction of TPs in the top-K ranking list. A larger K value may result in a smaller precision. This paper evaluates how ranking metrics change with the K value. It is worth noting that the ranking metrics of Asm2Vec are minimal. It is difficult to observe the metric changes. Thus we do not evaluate the influence of K on Asm2Vec. From the experimental results shown in Table 3, we observe that (1) except for BinaryAI-bert2 and MGMN, the precision@1, NDCG@1, and MAP@1 of all the evaluated approaches are larger than 90%, which demonstrates that when K is 1, we cannot differentiate excellent BinSD approaches because most BinSD approaches can achieve similarly good performance. (2) Except for recall, the values of other metrics decrease when k increases. Among all the evaluated

[1]https://github.com/SoftSec-KAIST/BinKit

Table 3: Evaluation metrics of similar function detection at top-K on cross-architecture under the seen program. We rank the experimental results according to precision@1.

Detector \ K	Precision@K						Recall@K						NDCG@K						MAP@K					
	1	3	5	10	30	50	1	3	5	10	30	50	1	3	5	10	30	50	1	3	5	10	30	50
Gemini	99.2	61.4	43.2	24.7	9.5	6.1	12.9	23.0	26.8	30.3	34.7	37.0	99.2	99.0	98.1	96.4	92.8	90.9	99.2	98.6	96.5	92.8	85.3	81.5
BinaryAI-skipt	99.2	49.9	31.9	17.3	6.5	9.4	12.1	17.0	18.0	19.4	21.8	20.8	99.2	99.1	98.54	97.0	94.0	95.3	99.2	98.8	97.42	94.3	88.2	90.9
VulSeeker-skip	99.1	58.6	38.6	20.7	7.6	4.8	12.8	21.4	23.5	25.1	27.6	29.0	99.1	98.9	98.4	97.3	94.8	93.3	99.1	98.6	97.4	94.7	89.7	86.8
UFE-rnn	99.1	34.6	21.1	10.8	3.8	2.4	19.6	20.3	20.5	20.8	21.7	22.4	99.1	99.6	99.4	99.0	97.6	96.5	99.1	99.4	99.0	98.2	95.7	93.7
Gemini-skip	99.0	60.1	40.7	22.2	8.5	5.5	13.0	22.4	25.0	27.1	30.9	33.1	99.0	98.3	96.9	93.5	91.3	99.0	98.9	98.5	97.0	94.0	86.9	82.8
Focus-skip	98.9	59.6	40.3	22.7	9.1	6.0	12.7	22.0	24.7	27.6	32.9	35.8	98.9	98.9	98.1	96.2	92.0	89.5	98.9	98.5	96.7	92.6	83.7	78.8
UFE-attention	98.7	37.0	23.1	12.3	4.6	3.0	19.2	20.8	21.3	22.3	24.4	25.7	98.7	99.2	98.8	97.7	95.0	93.0	98.7	98.7	97.9	95.8	90.6	87.0
UFE-mean	98.6	35.7	21.9	11.3	4.0	2.5	19.1	20.2	20.6	21.0	22.1	22.9	98.6	99.2	99.0	98.4	96.8	95.6	98.6	98.9	98.3	97.3	94.2	92.1
VulSeeker	97.8	49.3	33.0	18.2	6.9	4.4	12.8	19.0	21.0	23.0	26.2	27.9	97.7	98.3	97.6	96.3	93.1	91.3	97.8	97.8	96.3	93.3	87.0	83.5
Focus	96.2	56.5	38.9	21.9	8.4	5.4	12.7	21.4	24.4	27.3	31.1	33.0	96.2	97.4	96.7	95.1	92.0	90.4	96.2	96.6	94.9	91.3	84.9	81.6
SAFE	91.6	59.8	45.8	30.1	13.9	9.5	18.0	29.9	35.9	44.1	57.5	64.0	91.6	94.2	93.3	91.1	86.0	83.4	91.6	92.7	89.9	84.1	72.3	66.7
BinaryAI-bert2	88.5	41.5	27.3	15.2	6.3	4.1	9.8	13.2	14.3	15.8	19.1	20.7	88.5	89.4	88.8	87.6	84.7	83.8	88.5	88.6	86.8	83.7	76.3	73.4
MGMN	63.3	39.3	29.5	18.9	8.5	5.8	8.3	15.5	19.1	24.1	32.2	36.1	63.3	69.9	72.3	73.1	72.0	70.9	63.3	67.7	67.7	64.8	58.4	55.1

metrics, precision@K is most significantly influenced by the value of K. When K is larger than 30, the precision of most BinSD approaches drops under 10%. It requires great manual effort to identify FPs. (3) When K is 50, the recall of most BinSD approaches is less than 40%, which indicates that most BinSD approaches still have considerable improvement space. To achieve both relatively good precision and recall, in the following evaluation in this section, we define K = 5.

0.7 Vulnerable Function Confirmation

Actually, vulnerability confirmation contains two phrases—rough match and precise comparison. In the rough match, researchers identify the semantically similar functions that are highly likely compiled from the same source code of the queried vulnerabilities in a large function repository (such as IoT devices). In the precise comparison, researchers need to determine whether the semantically similar functions are buggy by performing accurate patch presence tests [9, 26]. For most BinSD approaches, including Gemini [23], VulSeeker [5], SAFE [17], and Oscar [19], they are essentially seeking the same affected functions in the search repository. Namely, BinSD is used to perform a rough match. Consequently, using state-of-the-art AI-powered BinSD approaches, the search results obtained are potentially vulnerable functions. To finally examine the patch presence for these potentially vulnerable functions, one needs to generate and compare the summaries of specific patches' semantics.

In the bug confirmation process, we manually compare the pseudocode snippets (an ISA agnostic language similar to source code) of the search result and query function obtained by IDA pro to examine whether they are semantically identical. Specifically, we perform data flow analysis, constant string comparison, and call function comparison to determine whether two pseudocode snippets are compiled from the same source code.

0.8 The difference between GNN over-smoothing issue and embedding collision problem.

We would like to clarify that the GNN over-smoothing issue is different from the embedding collision problem we uncovered in this paper. The differences between GNN over-smoothing and embedding collision are as follows.

(1) Different embedding levels. Both of these two problems are caused by similar embeddings. However, for the embedding collision problem, the embedding refers to graph embedding (the summation of all the node embeddings within the function). Embedding collision occurs when two graph embeddings that are supposed to be different happen to be similar. By contrast, the embedding in the over-smoothing issue refers to node embedding. Over-smoothing issue means two node embeddings with different labels within a graph tend to be similar.

(2) Different mechanisms. The root cause for embedding collision is that the node embedding summation of different functions happens to be similar. For instance, the embedding of set_keygen_ctx (a function in OpenSSL-1.0.1f), which contains 48 basic blocks with large embedding values, is very similar to the one of crl2pkcs7_main (another totally different function in OpenSSL-1.0.1f), which includes 92 basic blocks with small embedding values (the summation of 48 large node embeddings is similar to the summation of 92 small node embeddings). Embedding collision ends up with similar embeddings for different functions, which significantly reduces the accuracy of binary code similarity detection.

By contrast, for the over-smoothing issue, it is expected that the embeddings of nodes (that are linked to each other and have common neighbors) are also similar because node embedding is the aggregation of node embeddings of neighbors. Over-smoothing ends up with similar embeddings for nodes that don't have the same label, which will result in mislabeling them.

In summary, even though the node embeddings within two different functions are significantly different (e.g., there is no over-smoothing issue), embedding collision may still happen. Following this comment, we will add more description and explanation about the difference between GNN over-smoothing and embedding collision in the revised manuscript to avoid potential confusion. Thus, we do not aim for the evaluation of all the AI-powered BinSD approaches but the representative state-of-the-art BinSD approaches using various embedding strategies.

References

[1] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. Ieee, 2017.

[2] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*, pages 2702–2711, 2016.

[3] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe 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)*, pages 472–489, 2019.

[4] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491. ACM, 2016.

[5] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang 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*, pages 896–899. ACM, 2018.

[6] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[7] K Greff, R. K. Srivastava, J Koutnik, B. R. Steunebrink, and J Schmidhuber. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks & Learning Systems*, 28(10):2222–2232, 2017.

[8] Hex-Rays. IDA: About. https://www.hex-rays.com/products/ida/, 2023. Accessed 2021-08-20.

[9] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. Pdiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1149–1163, 2020.

[10] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*, pages 1–23, 2022.

[11] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. Improving cross-platform binary analysis using representation learning via graph alignment. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 151–163, 2022.

[12] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[13] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[14] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 667–678. ACM, 2018.

[15] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search. 2023.

[16] Luca Massarelli, Giuseppe A Di Luna, Fabio Petroni, Leonardo Querzoni, Roberto 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)*, pages 1–11, 2019.

[17] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.

[18] pancake and the community. Radare2: About. https://github.com/radareorg/radare2, 2022. Accessed 2022-08-12.

[19] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. How could neural networks understand programs? In *International Conference on Machine Learning*, pages 8476–8486. PMLR, 2021.

[20] Noam Shalev and Nimrod Partush. Binary similarity detection using machine learning. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, pages 42–47. ACM, 2018.

[21] Daniel Valcarce, Alejandro Bellogín, Javier Parapar, and Pablo Castells. Assessing ranking metrics in top-n recommendation. *Information Retrieval Journal*, 23:411–448, 2020.

[22] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.

[23] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn 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*, pages 363–376. ACM, 2017.

[24] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang 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)*, pages 224–236. IEEE, 2021.

[25] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1145–1152, 2020.

[26] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 887–902, 2018.

[27] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS)*, 2019.