CEBin: A Cost-Effective Framework for Large-Scale Binary Code Similarity Detection

Hao Wang[*]
Tsinghua University
Beijing, China
hao-wang20@mails.tsinghua.edu.cn

Zeyu Gao[*]
Tsinghua University
Beijing, China
gaozy22@mails.tsinghua.edu.cn

Chao Zhang[*][†][‡]
Tsinghua University
Beijing, China
chaoz@tsinghua.edu.cn

Mingyang Sun
University of Electronic Science and Technology of China
Chengdu, China
2020090918021@std.uestc.edu.cn

Yuchen Zhou
Beijing University of Technology
Beijing, China
zhouyuchen@emails.bjut.edu.cn

Han Qiu[*]
Tsinghua University
Beijing, China
qiuhan@tsinghua.edu.cn

Xi Xiao
Tsinghua University
Shenzhen, China
xiaox@sz.tsinghua.edu.cn

ABSTRACT

Binary code similarity detection (BCSD) is a fundamental technique for various applications. Many BCSD solutions have been proposed recently, which mostly are embedding-based, but have shown limited *accuracy* and *efficiency* especially when the volume of target binaries to search is large. To address this issue, we propose a cost-effective BCSD framework, CEBin, which fuses embedding-based and comparison-based approaches to significantly improve accuracy while minimizing overheads. Specifically, CEBin utilizes a refined embedding-based approach to extract features of target code, which efficiently narrows down the scope of candidate similar code and boosts performance. Then, it utilizes a comparison-based approach that performs a pairwise comparison on the candidates to capture more nuanced and complex relationships, which greatly improves the accuracy of similarity detection. By bridging the gap between embedding-based and comparison-based approaches, CEBin is able to provide an effective and efficient solution for detecting similar code (including vulnerable ones) in large-scale software ecosystems. Experimental results on three well-known datasets demonstrate the superiority of CEBin over existing state-of-the-art (SOTA) baselines. To further evaluate the usefulness of BCSD in real world, we construct a large-scale benchmark of vulnerability, offering the first precise evaluation scheme to assess BCSD methods for the 1-day vulnerability detection task. CEBin could identify the similar function from millions of candidate functions in just a few seconds and achieves an impressive recall rate of 85.46% on this more practical but challenging task, which are several order of magnitudes faster and 4.07× better than the best SOTA baseline.

CCS CONCEPTS

• **Security and privacy** → **Systems security**; • **Computing methodologies** → **Artificial intelligence**.

KEYWORDS

Deep Learning, Binary Analysis, Similarity Detection, Vulnerability Discovery

ACM Reference Format:
Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xi Xiao. 2024. CEBin: A Cost-Effective Framework for Large-Scale Binary Code Similarity Detection. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652117

1 INTRODUCTION

Binary code similarity detection (BCSD) is an emerging and challenging technique for addressing various software security problems. BCSD enables determining whether two binary code fragments (e.g., functions) are similar or homologous. It can be broadly adopted for many downstream tasks like 1-day vulnerability discovery [1, 5–8, 12, 14, 15, 17, 23, 35, 38, 39, 47, 48, 53, 62, 65], malware detection and classification [4, 21, 29], third-party library detection [32, 54, 69], software plagiarism detection [36, 37] and patch analysis [22, 28, 63]. BCSD's growing importance in these areas highlights its role as a versatile tool in enhancing software security.

Recently, we have witnessed numerous BCSD solutions deploying deep learning (DL) models for feature extraction and comparison [9, 10, 17, 35, 40, 41, 49, 50, 60, 62, 64, 66, 70], showing that DL models can learn features of binary functions to identify similar ones across different compilers, compilation optimization levels,

[*] Institute for Network Sciences and Cyberspace
[†] Zhongguancun Laboratory
[‡] Corresponding author: chaoz@tsinghua.edu.cn

ISSTA '24, September 16–20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652117

instruction set architectures (ISAs), or even some obfuscation techniques. Among them, the SOTA approaches [1, 26, 33, 38, 45, 58, 66] train *large assembly language models* to learn the representation of binary code.

Despite the promising progress, current DL-based BCSD solutions are facing practical challenges when applying to real-world tasks, such as detecting 1-day vulnerabilities in the software supply scenario where the volume of target binaries to match is huge. For instance, once a new vulnerability is discovered in the upstream codes, efficiently and accurately identifying which downstream software has similar code and may be affected is crucial. For such real world tasks, a large collection of functions (e.g., all functions of the software ecosystem) must be maintained and matched against the query function (e.g., the function with the 1-day vulnerability), which brings the following three primary challenges.

First, existing BCSD methods have a poor balance between accuracy and efficiency. Existing BCSD methods can be roughly classified into *comparison-based* [3, 11, 34, 35, 51] and *embedding-based* approaches [1, 9, 19, 33, 38, 40, 45, 58, 66]. Comparison-based methods build a model to take a pair of binary functions as inputs and compare their similarity directly, which often have high overheads and higher accuracy. For a given query function, it has to query the model to compare with each function in the target dataset to locate similar ones, which makes it non-scalable. On the other hand, embedding-based methods only take a single binary code as input and encode its higher-level features to an embedding space (i.e., numerical vectors), and then approximate the similarity of a given pair of functions in this embedding space using the vector distance (e.g., cosine), which are more scalable but have lower accuracy. The embedding-based approach is more efficient, since each input function only needs to be encoded once and its similar ones could be located in the embedding space with fast neighbour search algorithm. But the comparison-based approach in general has higher accuracy, since it takes a pair of binary functions as inputs and enables the model to learn pairwise features, while the embedding-based approach only takes one function as input and can only learn the feature of one function.

The second challenge is that existing BCSD methods cannot provide an acceptable accuracy performance (i.e., recall) when searching similar functions from a large pool of function sets. Pointed out in both previous study [58, 61] and our experimental results (see Section 5.1), the performance of existing BCSD declines rapidly as the scale of functions to be searched expands. The main reason is that *the training objective of these models does not match this more challenging task*. For instance, existing works typically either use supervised learning to distinguish between similar or dissimilar function pairs, or employ contrastive learning to ensure the distance between similar functions is closer. Such models are only trained to differentiate which function **from a small number of function sets** is similar to the query function. This training objective cannot be simply adapted to the large-scale function datasets such as the 1-day vulnerability detection task (e.g., millions of functions to compare in the software supply chain), since in the real-world scenarios the ratio of negative samples (i.e., dissimilar functions) is way larger than the settings of model training.

The third challenge is that the community has no large-scale accessible validation dataset for BCSD tasks, such as 1-day vulnerability detection. Existing BCSD in general only demonstrates a proof-of-concept experiment, which involves a small vulnerability dataset consisting of some CVEs (usually less than 20) [1, 9, 10, 24, 38–40, 57, 58, 64] and a number of target codes to search (e.g., a batch of IoT firmware). These methods have two drawbacks. (1) They choose different sets of CVEs, causing the search performance is not comparable. (2) They cannot evaluate the recall rate since it is impossible to determine how many vulnerabilities exist in the firmware to be tested. Note that, the recall rate is critical to ensure coverage comprehensiveness for cunderstanding how 1-day vulnerabilities affect downstream software.

To address the above challenges, we propose CEBin, a novel **C**ost-**E**ffective **Bin**ary code similarity detection framework. CEBin fuses embedding-based and comparison-based approaches to significantly improve accuracy while minimizing overheads. To improve the accuracy performance of the embedding model component, CEBin proposes a Reusable Embedding Cache Mechanism (RECM) to introduce more negative samples during model fine-tuning *by reusing the negative embeddings*. This embedding model could efficiently locate similar functions with relatively high accuracy, thus greatly narrowing down the scope of candidate similar functions. To further improve the accuracy performance, CEBin adopts an extra comparison model component, which searches similar functions among the remained candidates in a pairwise comparison manner.

Specifically, we fuse the embedding-based and comparison-based models. CEBin adopts an embedding model for speed and introduces a comparison model for accuracy. To address the inability of the comparison model to scale to large-scale functions, CEBin adopts a hierarchical approach, with the embedding model retrieving top-K functions from a large function pool, followed by the comparison model that selects the final similar ones from the top-K functions. With this inference process, we constrain the cost to be related to K. The experiments show that CEBin can increase the performance by a large margin with a high speed achieved.

For the second challenge, we propose a Reusable Embedding Cache Mechanism (RECM) to introduce more negative samples to fine-tune the embedding model. As directly adding a large number of negative samples introduces significant training cost, RECM solves this challenge by *maintaining an embedding cache of negative samples* during training and reusing previous embeddings. Then, the embedding model was trained using momentum contrastive learning [20] by splitting the encoder model into two encoders, including (1) the query encoder to get the representation of the query function, and (2) the reference encoder to get the representation of functions in the function set. In this way, CEBin does not need to record the gradient of the reference encoder during training, which significantly reduces the training costs while achieving a great improvement in the embedding model's performance.

Addressing the third challenge, we aim for an objective and comprehensive evaluation of BCSD's vulnerability detection capabilities. To this end, we chose a range of widely used libraries and software incorporating 187 vulnerabilities listed in the CVE database. We identify the vulnerable functions corresponding to each CVE and build a benchmark with 27,081,862 functions and 12,086 vulnerable functions in total. With this benchmark, we take a solid step towards evaluating BCSD schemes in real-world scenarios and help future research in this domain.

We implement CEBin and evaluate it on three well-regarded BCSD datasets. The results show that CEBin considerably surpasses existing SOTA solutions. In the BinaryCorp dataset, CEBin leads with an 84.5% accuracy in identifying functions from 10,000 candidates, surpassing the current best solution's result (i.e., 57.1%). On two more demanding cross-architecture datasets, CEBin attains 94.6% and 87.0% Recall@1, respectively, significantly outperforms the best baselines (i.e., 9.6% and 10.9%). Additionally, we conduct experiments on a large-scale cross-architecture 1-day vulnerability detection task and obtain a recall of 85.46%, which is 4.07× greater than the SOTA. In summary, our contributions are as follows:

- We propose a cost-effective BCSD framework CEBin, which fuses embedding-based and comparison-based approaches in a hierarchical inference pipeline to significantly improve accuracy performance while maintaining efficiency.
- We propose a Reusable Embedding Cache Mechanism (RECM) to enhance the performance of embedding models while preserving efficient training.
- We construct a large benchmark of vulnerabilities and binary functions, offering a precise evaluation scheme to assess BCSD methods for the 1-day vulnerability detection task.
- We conduct thorough experiments and demonstrate the outstanding performance of CEBin for large-scale BCSD tasks, which could identify the similar function from millions in just a few seconds and achieve an impressive average recall of 85.46%.
- We release our code and the large benchmark of vulnerabilities to the research community to facilitate future research[1].

2 BACKGROUND AND RELATED WORKS

2.1 Binary Code Similarity Detection (BCSD)

BCSD technique is utilized to identify the similarities between binary code fragments such as functions. BCSD can be adopted for many tasks like vulnerability detection, malware classification, and code plagiarism detection. One of the most challenging tasks is the software supply chain vulnerability detection [38, 58]. For instance, once a 1-day vulnerability is discovered in a widely-used foundational open-source component, efficiently and accurately locating the affected downstream software (mostly only binaries without source code) as comprehensive as possible is crucial.

Various BCSD approaches have been investigated, including graph matching [16, 71], tree-based methods [48], and feature-based techniques [13, 42]. Recently, deep learning techniques have emerged as popular methods for BCSD for their accuracy and ability to learn complex features automatically. In deep learning models, two primary methods can be distinguished: embedding-based and comparison-based approaches (see overview in Figure 1).

2.1.1 *Embedding-based Approaches.*

The recent development of deep neural networks (DNNs) has inspired researchers to delve into embedding-based BCSD. Embedding-based BCSD methods primarily focus on extracting features from functions and represent them in a lower dimension space (i.e., "embedding"). Prior research has employed DNNs as feature extractors for transforming binary functions into an embedding space. To determine similarity, these functions' embeddings can be compared using distance metrics. One advantage of the embedding-based approach is the use of fixed representations that can be precomputed. When calculating the similarity between a new function and existing ones, only the new function's embedding must be extracted for distance measurement.

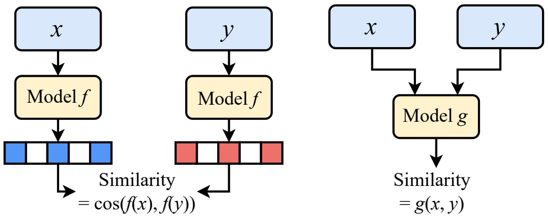

Figure 1: The embedding-based model (left) represents functions x, y as embeddings and calculate similarity with similarity metrics (e.g cosine). The comparison-based model (right) takes a pair of functions and outputs their similarity.

Genius [15] and Gemini [62] employ clustering and graph neural networks (GNNs) for functional vectorization but are hampered by capturing limited semantics on control flow graph (CFG), akin to SAFE's [40] approach marred by out-of-vocabulary (OOV) challenges. Extending beyond these confines, subsequent models like GraphEmb [41], OrderMatters [66] and CodeCMR [67] utilize deep neural networks to encode semantic information, with Asm2Vec [9] addressing the CFG's structural nuances through unsupervised learning. This progress sets the stage for the integration of advanced pre-trained models such as jTrans [58], Trex [45], BinShot [2] and VulHawk [38] which leverage these models' capabilities to enhance the understanding and identification of binary code functionalities.

2.1.2 *Comparison-based Approaches.*

These approaches in binary analysis directly measure function similarity using raw data or feature analysis. FOSSIL [3] integrates Bayesian networks to assess free open-source software functions through syntax, semantics, and behavior. In contrast, α-Diff [35] applies CNNs to raw bytes, requiring extensive training data. BinDNN [31] combines CNN, LSTM, and deep neural networks to ascertain function equivalence across compilers and architectures, while another work [52] decompose code into fragments for fast, accurate analysis using feed-forward neural networks. GMN [34] introduces a cross-graph attention mechanism within its DNN model for graph matching to evaluate similarity scores between graphical elements.

2.1.3 *Summary of Existing Approaches.*

The embedding-based approach has become mainstream in BCSD research in recent years. Compared to the comparison-based approach, it can provide efficient inference and *is therefore suitable for scaling to large-scale BCSD application scenarios, but has lower accuracy.* On the one hand, a recent paper [39] measured that the comparison-based model GMN [34] achieved the best performance among all publicly available BCSD solutions. On the other hand, we can infer this result in a theoretic way. As shown in Figure 1, given raw inputs x and y of two binary codes, the embedding-based model learns f and uses distance metrics (e.g. cosine) to calculate similarity as $\cos(f(x), f(y))$, while the comparison-based model learns g and

[1] https://github.com/Hustcw/CEBin

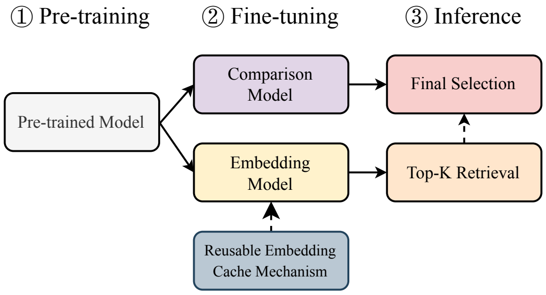

Figure 2: The Workflow of CEBin.

calculates similarity as $g(x, y)$. It is obvious that, $g(x, y)$ **is more expressive than** $\cos(f(x), f(y))$. In other words, a well-trained comparison model can outperform a well-trained embedding model, to better capture the features between two binary codes.

2.2 Contrastive Learning

The goal of contrastive learning is to increase the similarity between semantically similar data points, which are called embeddings, while increasing dissimilarity between semantically unrelated data points in the latent representation space. This is achieved by using pairwise comparison in unsupervised or self-supervised manners, measuring instance distance using a contrastive loss function. For instance, Trex [45] employs a pairwise loss function to minimize the distance between the ground truth. Some previous works [34, 58, 59, 67, 68] utilize triplet loss to reduce the distance between positive pairs and increase the distance between negative pairs. SAFE [40] and OrderMatters [66] implement the output of a Siamese network as a loss function, minimizing the distance between positive pairs. Vulhawk [38], BinaryAI [25] and CLAP [56] apply cross-entropy loss to reduce the distance between ground truth and maintain distance from negative pairs using a many-to-many approach.

3 METHODOLOGY

3.1 Overview of the Framework

The CEBin framework operates in three primary stages: pre-training, fine-tuning, and inference, depicted in Figure 2. In the pre-training phase, we utilize a comprehensive dataset to train a language model for representing binary code. During fine-tuning, this pre-trained language model is further refined to produce two distinct models: an embedding model and a comparison model. A notable enhancement during this stage is the integration of the Reusable Embedding Cache Mechanism (RECM), designed to introduce a plethora of negative samples for the fine-tuning of the embedding model. In the final inference phase, we employ the embedding model to retrieve the top K candidate functions closest to the query function. Subsequently, the comparison model facilitates precise final selections.

3.2 Pre-training

3.2.1 Data Preparation. We use three datasets, BinaryCorp [58], Cisco [39], and Trex [45] as our pre-training corpus. We employ BinaryNinja[2] following PalmTree [33] to extract functions and lift the functions to BinaryNinja's Intermediate Language (IL) to normalize binary functions across various ISAs. We use the WordPiece [30] algorithm to train a tokenizer on the whole assembly code datasets and perform a lossless encoding of assembly code without normalization on string and number, solving the problem of Out-of-Vocabulary (OOV).

3.2.2 Model Architecture. The base architecture of our model is Transformer [55] because both previous work [39] and our evaluation results in Section 5.1 of the baselines indicate that Transformer-based methods outperform other deep learning approaches. Because jTrans [58] performs best in our evaluation, we choose to use jTrans as the base model and utilize the same pretraining tasks.

3.3 Fine-tuning

The fine-tuning process is divided into two stages as shown in Figure 3. Stage 1 focuses on training the embedding model, while Stage 2 trains the comparison model.

3.3.1 RECM Integrated Embedding Model Training. As mentioned in the Section 1, the second challenge points out that the main issue with the current SOTA methods is that the training objective of these approaches does not match the more challenging real-world scenarios. Note that introducing more negative samples is essential to improving the model's discrimination capabilities. A straightforward approach would be to directly sample a large number of negative examples during the training phase of the embedding model for contrastive learning. However, adding a large number of negative samples in an end-to-end training manner requires substantial computational resources, such as a vast amount of GPUs.

To address this challenge, we propose a Reusable Embedding Cache Mechanism (RECM) to reuse previously encoded embeddings. We first split the embedding model into a query encoder and a reference encoder. The training data is formatted as a pair (Q_i, R_i) fed into the model, where Q_i and R_i respectively represent the query function and the reference function, and they are semantically equivalent because they are compiled from the same source code. As shown in the stage 1 of Figure 3, after being encoded by the embedding model, the query function and the reference function are encoded into embeddings, represented as $Q_{1:n}$ and $R_{1:n}$ respectively. We then retrieve the embeddings $R'_{1:L}$ in the embedding cache, the size of the ebmedding cache is denoted as L. We compute the dot product between $Q_{1:n}$ and $Concate(R_{1:n}, R'_{1:L})$. Only the pairs $Q_{1:n} \cdot R_{1:n}$ are positive samples, while all others are negative samples. After updating the embedding model, the embedding cache will also be updated by the newly encoded reference functions $R_{1:n}$.

While the query encoder is updated by gradients, the reference encoder is freezed during encoding and then updated using a momentum-based approach [20]. When fine-tuning the embedding model, we apply the InfoNCE loss [43] to maximize the mutual information between positive pairs and negative samples. The InfoNCE loss, given a positive pair (Q_i, R_i) and a set of negative pairs

[2] https://binary.ninja

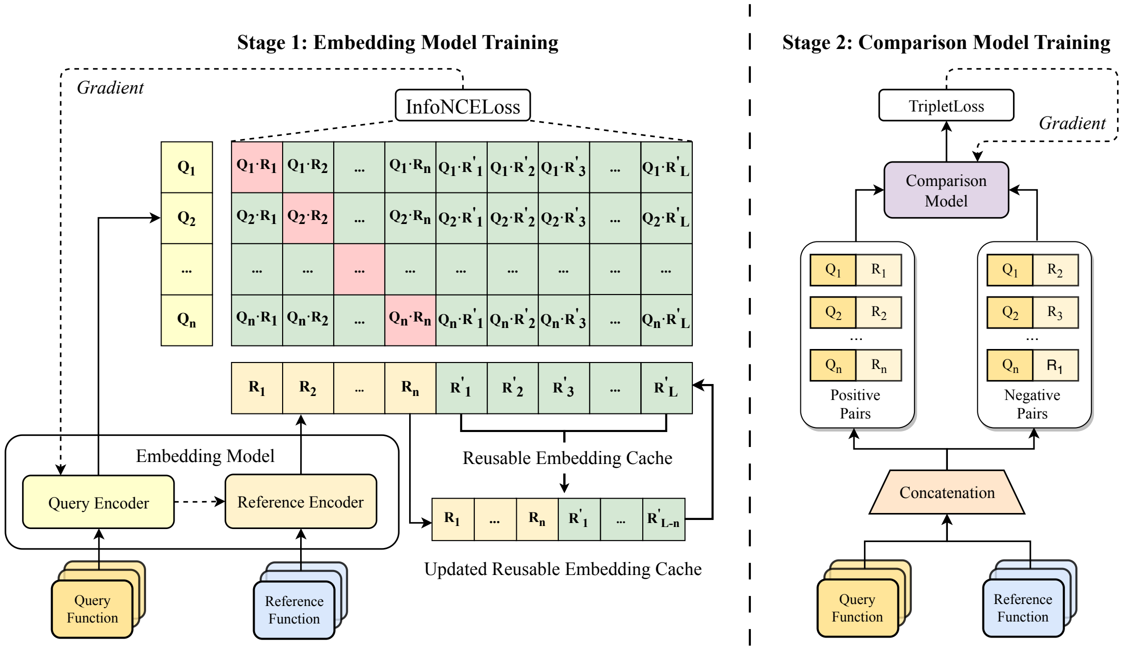
Stage 1: Embedding Model Training
Gradient
InfoNCELoss
Q1
Q2
...
Qn
Q1·R1 Q1·R2 ... Q1·Rn Q1·R'1 Q1·R'2 Q1·R'3 ... Q1·R'L
Q2·R1 Q2·R2 ... Q2·Rn Q2·R'1 Q2·R'2 Q2·R'3 ... Q2·R'L
Qn·R1 Qn·R2 ... Qn·Rn Qn·R'1 Qn·R'2 Qn·R'3 ... Qn·R'L
R1 R2 ... Rn R'1 R'2 R'3 ... R'L
Reusable Embedding Cache
R1 ... Rn R'1 ... R'L-n
Updated Reusable Embedding Cache
Embedding Model
Query Encoder
Reference Encoder
Query Function
Reference Function
Stage 2: Comparison Model Training
TripletLoss
Gradient
Comparison Model
Q1 R1
Q2 R2
Qn Rn
Positive Pairs
Q1 R2
Q2 R3
Qn R1
Negative Pairs
Concatenation
Query Function
Reference Function

Figure 3: The illustration of fine-tuning phase for CEBin. In stage 1, semantically equivalent function pairs (Q_i, R_i) are encoded with query encoder and the reference encoder respectively. The corresponding pairs (Q_i, R_i) are considered as positive pairs. And other pairs $(Q_i, R_j)_{i \neq j}$ along with all pairs (Q_i, R'_j) containing previous reference functions in the Resuable Embedding Cache are considered as negative paris. The InfoNCELoss is calcuated given positive pairs and massive negative pairs. The loss is back-propagated to update query encoder and momentum is used to update the refernece encoder. In stage 2, pairs of functions are feed into model simultaneously after concatenation. (Q_i, R_i) is considered as a positive pair and (Q_i, R_{i+1}) is considered as a negative pair. Then we use the triple loss to train the comparison model.

$(Q_i, R_j)_{j \neq i}$, is defined as:

$$\mathcal{L}_E = -\log \frac{\exp(f(Q_i, R_i))}{\sum_{j=1}^{N} \exp(f(Q_i, R_j))}, \tag{1}$$

where N is the total number of pairs, and $f(\cdot,\cdot)$ is the similarity function between two embeddings. We denote the parameters of query encoder and reference encoder as θ_q and θ_r respectively. We use momentum to update the reference encoder at the same time:

$$\theta_r \leftarrow m\theta_r + (1 - m)\theta_q \tag{2}$$

where m is the momentum coefficient and is usually set large (e.g., 0.99). During the training of the embedding model, we only update θ_q with back-propagation.

With the integration of RECM, we can enlarge the size of training batches and introduce large negative samples with increasing tiny training costs. Compared to not integrating RECM, when the number of reference functions reaches $N = n + L$, training one step requires an increase of L/n times in forward and backward computations, and the memory usage also increases by approximately L/n times. For instance, in experiments of Section 5 where we used 8 V100 GPUs for training with $n = 128$ and $L = 8,192$. Without integrating RECM, it would require about 512 V100 GPUs which is extremely expensive. Figure 8 shows the impact of the size of the embedding cache on the performance of the embedding model, which shows that our design can greatly enhance the performance of the embedding model.

3.3.2 *Comparison Model Training.*

Our motivation for introducing the comparison model is inspired by image similarity detection scenarios. The direct comparison method enables the model to compare instances side-by-side, allowing for a token-by-token comparison of the functions. This approach is more precise for similarity detection tasks than the indirect comparison method.

To train the comparison model, we initialize it with the pre-trained model and modify the input to accept a pair of functions simultaneously. The output of the comparison model represents the similarity between the given pair of functions. During training, we input a batch of positive pairs, where Q_i and R_i form a positive pair, and Q_i and R_{i+1} serve as a negative pair. We concatenate the function pairs and provide them as input to the model. We then use the triplet loss to train the comparison model to discriminate between positive and negative pairs effectively, which can be formulated as:

$$\mathcal{L}_C = \max(0, -D(Q_i, R_i) + D(Q_i, R_{i+1}) + \alpha), \tag{3}$$

where $D(\cdot,\cdot)$ represents the similarity score output by comparison model and α is the margin of the positive and negative pairs. By combining the fine-tuning of the embedding model in Stage 1 with the introduction of the comparison model in Stage 2, we accommodate the domain-specific requirements of the BCSD task and improve the model's ability to discern the similarity between binary code pairs effectively.

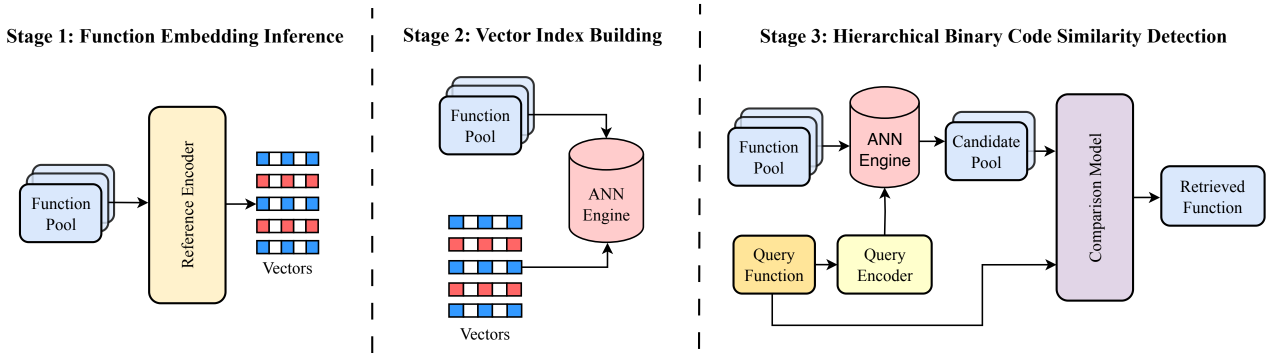

Figure 4: The illustration of inference for CEBin. In stage 1, we use a reference encoder to encode all functions we aim to compare into vectors. In stage 2, we build a vector index for each function and its corresponding vector using ANN algorithm, so that we can retrieve K most similar vectors given a query vector. In stage 3, given a query function, we use a query encoder to obtain the embedding vector and retrieve the top-K closest functions from the function pool using a pre-built vector index. Then the K candidate functions along with the query function are fed into the comparison model to perform the final selection.

3.4 Inference

The CEBin inference process has three stages as shown in Figure 4. The three stages perform function embedding inference, vector index building, and binary code similarity detection, respectively. This hierarchical design aims to balance performance and inference cost, integrating the advantages of both the embedding model and the comparison model.

Stage 1: Function Embedding Inference. In the first stage, we use the embedding model to construct embedding vectors for each function within the function pool that we aim to compare. As shown in Figure 4, we employ the reference encoder from the embedding model to generate vectors for the functions in the function pool.

Stage 2: Vector Index Building. In the second stage, we build an index for the embedding vector of each function in the function pool. We use the approximate nearest neighbor (ANN) algorithm to enable efficient inference. ANN approximates the nearest neighbors in high-dimensional spaces, allowing for fast similarity search and comparison among the function pool's embedding vectors. By building a vector index using ANN, our model can handle large-scale BCSD scenarios in a resource-efficient manner.

Stage 3: Hierarchical Binary Code Similarity Detection. The third stage of the inference process involves utilizing the fine-tuned embedding model and the comparison model from CEBin to perform BCSD. Given a query function, we first use the embedding model to retrieve the top-K closest functions from the function pool using the ANN-based vector index. This helps to narrow down the most similar functions while maintaining high efficiency. After obtaining the k candidate functions, we use the comparison model to do BCSD with the query function and candidate functions.

The hierarchical combination of the embedding model and the comparison model ensures that our BCSD method is both efficient and accurate. The computational cost of concatenating the query function and top-K candidate functions as the comparison model's input is manageable and does not increase substantially as the size of the function pool grows. This allows CEBin to effectively identify the most similar binary code sequences within massive functions.

Our extensive experiments in Section 5 show that CEBin achieves better performance without incurring a dramatic increase in computational cost through the hierarchical inference framework. Furthermore, the comparison model significantly enhances the detection results by providing a fine-grained similarity assessment.

4 EXPERIMENTAL SETUP

We compare CEBin against multiple baselines: Genius [15], Gemini [62], SAFE [40], Asm2Vec [9], GraphEmb [41], OrderMatters [66], Trex [46], GNN and GMN [34], and jTrans [58]. We implement CEBin and baselines using Faiss [27] and Pytorch [44]. Our experiments are conducted on several servers to accelerate training. The GPU setup includes 8 Nvidia-V100. The experiment environment consists of three Linux servers running Ubuntu 20.04 with Intel Xeon 96-core and equipped with 768GB of RAM.

We evaluate CEBin with the following three datasets. We label functions as equal if they share the same name and were compiled from the same source code.

- **BinaryCorp** [58] is sourced from the ArchLinux official repositories and Arch User Repository. Compiled using GCC 11.0 on X64 with various optimizations, it features a highly diverse set of projects.
- **Cisco Dataset** [39] comprises seven popular projects, it yields 24 distinct libraries upon compilation. Binaries in the Cisco dataset are compiled with GCC and Clang compilers, spanning four versions each, across six ISAs (x86, x64, ARM32, ARM64, MIPS32, MIPS64) and five optimization levels (O0-O3, Os). This setup allows for cross-architecture analysis and evaluation of compiler versions, with a moderate number of projects.
- **Trex Dataset** [46] is built upon binaries released by [46], which consists of ten libraries chosen to avoid overlap with the Cisco dataset. Similar to the Cisco Dataset, the Trex dataset facilitates cross-architecture and cross-optimization evaluation.

Many previous work [38–40, 46, 62] use the area under curve (AUC) of the receiver operating characteristic (ROC) curve or precision to evaluate the performance of BCSD solutions. But these metrics are too simple for existing solutions so SOTA BCSD solutions

Table 1: Comparison between CEBin and baselines for the cross-optimization task on BinaryCorp-3M (Poolsize=10,000)

	MRR							Recall@1						
Models	**O0,O3**	**O1,O3**	**O2,O3**	**O0,Os**	**O1,Os**	**O2,Os**	**Average**	**O0,O3**	**O1,O3**	**O2,O3**	**O0,Os**	**O1,Os**	**O2,Os**	**Average**
Genius	0.041	0.193	0.596	0.049	0.186	0.224	0.214	0.028	0.153	0.538	0.032	0.146	0.180	0.179
Gemini	0.037	0.161	0.416	0.049	0.133	0.195	0.165	0.024	0.122	0.367	0.030	0.099	0.151	0.132
GNN	0.048	0.197	0.643	0.061	0.187	0.214	0.225	0.036	0.155	0.592	0.041	0.146	0.175	0.191
GraphEmb	0.087	0.217	0.486	0.110	0.195	0.222	0.219	0.050	0.154	0.447	0.063	0.135	0.166	0.169
OrderMatters	0.062	0.319	0.600	0.075	0.260	0.233	0.263	0.040	0.248	0.535	0.040	0.178	0.158	0.200
SAFE	0.127	0.345	0.643	0.147	0.321	0.377	0.320	0.068	0.247	0.575	0.079	0.221	0.283	0.246
Asm2Vec	0.072	0.449	0.669	0.083	0.409	0.510	0.366	0.046	0.367	0.589	0.052	0.332	0.426	0.302
Trex	0.118	0.477	0.731	0.148	0.511	0.513	0.416	0.073	0.388	0.665	0.088	0.422	0.436	0.345
jTrans	0.475	0.663	0.731	0.539	0.665	0.664	0.623	0.376	0.580	0.661	0.443	0.586	0.585	0.571
CEBin-E	0.787	0.874	0.924	0.858	0.909	0.893	0.874	0.710	0.818	0.885	0.795	0.863	**0.842**	0.819
CEBin	**0.850**	**0.886**	**0.953**	**0.903**	**0.927**	**0.895**	**0.902**	**0.776**	**0.826**	**0.920**	**0.839**	**0.874**	0.834	**0.845**

perform similarly. However, we notice that previous works [39, 58] announced that ranking metrics, the mean reciprocal rank (MRR), and the recall (Recall@K) are more practical for BCSD especially when the size of the function pool becomes very large. Therefore, we use MRR and Recall@1 following [58] to evaluate and compare the performance of CEBin and the baseline methods.

For CEBin's inference phase, we retrieve the top-50 closest function for experiments on BinaryCorp, Cisco, and Trex datasets. We choose K=50 because we find that the embedding model's Recall@50 is almost close to 1.0 as shown in the Section 5.2.1, thus providing a sufficiently good set of candidate functions for the comparison model. We retrieve top-300 closest function for the vulnerability search experiments because the maximum number of vulnerable functions could be up to 240 for each query.

5 EVALUATION

To prove CEBin's effectiveness in addressing previous challenges, we propose these research questions (RQs):

- **RQ1:** How does CEBin perform compared to SOTA BCSD solutions in different settings, including cross-architecture, cross-compilers, and cross-optimizations?
- **RQ2:** How do the design choices within the CEBin framework contribute to the overall performance?
- **RQ3:** How does CEBin perform in vulnerability search over a challenging vulnerability searching benchmark?
- **RQ4:** How is the generalization ability of the CEBin?
- **RQ5:** What is the inference time cost of CEBin compared with other SOTA baselines?

5.1 Performance (RQ1)

5.1.1 Cross-Optimizations: BinaryCorp. In this experiment, we assess CEBin's performance on the BinaryCorp dataset, which includes x64 binaries compiled with GCC-11 across various optimization levels (O0, O1, O2, O3, and Os). We conduct extensive experiments to evaluate the performance of the selected baselines, which are limited to a single architecture and those supporting cross-architecture. We evaluate the performance of cross-optimization BCSD tasks with varying difficulty optimization pairs (e.g., O0 v.s. O3) while maintaining consistent experimental setups with previous work [58] for fair comparison. We report the experimental results for function poolsize 10,000 as shown in Tables 1. CEBin-E denotes for the embedding model of CEBin.

The experimental results in Table 1 demonstrate that CEBin significantly outperforms all baselines. CEBin outperforms the best-performing baseline jTrans by significantly improving MRR by 44.8% and Recall@1 by 47.9%. The experimental results show the advantage of CEBin in cross-optimization tasks, improving the effectiveness of the embedding model training by using more negative samples during training.

5.1.2 Cross-Architectures, Compilers, and Optimizations: Cisco and Trex Dataset. We evaluate CEBin and baselines on Cisco and Trex datasets across various factors, including architectures, compilers, optimizations, and their combinations. In this experiment, we select several cross-architecture baselines for comparisons, such as GNN, and Trex. Consistent with previous work, we train CEBin and GNN on Cisco's training set and assess performance on Cisco's test set. As previous research [39] highlights, retraining Trex on Cisco dataset is challenging, we directly use the model released by Trex authors.

To ensure comprehensive tests, we employ six evaluation tasks. (1) XO refers to function pairs with varying optimizations but identical compiler, compiler version, and architecture. (2) XC refers to function pairs with different compilers but the same architecture and optimization. (3) XA refers to function pairs with varying architectures but identical compiler, compiler version, and optimization. (4) XC+XO refers to function pairs with various compilers and optimizations but the same architecture. (5) XA+XO refers to function pairs with varying architectures and optimizations but identical compiler and compiler versions. (6) XA+XC+XO refers to function pairs from any architecture, compiler, compiler version, and optimization. We test the six tasks on the Cisco Dataset. We test tasks (1), (3), and (5) on the Trex dataset since it only uses the GCC 7.5 compiler. We evaluate performance in a more challenging scenario where poolsize=10,000 compared to previous works.

Table 2–3 report the experimental results. The results reveal that CEBin significantly outperforms baselines in cross-architecture, cross-compiler, and cross-optimization tasks. Compared to the best baseline Trex, the MRR increases from 0.124 to 0.961, and Recall@1 increases from 0.096 to 0.946. On the Trex Dataset, CEBin outperforms the best result of the baseline, with MRR increasing from 0.175 to 0.911 and Recall@1 increasing from 0.109 to 0.870.

Table 2: Results of different binary similarity detection approaches on Cisco (poolsize=10,000)

	MRR						Recall@1					
Models	**XA**	**XC**	**XO**	**XA+XO**	**XC+XO**	**XA+ XC+XO**	**XA**	**XC**	**XO**	**XA+XO**	**XC+XO**	**XA+ XC+XO**
GNN	0.205	0.158	0.104	0.119	0.189	0.093	0.129	0.104	0.080	0.084	0.165	0.063
Trex	0.085	0.401	0.410	0.145	0.313	0.124	0.052	0.341	0.360	0.113	0.268	0.096
CEBin-E	0.760	0.907	0.859	0.817	0.866	0.766	0.692	0.871	0.816	0.766	0.823	0.706
CEBin	**0.977**	**0.992**	**0.973**	**0.978**	**0.984**	**0.961**	**0.968**	**0.988**	**0.963**	**0.969**	**0.977**	**0.946**

Table 3: Results of different binary similarity detection approaches on Trex (poolsize=10,000)

	MRR			Recall@1		
Models	**XA**	**XO**	**XA+XO**	**XA**	**XO**	**XA+XO**
Trex	0.142	0.218	0.175	0.065	0.123	0.107
GNN	0.163	0.148	0.151	0.145	0.102	0.109
CEBin-E	0.612	0.646	0.576	0.509	0.553	0.474
CEBin	**0.911**	**0.933**	**0.911**	**0.882**	**0.906**	**0.870**

The results demonstrate CEBin's advantages in challenging BCSD tasks. Training more efficiently on a larger quantity of negative samples enables the embedding model to perform better. As our training goal involves discriminating similar binary codes in larger batches of negatives, CEBin significantly outperforms the baseline, especially for this challenging large poolsize settings. Additionally, integrating the comparison model further enhances performance as it achieves more fine-grained similarity detection. In the XA+XO experiment conducted on the Trex dataset, the comparison model significantly improves Recall@1 from 0.474 to 0.870.

5.1.3 The Impact of Poolsize. As indicated in Section 1, for practical tasks like 1-day vulnerability detection in software supply chains, maintenance of a particularly large poolsize is necessary and valuable. However, in our prior experiments, we discover that as poolsize increases, performance declines across the three datasets. Thus, we explore the influence of different poolsize while maintaining other settings. The poolsize is set as $2^i, i \in [1, 13]$, and 10,000. We record Recall@1 for different poolsize.

The Figure 5–7 presents the results, clearly showing that as the poolsize increases, the relative performance of all baselines is inferior to CEBin. Furthermore, the decline in the performance of CEBin is not so obvious which suggests that CEBin is more capable of addressing large poolsize settings. The results also show that CEBin offers a greater performance enhancement compared to CEBin-E in more difficult scenarios such as O0 and O3 optimization options in the BinaryCorp experiment and the XA+XC+XO in the Cisco dataset, where binary functions exhibit larger discrepancies. CEBin, trained on the Cisco dataset, displays remarkable poolsize robustness on the Trex dataset, demonstrating outstanding generalization performance. Finally, we emphasize that when the poolsize is small (e.g., poolsize=2), the difference in recall@1 among different methods is tiny, indicating results measured with a very small poolsize in many existing works cannot accurately represent these BCSD solutions' performance in real-world.

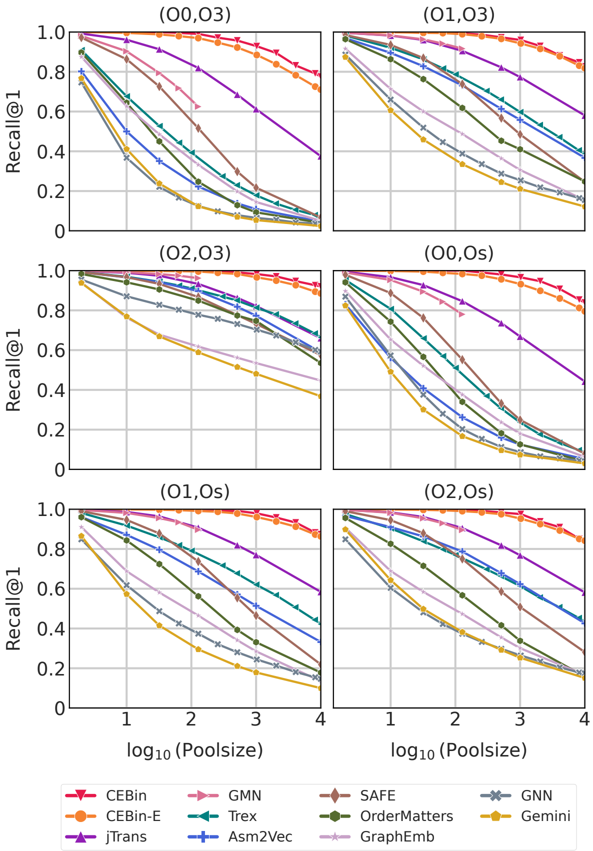

Figure 5: The performance of different binary similarity detection methods on BinaryCorp. The x-axis is logarithmic and denotes the poolsize.

5.2 Impact of Our Design Choices (RQ2)

In this section, we aim to validate the effects of our two core designs: introducing more negative samples through RECM during training and hierarchical inference on performance.

5.2.1 Reusable Embedding Cache Mechanism. To investigate the impact of the number of negative samples during training, we keep other parts of the embedding model training consistent and only change the size of the RECM, also represents the number of negative samples, used during training. We set L, the size of RECM, as powers of 2 ranging from 2 to 65536. Then we evaluate the Recall@1 for different optimization pairs at a poolsize of 10,000 on BinaryCorp. The experimental results are displayed in Figure 8, where the x-axis represents the poolsize (a logarithmic axis).

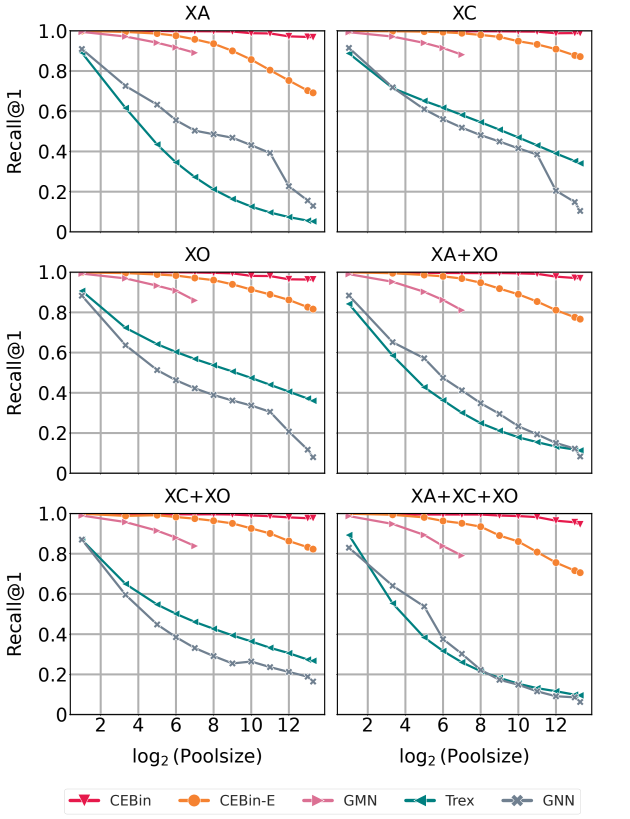

Figure 6: The performance of different binary similarity detection methods on Cisco Dataset. The x-axis is logarithmic and denotes for the poolsize.

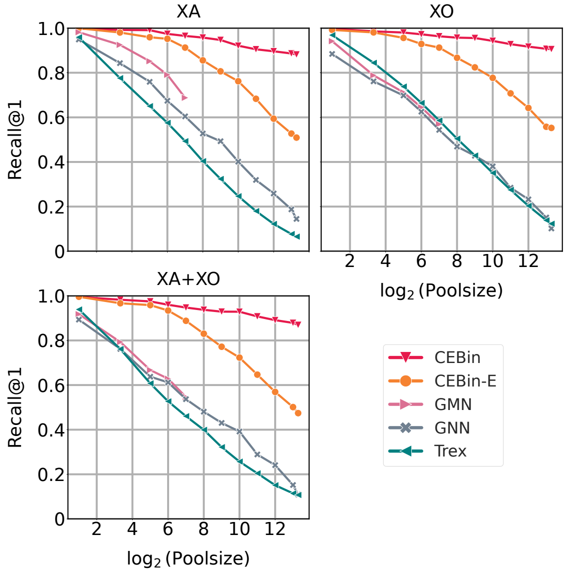

Figure 7: The performance of different binary similarity detection methods on Trex Dataset. The x-axis is logarithmic and denotes for the poolsize.

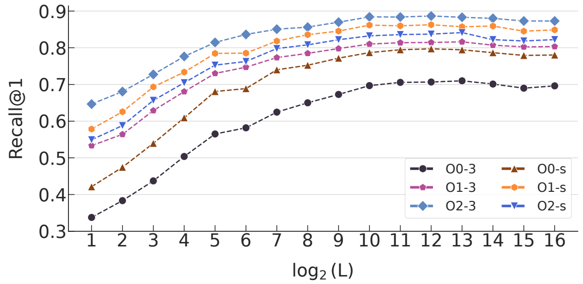

Figure 8: The performance of `CEBin-E` on BinayCorp using different size of embedding cache.

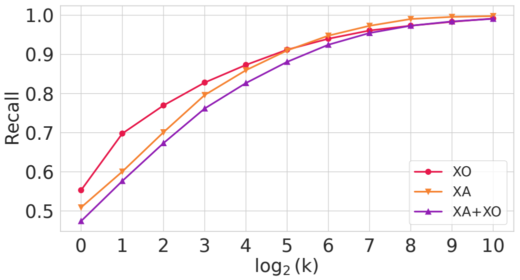

Figure 9: Recall@K of `CEBin-E` on Trex Dataset for poolsize=10,000.

Based on the experimental results, we observe that an increased number of negative samples substantially improves the overall effectiveness of the embedding model across various cross-optimization tasks. For example, in the most challenging task (O0 and O3), Recall@1 increases from 0.337 (L=2) to 0.709 (L=8192). In less challenging scenario such as comparing O2 and O3, recall@1 rose from 0.647 (L=2) to 0.887 (L=4096). Interestingly, we found that a larger L does not always lead to better results. This can be attributed to the continuous update of encoders during the encoding process where RECM is employed for training. Although a large momentum maintains a slow update to ensure consistency of embeddings in the embedding cache, excessive reuse with an exceedingly large size L introduces inconsistencies that slightly reduce performance. According to experimental results, performance improvement begins to dwindle when L exceeds 1024. A comparative analysis revealed that the optimal average performance among the six cases is attained at L=8,192 with an average recall@1 of 0.819. The experimental results verify that the integration of RECM significantly improves the performance.

5.2.2 Comparison Model. To investigate the role of the comparison model in `CEBin`, we examine both `CEBin` and `CEBin-E` across all RQ1 experiments, where `CEBin-E` employs only the embedding model. The enhancement of `CEBin` in relation to `CEBin-E` indicates the point of the comparison model during hierarchical inference. Results are shown in Tables 1–3 and Figures 5–7.

Our findings reveal that the comparison model delivers performance gains in cross-architecture, cross-compiler, and cross-optimization contexts, with larger improvements observed in more challenging tasks. In the cross-optimization task of the BinaryCorp dataset with poolsize=10,000, the comparison model boosts the average Recall@1 by 3.0%, with the most arduous O0, O3 task elevating Recall@1 by 9.3%. For the XA+XC+XO task in the Cisco dataset when poolsize=10,000, CEBin's Recall@1 rises by 34.0% compared to CEBin-E. In the XA+XO task on the Trex dataset when poolsize=10000, Recall@1 increases from 0.474 to 0.870, which is an 83.5% improvement. As CEBin is trained on Cisco and the training set lacks the GCC7.5 compiler, the Trex dataset represents an out-of-distribution (OOD) dataset. The performance significantly declines with only the embedding model, yet incorporating the comparison model markedly heightens CEBin's robustness.

To show the potential improvement the comparison-based model can bring, Figure 9 presents the Recall@K of CEBin-E on the Trex dataset. The experimental result shows that the Recall@50 of CEBin-E is significantly higher than Recall@1, which indicates the potential improvements that can be brought about by using the comparison model. However, the Recall@50 of CEBin-E is only slightly higher than the Recall@1 of CEBin, which suggests that our comparison model performs very well. The experiments demonstrate that the comparison model effectively learns more intricate features, augmenting BCSD performance. It's worth noting that, the Recall@50 of the embedding model is almost convergent. Considering the balance between overhead and performance, we chose K=50 for all the experiments presented earlier as the candidate results are good enough for comparison model.

5.3 Vulnerability Detection (RQ3)

Previous works [18, 38–40, 58] indicate that collecting high-quality vulnerability datasets is challenging. To develop a realistic vulnerability dataset, we gather commits that fix 187 CVEs from 5 projects[3]. These projects are compiled using multiple compilers and architectures to represent the diverse configurations found in real-world software supply chains. We provide details about the dataset including the number of vulnerable functions and the poolsize corresponding to each CVE search in our released code. We first pinpoint relevant functions within commits addressing CVEs by analyzing their root causes, given the CVE and its associated commit. Next, we evaluate CEBin by establishing a search pool comprising all compiled functions. We then select one vulnerable function to serve as a query and attempt to identify other instances of the same vulnerability in the entire pool. Given t total vulnerable functions, we extract the t most similar matches to the query and determine the number of these (denoted as m) that are indeed vulnerable. Finally, we calculate the recall rate by $\frac{m}{t}$, which allows us to assess the effectiveness of different approaches.

Result Analysis In Figure 10, we present the recall rate distribution for CEBin, GNN, and Trex on a vulnerability dataset. Among 187 CVEs, CEBin achieves an average recall rate of 86.46%, while Trex and GNN have average recall of 15.89% and 16.85%, respectively. The recall rate distribution reveals that most of CEBin's recall rates fall within a range greater than 0.9. In contrast, the recall rates for Trex primarily fall within the 0.05 to 0.15 range. This observation implies that for large poolsize vulnerability searches, Trex and GNN experience a significant decrease in performance, while the impact on CEBin is relatively small. As a result, CEBin significantly outperforms both Trex and GNN. For instance, in CVE-2012-0884, CVE-2013-4353, CVE-2015-0288, and CVE-2015-1794, CEBin successfully identifies all vulnerabilities within poolsizes of over 60,000. Meanwhile, Trex and GNN could only detect less than 25% of the vulnerable functions in these cases.

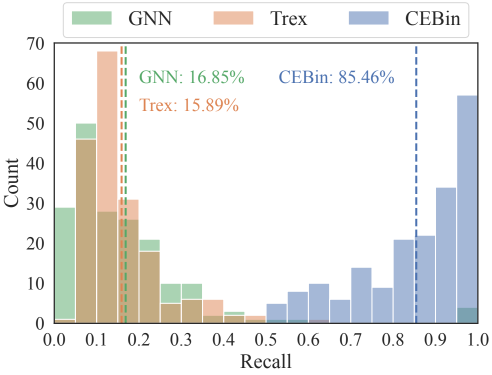

Figure 10: The Vulnerability Search Results of CEBin and Trex. This figure represents the distribution of recall values of different methods. The dashed lines show the mean recall of different models.

We investigate various high-ranking failure cases in this experiment and identify several potential causes behind these failures.

- **Truncation of excessively long functions.** Functions with tokens exceeding the maximum length are truncated when embedding. If two functions are similar before truncation, one might be misidentified as similar to the other after truncation. For example, in OpenSSL's CVE-2008-1672, the model confuses ssl3_send_client_key_exchange (vulnerable) with dtls1_send_client_key_exchange as they share highly similar semantic information within the maximum token length.
- **Insensitivity to specific instructions.** If functions only differ by very few instructions and the model cannot adequately distinguish them, it may consider them as similar. In OpenSSL's CVE-2012-2110, the model treats CRYPTO_realloc_clean (vulnerable) and CRYPTO_realloc as alike, despite the latter having three fewer consecutive function calls.
- **Function name altered during compilation.** Compiler may modify function names leading to misjudgments. In curl's CVE-2022-27780, the model identifies the vulnerable function as hostname_check.isra.1 instead of hostname_check. We confirm that the misjudgment stems from the compiler changing the function name with manual checks.

5.4 Generalization Performance (RQ4)

As previously discussed and shown in Table 3, and Figure 7, after training on the Cisco Dataset, we observe that CEBin achieves excellent results on the Trex Dataset, demonstrating its strong generalization capability. To further investigate the generalization of CEBin, we conduct an even more challenging experiment. We label the models trained separately on BinaryCorp and Cisco as CEBin-BinaryCorp and CEBin-Cisco. Notably, BinaryCorp is a cross-optimization (XO) dataset compiled on x64 using one compiler, GCC

[3] curl, vim, libpng, openssh and openssh-portable

Table 4: Recall@1 comparison between CEBin-Cisco and CEBin-BinaryCorp of cross-optimization task on Trex dataset for poolsize=10,000.

Architecture	Model		Improvement
	CEBin-Cisco	CEBin-BinaryCorp	
x86	0.907	0.948	↑ 0.041
x64	0.889	0.945	↑ 0.056
MIPS32	0.851	0.885	↑ 0.034
MIPS64	1.000	1.000	–
ARM32	0.862	0.909	↑ 0.047
ARM64	0.925	0.917	↓ 0.008
Average	0.906	**0.934**	**↑ 0.028**

Table 5: Inference speed (seconds) of GNN, Trex, CEBin-E and CEBin on various poolsize.

Model	Poolsize			
	100	10,000	1,000,000	4,000,000
GNN	0.004	0.004	0.012	0.037
Trex	0.018	0.018	0.050	0.182
CEBin-E	0.807	0.814	0.887	1.034
CEBin	2.717	2.772	2.898	3.105

11.0, while the Cisco Dataset comprises six architectures and eight compilers. Because CEBin-BinaryCorp is trained based on cross-optimization tasks, we measure its performance in terms of cross-optimization BCSD on the Trex Dataset for different architectures, alongside CEBin-Cisco. Table 4 indicates that CEBin-BinaryCorp outperforms CEBin-Cisco in the XO task on x64 and also other architectures, with an average recall@1 increase of 0.028.

The results show that a IL-based model fine-tuned with the XO dataset can effectively transfer across different architectures. It is worth noting that CEBin-BinaryCorp performs better than CEBin-Cisco, which we speculate may be attributed to the BinaryCorp dataset containing 1,612 projects and offering more diverse binary functions. The Cisco Dataset consists of only seven projects, meaning that it may not be sufficiently diversified from a function semantics perspective, despite encompassing many different architectures and compiler variations. The use of a pre-trained model with IL has to some extent mitigated the effects of different architectures. Compared to the Cisco datasets, the BinaryCorp dataset is larger and exhibits better diversity, which explains why CEBin-BinaryCorp outperforms CEBin-Cisco. We believe the results prove that the IL-based pre-trained CEBin model is an effective solution for addressing cross-architecture binary code representation and scaling the size and diversity of the binary code dataset is crucial for training an effective model.

5.5 Inference Cost (RQ5)

We evaluate the inference cost of CEBin against varying poolsizes and compare it to baselines. For embedding-based approaches, we pre-compute the corresponding vectors for the function pool. When assessing a new query function, we measure the cost from acquiring the embedding vector with the embedding-based model to comparing it with each function in the pool to obtain the results. The experimental results in Table 5 reveal the time each method requires for handling different poolsizes.

Though CEBin's inference cost is relatively higher than the baseline, it does not increase rapidly as the poolsize expands. Even with a poolsize of 4 million, CEBin can process it in only 3.1 seconds, thereby demonstrating scalability for real-world software supply chain 1-day vulnerability discovery tasks.

6 DISCUSSION

In adapting momentum contrast learning for BCSD, we encountered unique challenges. The diversity in assembly code across ISAs necessitated normalization via MLIL, which might affect detection granularity. Contrary to original findings of MoCo [20], our model showed heightened sensitivity to hyperparameters, leading to extensive tuning efforts. Additionally, maintaining dual encoders, contrary to settings from MoCo, was essential for optimal performance in BCSD. While our study presents significant advancements with CEBin, we also recognize certain limitations.

Fine-grained Function Similarity Detection. Our work only focuses on coarse-grained function level similarity detection. We cannot solve the 1-day vulnerability detection problem well as the general function-level BCSD cannot distinguish whether a function is patched or not. Future works can combine BCSD solutions with fine-grained techniques such as directed fuzzing or security patch identification techniques to better detect 1-day vulnerabilities.

False Negatives in Datasets. Currently, all datasets suffer from the false negative problems, such as the existence of cloned functions across different projects. Even though the proportion is low, it will bring negative impacts on the model performance and the final results. Future works can explore constructing a better dataset, such as fine-grained deduplication.

7 CONCLUSION

In this paper, we propose CEBin, a novel cost-effective binary code similarity detection framework that bridges the gap between embedding-based and comparison-based approaches. CEBin employs a refined embedding-based approach to extract robust features from code, efficiently narrowing down the range of similar code. Following that, it uses a comparison-based approach to implement pairwise comparisons and capture complex relationships, significantly improving similarity detection accuracy. Through comprehensive experiments on three datasets, we demonstrate that CEBin outperforms SOTA baselines in various settings. We also showcase that CEBin successfully handles the challenge of large-scale function search in binary code similarity detection, making it an effective tool for real-world applications, such as detecting 1-day vulnerabilities in large-scale software ecosystems.

ACKNOWLEDGEMENTS

We would like to sincerely thank all the reviewers for their insightful feedback that greatly helped us to improve this paper. Additionally, special thanks are extended to Jingwei Yi and Bolun Zhang for their invaluable comments and assistance with our experiments. This work was supported in part by the National Key Research and Development Program of China (2021YFB2701000) and National Natural Science Foundation of China (61972224).

REFERENCES

[1] Sunwoo Ahn, Seonggwan Ahn, Hyungjoon Koo, and Yunheung Paek. 2022. Practical Binary Code Similarity Detection with BERT-Based Transferable Similarity Learning. In *Proceedings of the 38th Annual Computer Security Applications Conference* (Austin, TX, USA) *(ACSAC '22)*. Association for Computing Machinery, New York, NY, USA, 361–374. https://doi.org/10.1145/3564625.3567975

[2] Sunwoo Ahn, Seonggwan Ahn, Hyungjoon Koo, and Yunheung Paek. 2022. Practical binary code similarity detection with bert-based transferable similarity learning. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 361–374.

[3] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2018. FOSSIL: A Resilient and Efficient System for Identifying FOSS Functions in Malware Binaries. *ACM Transactions on Privacy and Security* (Feb 2018), 1–34. https://doi.org/10.1145/3175492

[4] Silvio Cesare, Yang Xiang, and Wanlei Zhou. 2013. Control flow-based malware variantdetection. *IEEE Transactions on Dependable and Secure Computing* 11, 4 (2013), 307–317.

[5] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *ACM SIGPLAN Notices* 51, 6 (2016), 266–280.

[6] 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.

[7] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices* 53, 2 (2018), 392–404.

[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] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium*.

[11] Thomas Dullien and Rolf Rolles. 2005. Graph-based comparison of executable objects (english version). *Sstic* 5, 1 (2005), 3.

[12] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code.. In *NDSS*, Vol. 52. 58–79.

[13] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. 2014. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*. IEEE, 78–87.

[14] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. 2017. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 346–359.

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

[16] Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*. Springer, 238–255.

[17] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 896–899.

[18] Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. 2023. How Far Have We Gone in Vulnerability Detection Using Large Language Models. arXiv:2311.12420 [cs.AI]

[19] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. [n. d.]. Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection.

[20] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 9729–9738.

[21] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. 2013. Mutantx-s: Scalable malware clustering based on static features. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*. 187–198.

[22] 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.

[23] 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.

[24] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *2012 IEEE Symposium on Security and Privacy*. 48–62. https://doi.org/10.1109/SP.2012.13

[25] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. arXiv:2401.11161 [cs.SE]

[26] Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2023. Nova$^+$: Generative Language Models for Binaries. arXiv:2311.13721 [cs.SE]

[27] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[28] Ulf Kargén and Nahid Shahmehri. 2017. Towards robust instruction-level trace alignment of binary code. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 342–352.

[29] TaeGuen Kim, Yeo Reum Lee, BooJoong Kang, and Eul Gyu Im. 2019. Binary executable file similarity calculation using function matching. *The Journal of Supercomputing* 75, 2 (2019), 607–622.

[30] Taku Kudo. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. *arXiv preprint arXiv:1804.10959* (2018).

[31] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[32] Siyuan Li, Yongpan Wang, Chaopeng Dong, Shouguo Yang, Hong Li, Hao Sun, Zhe Lang, Zuxin Chen, Weijie Wang, Hongsong Zhu, et al. 2023. Libam: An area matching framework for detecting third-party libraries in binaries. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–35.

[33] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 3236–3251. https://doi.org/10.1145/3460120.3484587

[34] 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.

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

[36] 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.

[37] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177.

[38] 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 *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/vulhawk-cross-architecture-vulnerability-detection-with-entropy-based-binary-code-search/

[39] 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)*. USENIX Association, Boston, MA, 2099–2116. https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli

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

[41] Luca Massarelli, Giuseppe A Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*.

[42] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. 2017. BinSign: Fingerprinting binary functions to support automated analysis of code executables. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 341–355.

[43] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library.* Curran Associates Inc., Red Hook, NY, USA.

[45] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680* (2020).

[46] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. Learning Approximate Execution Semantics From Traces for Binary Function Similarity. *IEEE Transactions on Software Engineering* (2022).

[47] Jannik Pewny, 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.

[48] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*. 406–415.

[49] Abdullah Qasem, Mourad Debbabi, Bernard Lebel, and Marthe Kassouf. 2023. Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures. In *Proceedings of the 2023 acm asia conference on computer and communications security*. 443–456.

[50] Kimberly Redmond, Lannan Luo, and Qiang Zeng. 2018. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. *arXiv preprint arXiv:1812.09652* (2018).

[51] 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* (Toronto, Canada) *(PLAS '18)*. Association for Computing Machinery, New York, NY, USA, 42–47. https://doi.org/10.1145/3264820.3264821

[52] 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.

[53] Paria Shirani, Leo Collard, Basile L Agba, Bernard Lebel, Mourad Debbabi, Lingyu Wang, and Aiman Hanna. 2018. Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 114–138.

[54] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. 2020. LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 104–115. https://doi.org/10.1109/SANER48275.2020.9054845

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[56] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024. CLAP: Learning Transferable Binary Code Representations with Natural Language Supervision. arXiv:2402.16928 [cs.SE]

[57] Huaijin Wang, Pingchuan Ma, Yuanyuan Yuan, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2023. Enhancing DNN-Based Binary Code Function Search With Low-Cost Equivalence Checking. *IEEE Transactions on Software Engineering* 49, 1 (2023), 226–250. https://doi.org/10.1109/TSE.2022.3149240

[58] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: Jump-Aware Transformer for Binary Code Similarity. *arXiv preprint arXiv:2205.12713* (2022).

[59] Hao Wang, Jia Zhang, Yingce Xia, Jiang Bian, Chao Zhang, and Tie-Yan Liu. 2020. COSEA: Convolutional Code Search with Layer-wise Attention. arXiv:2010.09520 [cs.SE]

[60] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1106–1118.

[61] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1106–1118. https://doi.org/10.1145/3597926.3598121

[62] 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.

[63] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 462–472.

[64] Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. 2021. Codee: A Tensor Embedding Scheme for Binary Code Search. *IEEE Transactions on Software Engineering* (2021).

[65] Shouguo Yang, Chaopeng Dong, Yang Xiao, Yiran Cheng, Zhiqiang Shi, Zhi Li, and Limin Sun. 2023. Asteria-Pro: Enhancing Deep Learning-based Binary Code Similarity Detection by Incorporating Domain Knowledge. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–40.

[66] 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.

[67] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems* 33 (2020), 3872–3883.

[68] Wenyu Zhu, Hao Wang, Yuchen Zhou, Jiaming Wang, Zihan Sha, Zeyu Gao, and Chao Zhang. 2023. kTrans: Knowledge-Aware Transformer for Binary Code Embedding. arXiv:2308.12659 [cs.SE]

[69] Xiaoya Zhu, Junfeng Wang, Zhiyang Fang, Xiaokang Yin, and Shengli Liu. 2023. BBDetector: A Precise and Scalable Third-Party Library Detection in Binary Executables with Fine-Grained Function-Level Features. *Applied Sciences* 13, 1 (2023). https://doi.org/10.3390/app13010413

[70] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706* (2018).

[71] zynamics. 2018. BinDiff. "https://www.zynamics.com/bindiff.html".

Received 16-DEC-2023; accepted 2024-03-02