Identifying Library Functions in Stripped Binary: Combining Function Similarity and Call Graph Features

ZhanPeng Liu and Xinhui Han(✉)

Peking University, Beijing, China
{liuzhanpeng,hanxinhui}@pku.edu.com

Abstract. Reverse engineering binary programs without debug information, such as malwares and embedded firmwares, is often a challenging and time-consuming process that relies heavily on manual analysis. Automating the process of identifying frequently used library functions can significantly improve the efficiency. While machine learning techniques have shown satisfactory results in computing binary function similarity in specific experimental contexts, their performance in open-set retrieval task remains largely unexplored. Notably, identifying known functions in stripped binaries falls under this category. To contribute to this area of research, we introduce a brand-new dataset derived from popular Rust projects. This dataset not only aims to stimulate further research on Rust program analysis but also serves as a robust platform for evaluating the performance of state-of-the-art methods in open-set function retrieval tasks. Through our analysis, we discover that similarity-only methods have limited effectiveness in rejecting negative samples. In response to this identified shortcoming, we present a novel approach that integrates features derived from function call graphs, enabling us to determine a function's identity by considering both its similarity and call relationships with other functions. Experimental results demonstrate that our method enhances overall performance compared to similarity-only solutions, especially under more challenging conditions.

Keywords: Binary Function Similarity · Reverse Engineering · Rust Programming Language

1 Introduction

Reverse engineering of stripped binaries presents a challenging task, requiring substantial expertise and considerable time investment. Frequently, analysts spend a considerable amount of time analyzing functions, only to discover they are functions in common libraries. Automating the process of identifying known functions will significantly reduce the effort involved, streamlining the reverse engineering workflow and improving overall efficiency for analysts.

Industrial solutions tend to use raw bytes [6,18]. Although fast and simple, these solutions are highly sensitive to even minor variations. In recent years,

H. Duan et al. (Eds.): SecureComm 2023, LNICST 568, pp. 395–408, 2025.
https://doi.org/10.1007/978-3-031-64954-7_20

there has been a rising interest in employing machine learning techniques to devise solutions capable of handling variations. These methods calculate vector representations for binary functions and use distance metrics to assess similarity. Although these approaches achieve satisfactory results within their specific datasets and benchmarks [2,10,13,21], their evaluative metrics, including AUC (Area Under Curve) and MRR (Mean Reciprocal Rank), are not comprehensive enough.

When compare functions between a stripped binary and a constructed function database, it becomes an open-set retrieval task. If a binary function is not present in the database, we can still retrieve the most similar target function. In such cases, an acceptance threshold is required to reject irrelevant matches. To the best of our knowledge, there are no existing studies that have explored the use of embedding based similarity measures for function identification in an open-set context.

To evaluate the performance of existing similarity calculation methods in open-set function retrieval task, we construct a brand new dataset, ROSP[1] (Rust Open Source Projects), sourced from popular Rust projects. Rust, a programming language known for speed and safety, has gained significant popularity since its launch in 2010. Even the Linux kernel has incorporated support for Rust as of October 2022 [1]. Our rationale for creating this dataset is twofold. Firstly, Rust binaries typically contain numerous library functions, making them a suitable focus for the challenges we aim to address. Secondly, despite Rust's growing popularity and adoption, Rust program analysis remains a relatively unexplored area. Consequently, ROSP serves as a valuable resource for future inquiries into Rust program analysis.

We conduct evaluations of leading embedding-based similarity calculation techniques using ROSP dataset. Although they still achieve high AUC and MRR score, they struggle to distinguish between positive pairs and the most similar negative pairs in open-set retrieval task, resulting in sub-optimal effectiveness.

To address the issue, we introduce the RCG (Retrieval with Call Graph), which merges function-level similarity with call graph features. Instead of identifying functions by similarity alone, RCG integrates call-related functions for improved decisions. When tested on the ROSP dataset, RCG outperforms the similarity-only approach, proving its efficacy in open-set function retrieval tasks.

Our contributions can be summarized as follows:

- We introduce the ROSP dataset from prominent Rust projects to facilitate the evaluation of open-set function retrieval tasks. It also enriches Rust analysis resources, and diversifies binary similarity datasets traditionally focused on C programs.
- We assess existing similarity methods in an open-set setting, identifying their limitations in separating positive pairs and most similar negative pairs.
- We propose a novel method, RCG, that infers a function's identity by integrating function-level similarity and call graph features, thereby providing a more comprehensive solution.

[1] https://github.com/pkuGenuine/ROSP.

– We perform experiments utilizing the ROSP dataset with our proposed RCG method. The results demonstrate significant improvements in performance, underscoring the effectiveness of RCG in overcoming the limitations inherent in existing embedding-based similarity calculation methods.

2 Related Work

2.1 Similarity Calculation with Machine Learning

A wide range of machine learning methods have been proposed to generate vector embedding for functions, which can be utilized to solve binary similarity problem and facilitate downstream tasks.

Gemini [19] pioneered embedding generation for binary functions using neural networks, outperforming non-embedding methods [4,5] with its manual feature extraction and graph neural networks. Asm2Vec [2], inspired by paragraph2doc [9], uses the function's control flow graph to obtain assembly sequences, producing robust function representations. This model's resilience to compiler changes and obfuscation has spurred NLP adoption in this domain, as seen in SAFE [14] and Trex [15]. CodeCMR [21] is a complicated model designed to match binary and source code at function level. Subsequent experiments by Andrea M. et al. [13] have shown that the model's binary encoding component is capable of generating high-quality embeddings.

Andrea M. et al. [13] conducted a comprehensive measurement study in this field. They re-implemented representative approaches and conducted a fair and meaningful comparison with a unified dataset and metrics. Notably, their results demonstrate that CodeCMR is currently the state-of-the-art in this field. Additionally, a much simpler GNN model proposed by Li et al. [10] also shows promise as an effective method for binary function embedding.

All these machine learning methods mentioned previously have shown impressive AUC and MRR scores on their datasets. Yet, none have been tested in an open-set retrieval setting that demands the rejection of irrelevant matches.

2.2 Third-Party Components Detection

Third-party components (TPC) detection refers to the process of identifying external components used in a software system. While closely related to our issue, most of the researches in this area [3,22] centers on library-level detection rather than function-level identification.

ModX [20] breaks down binaries into separate modules based on call graphs and computes module similarity scores. This approach also matches functions across modules to determine module similarity, paving the way for function-level identification. Nevertheless, its effectiveness hinges on the modularization process and the stability of library module partitions.

2.3 Binary Code Clone Detection

Another research domain closely related to our issue is code clone detection. Studies in this field [7,8,11] that focus on binary programs at the function level also rely on similarity scores to identify clones. However, the algorithms used for this purpose are often complex, sometimes simulating program execution to generate signatures, which limits their ability to scale effectively in large repositories. In contrast, our proposed method uses function embeddings to calculate similarity scores, a process that can be accelerated with commonly available GPUs.

2.4 Retrieval Systems

In fields such as computer vision and face identification, open-set recognition and retrieval have emerged as crucial concerns. FaceNet [16] makes significant contributions by introducing the triplet loss function for training deep neural networks for face recognition. Moreover, the system presents a novel online negative exemplar mining strategy that use semi-hard triplets, ensuring that the difficulty of triplets consistently increases as the network is trained. However, Wu et al. demonstrate that distance weighted sampling [12] can achieve better results than semi-hard sampling. In the paper which introduces CodeCMR, a variation of distance weighted sampling called norm weighted sampling [21] is proposed and shown to produce even better results than previous methods.

3 Problem Definition

In this section, we provide a formal definition of the problem we aim to address, specifically, the open-set function retrieval problem:

Given a stripped binary program Q, and a database D containing labeled functions typically sourced from extensively utilized libraries, we can categorize the functions in Q into two groups: positive query functions $Q_p \subseteq Q$ with a corresponding match in D and negative query functions $Q_n = Q - Q_p$ without any match in D. Our objective is to develop a method that accurately identifies the corresponding function in D for each function in Q_p, while also being able to reject functions in Q_n.

For evaluation, we rely on precision and recall as key metrics. Let M_r represent all retrieved pairs that are accepted, and M_p denote all positive pairs. Precision and recall can then be computed as:

$$\text{Precision} = \frac{|M_r \cap M_p|}{|M_r|}, \text{Recall} = \frac{|M_r \cap M_p|}{|M_p|}. \tag{1}$$

4 Dataset Construction

4.1 Motivation

The construction of the ROSP dataset is motivated by two main reasons.

Firstly, Rust programs provide an ideal testing ground for our research objectives. The Rust language boasts a versatile standard library and a vibrant open-source ecosystem supplying a plethora of third-party packages. Furthermore, Rust libraries are typically co-compiled with the output binary, which often results in a significant proportion of library functions being included in Rust programs.

Secondly, at the time of our study, there is a notable absence of comprehensive Rust program datasets available for research. Rust binaries differ from those of C/C++ languages in two key ways: 1) they follow a different calling convention, leading to distinct instruction distributions, which introduces additional challenges for embedding techniques; 2) as mentioned earlier, Rust libraries are usually co-compiled with the output binary, meaning only a subset of library functions are included in the final binary. Many traditional program-dependence (TPD) methods are ineffective under these conditions. Existing datasets for binary-related research primarily consist of projects written in C and cover only a limited range of program types. Our proposed ROSP dataset serves as a valuable supplement in this context.

4.2 Gathering Feathers and Ground Truth

We collect Rust open-source projects from "GitHub Ranking", a GitHub repository that periodically updates the top 100 most-starred Rust projects. We try to compile these projects with four optimization levels (O0-O3) and use Ghidra to extract various features, including assembly code, control flow graphs and function call graphs. Some projects couldn't be compiled due to dependency issues or specific rustc version requirements, or because they only contained library crates that couldn't be compiled into binaries. We successfully extract features from 367 binaries with varying optimization levels, covering 35 projects.

We also made efforts to sanitize the extracted functions by comparing the function range in DWARF information with those identified by Ghidra. Functions exhibiting significant discrepancies are filtered out. Also, like most of previous works, we filtered out functions containing less than 5 basic block.

4.3 Dataset for Training Similarity Models

To construct the dataset for training and testing similarity calculation models, we group all functions by label and attempted to select one function from each optimization level for each label. We calculate a hash based on the assembly code of each function, ensuring that no two chosen functions share identical features. We manage to generate over 40,000 labels and 140,000 functions. Based on labels, we divide dataset into train, validation, and test sets, with a ratio of 0.6, 0.2, and 0.2, respectively.

To validate and test the machine learning models, positive and negative pairs are generated. For each label, a positive pair is created across different optimization levels, while a negative pair is generated across different labels. In addition,

another positive pair and 1,000 negative samples are selected for each label to be used in the ranking task.

4.4 Dataset for Open-Set Retrieval Test

To evaluate the performance of different methods on open-set retrieval task, two components are necessary: one for building a database and another for querying. For the query component, all functions within each binary constitute a query set. For the database component, we gather common library functions, such as those found in crates like std, core, alloc, etc., under the same optimization level, generating 4 repositories.

5 Proposed Method

5.1 Solution Overview

The overall framework of our proposed method is shown in Fig. 1.

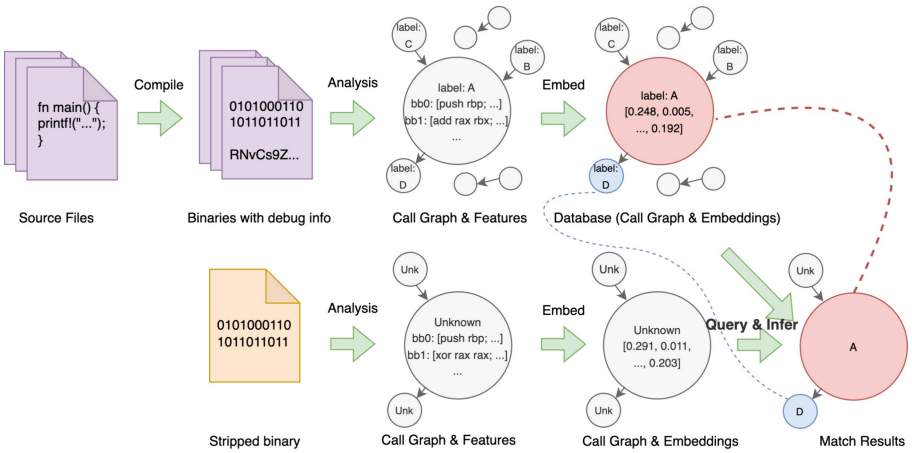

Fig. 1. Library Function Retrieval System Overview

First, we extract commonly used library functions from a collection of repository binaries and compute their vector representation using an embedding-based similarity calculation model. Additionally, we record the call relationships between the selected functions. The combination of vectors and call graph constitutes the database.

While analyzing a stripped binary, we also generate vector embeddings and a call graph for its functions. We then adopt the "Retrieval with Call Graph" (RCG) approach to identify known functions in the query binary.

5.2 Details of the Retrieval Algorithm

RCG is a three-phase matching algorithm that combines vector space proximity and call relations.

The initial phase involves the measurement of similarities and the identification of potential mappings. The similarity is calculated based on the proximity of embeddings within the vector space. For each function in the query binary Q, its similarity with all functions in the database D is computed, and the most similar target is selected. These pairs exhibiting the highest similarity are considered as potential mappings. In addition, a predefined threshold is set, and if the similarity between two functions surpasses this threshold, they are classified as "highly probable mappings".

This is followed by the "neighbor validation phase". For each potential mapping that has been identified, we enumerate the pairs of neighbors, which includes both callers and callees, that also qualify as highly probable mappings. The process is illustrated in Fig. 2, which uses an example to delineate the concept clearly. In this figure, two red circles symbolize two functions perceived as potential mapping pairs. The function within the query binary has two callers and two callees, with one of its callers and both its callees exhibiting highly probable mappings within the function database. These mappings also represent a caller or callee of the potential mapping, thus affirming that this mapping is neighbored by three highly probable mappings. An extra similarity score is awarded to the potential mapping based on the count of highly probable mapping neighbors. If this addition raises the similarity score beyond the threshold, they too are classified as highly probable mappings. This phase is recursively carried out until no new highly probable mappings are identified.

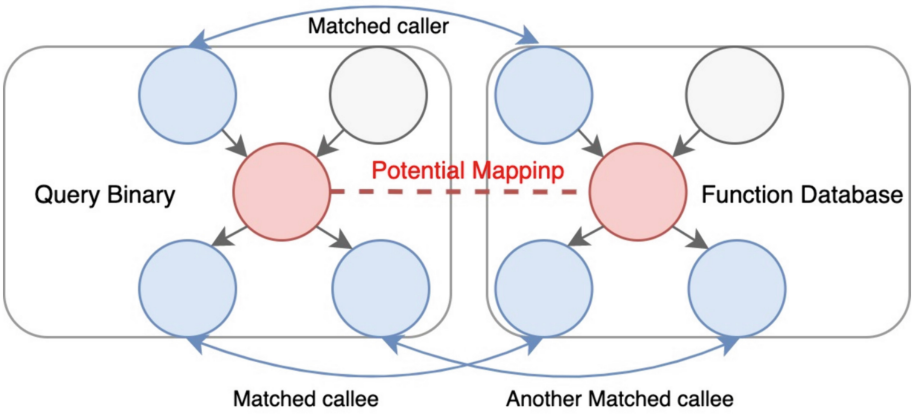

Fig. 2. Example of Neighbor Validation

"Final deduction" is our last phase. Based on the similarity derived from the embeddings and updated during the neighbor validation phase, we apply

another threshold to decide whether to accept or reject potential mappings. This concluding step ensures a comprehensive appraisal of the probable mappings, aiming to yield optimal outcomes.

A pseudo code implementation is provided in appendix.

6 Experiments and Results

In this section, we present the experimental setup and results of our study, which consist of three subsections. The first subsection discusses the training and testing for the similarity calculation model designed to solve binary function similarity tasks. The second subsection examines the distance distribution, while the final subsection focuses on the open-set retrieval test.

6.1 Training and Testing Similarity Calculation Models

Based on numerous experiments conducted by Andrea M. et al. [13], the GNN model proposed by Li et al. [10] and Tencent team's CodeCMR [21] have both been found to be effective function encoders. To evaluate their performance on our ROSP dataset, we re-implemented the GNN model and the part of CodeCMR that encodes control flow graphs. We trained these models under different settings and evaluated their effectiveness. The results are shown in Table 1.

Table 1. Performance of different models on test set. With triplet loss, margin=0.5 and distance weighted sampling

Model	AUC	MRR@10	Recall@1	Recall@10
GNN(BoW, euc)	98.96%	76.38%	68.21%	91.57%
GNN(BoW, cos)	99.59%	89.72%	84.97%	97.27%
CMR(Asm, euc)	99.73%	93.47%	90.61%	98.30%
CMR(Asm, cos)	99.95%	97.96%	96.83%	99.54%

In previous research, some studies used euclidean distance to calculate similarity, while others used cosine distance. However, there is no clear consensus on which one is better. In our experiments, we demonstrate that cosine distance produces better results. Additionally, we tested the effects of various negative sampling methods, including semi-hard sampling [16], distance-weighted sampling [17], and norm-weighted sampling [21], as well as different margin values. However, our experiments do not reveal any significant differences among these settings, and thus they are not included in the table.

Informed by the outcomes of our experimental evaluations, the CodeCMR like model, when trained with cosine distance shows best performance. Hence, we elect to employ this configuration for subsequent tests.

6.2 Distance Distribution

When adapting function-level similarity calculation methods for our open-set function retrieval task, a large acceptance threshold for distance will result in high recall but low precision, and vice versa. To gain a more comprehensive insights, we examine two sets of distances. The first set is constituted of distances between all positive pairs, which includes the functions in the positive query set Q_p and their corresponding functions in the database D. Conversely, the second set comprises the closest negative pairs, which are the functions in the negative query set Q_n and the nearest function in the database D.

We collected the distances between query binaries and function databases with different optimization levels. We specifically chose binaries compiled with the O1 optimization to build the database, given that it represents an intermediate level of optimization, thereby providing a balanced base for our analysis and subsequent function retrieval operations.

Our observations generally indicate that distinct query binaries, when subject to the same optimization setting, exhibit similar distance distributions. However, a noticeable contrast arises when binaries, despite originating from the same source code, are compiled with different optimization levels. This is highlighted through our examination of two distinct projects: RustPython and Relay. The distribution of distances for these projects under different optimization settings is illustrated in Fig. 3.

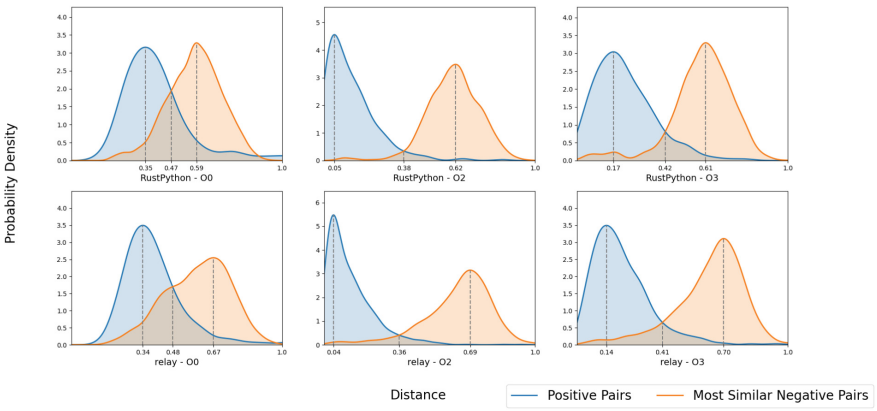

Fig. 3. Distance Distributions Comparison

When the query binary, compiled with the O0 optimization level, is compared against the database constructed from binaries compiled with the O1 optimization level, the overlap between the two distributions is considerably larger than when compared with binaries compiled at the O2 level. This greater overlap indicates a substantial difficulty in conducting open-set retrieval tasks, as the distinction between matched and unmatched functions becomes less pronounced.

6.3 Open-Set Recognition

In this section, we conduct a comprehensive comparative analysis between the conventional solution that relies solely on the proximity within the functions' vector space and our integrated approach RCG. Given that the O1 optimization level represents an intermediate point in the range of optimization options, we use binaries compiled with O1 optimization to populate the database. For the query set, we employ binaries compiled with O0, O2, and O3 optimizations.

We carefully select query binaries that encompass at least 4000 functions to ensure a substantial sample size that lends statistical significance to our analysis. Furthermore, within each project, we choose a single binary to avoid potential bias stemming from any inherent similarities.

Given the typical inverse relationship between precision and recall, we strategically set the recall rate at 0.5 and monitor the corresponding differences in precision. Additionally, to gain a comprehensive understanding of the trade-off between precision and recall, we compute the area under the precision-recall curve, following a method analogous to that used in Receiver Operating Characteristic (ROC) curve analysis. Selected results of this analysis are presented in Table 2. In each column, the left number is derived from the proximity-only method, denoted as CMR, while the right number is obtained from our pro-

Table 2. Comparative Analysis: Precision (Recall=0.5) & Area under Precision-Recall Curve

Optimization Level	O0		O3	
	Precesion	AUC	Precesion	AUC
Binary	CMR/RCG	CMR/RCG	CMR/RCG	CMR/RCG
bat/bat	0.74/0.83	0.58/0.68	0.88/0.94	0.82/0.89
bevy/animated_fox	0.56/0.69	0.43/0.58	0.75/0.83	0.66/0.73
coreutils/coreutils	0.63/0.77	0.53/0.64	0.92/0.93	0.83/0.86
gitui/gitui	0.70/0.83	0.54/0.68	0.92/0.97	0.85/0.89
helix/hx	0.59/0.71	0.46/0.60	0.81/0.89	0.71/0.79
nushell/nu	0.67/0.80	0.54/0.66	0.86/0.93	0.77/0.83
relay/relay	0.67/0.78	0.53/0.64	0.87/0.92	0.78/0.84
ruffle/exporter	0.59/0.73	0.47/0.62	0.78/0.88	0.74/0.82
RustPython/rustpython	0.43/0.47	0.33/0.44	0.58/0.65	0.54/0.61
sonic/sonic	0.63/0.79	0.47/0.64	0.86/0.91	0.79/0.85
spacedrive/prisma	0.48/0.55	0.38/0.45	0.69/0.69	0.62/0.63
starship/starship	0.64/0.75	0.51/0.64	0.87/0.92	0.79/0.84
swc/minify	0.51/0.63	0.39/0.55	0.84/0.93	0.77/0.82
tools/rome	0.53/0.60	0.4/0.48	0.77/0.85	0.65/0.74
zola/zola	0.67/0.77	0.53/0.64	0.87/0.92	0.78/0.83
Average	0.60/0.71	0.47/0.60	0.82/0.88	0.74/0.80

posed RCG method. We exclude O2 from this table due to the negligible difference between the two methods, as they achieve similar precision (0.90/0.92) and AUC (0.85/0.87) scores on average. It's important to highlight that when calculating precision and recall, we deliberately exclude function pairs that are exactly identical.

The complexity of open-set function retrieval tasks varies significantly with different optimization settings. When we employ O1 binary functions as the database and query with O2 binary functions, the embedding-based similarity solution exhibits robust performance. With recall set to 0.5, it can achieve an average precision of 0.9. Additionally, the area under the precision-recall curve (AUC) is up to 0.85. However, when the query functions are derived with O0 optimization, the performance deteriorates significantly, yielding a precision of 0.62 and an AUC of 0.48.

Our proposed methodology, which combines function-level similarity with call relations, demonstrates consistent superiority over the traditional approach in the domain of open-set function retrieval tasks. Under varying conditions of query function compilations, namely O0, O2, and O3 optimizations, our approach exhibits an enhancement in performance. Specifically, it yields precision improvements of 0.11, 0.02, and 0.06, respectively, and augments the AUC by 0.13, 0.02, and 0.06. The trend is clear: as the problem becomes more challenging, our proposed methodology shows even greater advancement, underscoring its robustness and effectiveness under demanding scenarios.

7 Conclusion

In this paper, we set out to scrutinize and enhance the performance of state-of-the-art function similarity solutions in the context of open-set function retrieval tasks. We constructed a brand-new Rust dataset to serve as a comprehensive testing ground for our experiments. Our empirical analysis reveals that relying solely on embedding-based similarity presents challenges when addressing open-set function retrieval tasks, particularly when the optimization levels differ significantly between query functions and the database.

To address this issue, we leverage the knowledge of function call relations, which enriches the comparison process and contributes to improved overall performance. Specifically, prioritizing potential matched pairs that exhibit a greater number of possibly matched neighbors offers a viable strategy to enhance retrieval accuracy, especially in scenarios where function-level similarity does not yield optimal results.

While our experimental framework primarily revolves around varying compilation optimization levels, the insights and conclusions drawn from our study can be extended to other contexts, such as when dealing with differences in system architecture.

Acknowledgement. This work is supported by the National Natural Science Foundation of China (61972224).

A Pseudocode for Retrieve with Call Graph

Algorithm 1 Retrieve with Call Graph

```
procedure FunctionMatch(Q, D, t1, t2)
    M_all, M_high, M_out ← ∅, ∅, ∅
    for f_q ∈ Q do
        f_d, similarity ← MostSimilar(q, D)
        M_all ← M_all ∪ {(f_q, f_d, similarity)}
        if similarity ≥ t1 then
            M_high ← M_high ∪ {(f_q, f_d)}
        end if
    end for
    while M_high changes do
        for (f_q, f_d, s) ∈ M_all do
            neighbor_match_cnt ← CountHighProbable(f_q, f_d, M_high)
            s ← s+ CalcuBonus(neighbor_match_cnt)
            if s ≥ t1 then
                M_high ← M_high ∪ {(f_q, f_d)}
            end if
        end for
    end while
    for (f_q, f_d, s) ∈ M_all do
        if s ≥ t2 then
            M_out ← M_out ∪ {(f_q, f_d)}
        end if
    end for
    return M_out
end procedure
```

References

1. Cook, K.: [git pull] rust introduction for v6.1-rc1. https://lore.kernel.org/lkml/202210010816.1317F2C@keescook/
2. Ding, S.H.H., Fung, B.C.M., Charland, P.: Asm2Vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 472–489 (2019). https://doi.org/10.1109/SP.2019.00003
3. Duan, R., Bijlani, A., Xu, M., Kim, T., Lee, W.: Identifying open-source license violation and 1-day security risk at large scale. In: CCS 2017, Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2169–2185. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134048,
4. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E.: discovRE: efficient cross-architecture identification of bugs in binary code. In: Network and Distributed System Security Symposium (2016)

5. Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., Yin, H.: Scalable graph-based bug search for firmware images. In: CCS 2016, Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 480–491. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978370
6. Hex-Rays: Ida f.l.i.r.t. technology: In-depth. https://hex-rays.com/products/ida/tech/flirt/in_depth/
7. Hu, Y., Zhang, Y., Li, J., Gu, D.: Binary code clone detection across architectures and compiling configurations. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 88–98 (2017). https://doi.org/10.1109/ICPC.2017.22
8. Hu, Y., Zhang, Y., Li, J., Wang, H., Li, B., Gu, D.: BinMatch: a semantics-based hybrid approach on binary code clone analysis. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 104–114 (2018). https://doi.org/10.1109/ICSME.2018.00019
9. Le, Q.V., Mikolov, T.: Distributed representations of sentences and documents. CoRR **abs/1405.4053** (2014). http://arxiv.org/abs/1405.4053
10. Li, Y., Gu, C., Dullien, T., Vinyals, O., Kohli, P.: Graph matching networks for learning the similarity of graph structured objects. CoRR **abs/1904.12787** (2019). http://arxiv.org/abs/1904.12787
11. Liao, Y., Cai, R., Zhu, G., Yin, Y., Li, K.: MobileFindr: function similarity identification for reversing mobile binaries. In: Lopez, J., Zhou, J., Soriano, M. (eds.) ESORICS 2018. LNCS, vol. 11098, pp. 66–83. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99073-6_4
12. Manmatha, R., Wu, C., Smola, A., Krähenbühl, P.: Sampling matters in deep embedding learning. In: 2017 IEEE International Conference on Computer Vision (ICCV), pp. 2859–2867 (2017)
13. Marcelli, A., Graziano, M., Ugarte-Pedrero, X., Fratantonio, Y., Mansouri, M., Balzarotti, D.: How machine learning is solving the binary function similarity problem. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 2099–2116. USENIX Association, Boston, MA (2022). https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli
14. Massarelli, L., Luna, G.A.D., Petroni, F., Querzoni, L., Baldoni, R.: SAFE: self-attentive function embeddings for binary similarity. CoRR **abs/1811.05296** (2018). http://arxiv.org/abs/1811.05296
15. Pei, K., Xuan, Z., Yang, J., Jana, S.S., Ray, B.: Trex: learning execution semantics from micro-traces for binary similarity. arXiv arXiv:2012.08680 (2020)
16. Schroff, F., Kalenichenko, D., Philbin, J.: FaceNet: a unified embedding for face recognition and clustering. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 815–823 (2015). https://doi.org/10.1109/CVPR.2015.7298682
17. Wu, C., Manmatha, R., Smola, A.J., Krähenbühl, P.: Sampling matters in deep embedding learning. CoRR arXiv:1706.07567 (2017)
18. Xorpd: Fcat - a catalog of function categories. https://www.xorpd.net/pages/fcatalog.html
19. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: CCS 2017, Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 363–376. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134018

20. Yang, C., Xu, Z., Chen, H., Liu, Y., Gong, X., Liu, B.: ModX: binary level partially imported third-party library detection via program modularization and semantic matching. In: ICSE 2022, Proceedings of the 44th International Conference on Software Engineering, pp. 1393–1405. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510003.3510627
21. Yu, Z., Zheng, W., Wang, J., Tang, Q., Nie, S., Wu, S.: CodeCMR: cross-modal retrieval for function-level binary source code matching. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems, vol. 33, pp. 3872–3883. Curran Associates, Inc. (2020). https://doi.org/10.5555/3495724.3496050
22. Zhao, B., et al.: A large-scale empirical analysis of the vulnerabilities introduced by third-party components in IoT firmware. In: ISSTA 2022, Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 442–454. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3533767.3534366