BinAug: Enhancing Binary Similarity Analysis with Low-Cost Input Repairing

Wai Kin Wong
The Hong Kong University of Science and Technology
Hong Kong SAR
wkwongal@cse.ust.hk

Huaijin Wang*
The Hong Kong University of Science and Technology
Hong Kong SAR
hwangdz@cse.ust.hk

Zongjie Li
The Hong Kong University of Science and Technology
Hong Kong SAR
zligo@cse.ust.hk

Shuai Wang*
The Hong Kong University of Science and Technology
Hong Kong SAR
shuaiw@cse.ust.hk

ABSTRACT

Binary code similarity analysis (BCSA) is a fundamental building block for various software security, reverse engineering, and re-engineering applications. Existing research has applied deep neural networks (DNNs) to measure the similarity between binary code, following the major breakthrough of DNNs in processing media data like images. Despite the encouraging results of DNN-based BCSA, it is however not widely deployed in the industry due to the instability and the black-box nature of DNNs.

In this work, we first launch an extensive study over the state-of-the-art (SoTA) BCSA tools, and investigate their erroneous predictions from both quantitative and qualitative perspectives. Then, we accordingly design a low-cost and generic framework, namely BinAug, to improve the accuracy of BCSA tools by repairing their input binary codes. Aligned with the typical workflow of DNN-based BCSA, BinAug obtains the sorted top-K results of code similarity, and then re-ranks the results using a set of carefully-designed transformations. BinAug supports both black- and white-box settings, depending on the accessibility of the DNN model internals. Our experimental results show that BinAug can constantly improve performance of the SoTA BCSA tools by an average of 2.38pt and 6.46pt in the black- and the white-box settings. Moreover, with BinAug, we enhance the F1 score of binary software component analysis, an important downstream application of BCSA, by an average of 5.43pt and 7.45pt in the black- and the white-box settings.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; • **Theory of computation** → *Machine learning theory.*

KEYWORDS

Binary analysis, DNNs, Input repairing

ACM Reference Format:
Wai Kin Wong, Huaijin Wang, Zongjie Li, and Shuai Wang. 2024. BinAug: Enhancing Binary Similarity Analysis with Low-Cost Input Repairing. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623328

*Corresponding author.

1 INTRODUCTION

Binary code similarity analysis (BCSA) compares two or more pieces of binary code and decides their similarity. As a fundamental analysis capability, it has been actively employed in a wide spectrum of real-world software (re-)engineering and security applications, including malware clustering [13, 41], malware detection [21, 32], vulnerability mining [28, 54, 62, 96, 106], license violation detection [38], code plagiarism detection [58], and software composition analysis [74, 98].

Establishing well-performing BCSA frameworks is challenging, as binary code is generally more obscure than source code. Various analysis-friendly features, e.g., variable names and symbols, no longer exist in the binary code after compilation. Even worse, modern compilers often employ numerous optimization passes during the compilation [1], "diversifying" possible binary codes originated from the same source code and making it even harder to understand and match binary code. For example, as an essential optimization pass, function inlining replaces a function call site with the body of its caller function. Such an optimization pass helps to reduce the overhead of procedure call, at the cost of complicating the derived binary code components. As noted in existing work [97], the detection rates for malware can significantly decrease when malware samples are compiled with non-default optimization settings.

To tackle BCSA, heuristic-based, graph isomorphism approach has been widely adopted in the industry [10]. Techniques based on dynamic or symbolic analysis [24, 25, 34] have also been applied. Holistically speaking, these techniques suffer from either low accuracy (e.g., using primarily syntactical features for comparison) or high cost (e.g., requiring launching symbolic execution of the assembly code). Recently, following the major success of deep learning and particularly representation learning, well-trained deep neural networks (DNNs) are employed in BCSA, and they have shown promising results by automatically discovering the representations needed for accurate and scalable BCSA from raw binary code or assembly instructions [14, 28, 29, 35, 37, 50, 62, 82, 96, 99, 100].

Despite the prosperous development, such DNN-based methods are imperfect, as they may face the inherent robustness issue. In particular, a small perturbation in the input binary code may cause a notable change in the latent representation learned by DNNs. Also, performance of DNN-based BSCA is often influenced by the DNN model architectures. For example, graph neural networks

(GNNs), a popular DNN paradigm widely-used in software engineering tasks [49, 110], are frequently affected due to the problem of over-smoothing (see details in Sec. 3), reducing the expressiveness and utility of the computed node embeddings. This happens because the so-called message passing phase in training a GNN model aggregates neighboring nodes and causes their representations to converge to similar vectors. Our preliminary observation (see details in Sec. 3) shows that this impedes GNN-based BCSA from precisely matching similar binary codes. Moreover, the out-of-distribution (OOD) issue can also adversely affect the performance of BCSA. The performance of DNNs heavily relies on the training data [91]. However, binary code with largely different representations may be generated from the same source code due to the varying combinations of possible compilers, optimization passes, and compilation toolchain versions. The significantly diversified binary codes result in the inherent OOD issue, which is hard to address by existing DNN-based BCSA methods.

To tackle the challenges in BCSA mentioned above, the intuition is to detect failure-inducing DNN inputs with various DNN testing methods proposed in the SE community [60, 67, 93]. Then, with the located failure-inducing inputs, one may explore offline adversarial re-training or data augmentation techniques proposed by the AI community so that the employed BCSA DNN models can generalize better. However, we argue that offline re-training is time-consuming and often requires expensive hardware. The model providers may thus run into an endless loop: they have to keep adding the failure-inducing inputs, but the cumulative cost may be unaffordable. Moreover, the enhanced DNN models are still constrained by the additional data, whereas real-world inputs are unpredictable in the wild. While in information retrieval, a similar problem has been studied [69, 72, 109], our tentative study shows that tactics used for enhancing DNN-based image re-trivial results will induce false positives towards BCSA, as they are designed for different objectives.

In this research, we aim to enhance the performance of BCSA without conducting expensive offline re-training/tuning. Instead, we propose a lightweight framework, BinAug, to automatically repair real-world BCSA inputs so that the employed DNN models can perform better. This is challenging, as without ground truth, it is unclear if an input, often in the form of assembly functions, is failure-inducing or not. To this end, we first manually examine a substantial number of failure-inducing inputs yielded by the state-of-the-art (SoTA) models, and summarize a set of failure patterns. Based on the empirical observations, we then introduce BinAug, a framework to automatically repair binary code inputs into more analysis-friendly ones with a set of lightweight transformations. We consider both black-box and white-box scenarios, where for the white-box cases, we assume the availability of DNN model internals and intermediate representations to guide the repairing. Our evaluation shows that BinAug can effectively correct the potential failure-inducing inputs and offers a general improvement, by an average of 2.38pt and 6.46pt in the black-box and white-box scenarios, across eight SoTA BCSA models. Moreover, BinAug boosts real-world BCSA applications, with an average of 1.72pt and 5.62pt for black-box and white-box settings of function clone search. BinAug also improves the F1 scores of binary software component analysis, an important downstream application of BCSA, by an average of 5.43pt and 7.45pt in the black-box and white-box settings, respectively. In sum, we make the following contributions:

- We advocate a novel viewpoint of enhancing DNN-based BCSA through input repairing, which is lightweight and does not require offline re-training, fine-tuning, or data augmentation.
- With manually summarized failure patterns, we propose an input repairing framework, BinAug, which features a set of lightweight transformations and DNN-aware procedures that are applicable to both black-box and white-box scenarios.
- BinAug consistently improves the performance of eight SoTA BCSA models with modest overhead (less than 0.07 seconds per input). It also notably boosts critical downstream applications. Our artifact (e.g., the codebase of BinAug) is at [3].

2 BACKGROUND

DNN-Based BCSA. In this research, we take a common BCSA setting [82], such that the comparison unit is *binary code functions*. That is, given a piece of binary code composing multiple functions, BCSA performs pair-wise comparisons to iteratively compare each binary function in the input binary against a database of binary functions. This way, BCSA can be formally defined as an information retrieval system for binary functions, such that given a query function f_q and a database of functions P, BCSA retrieves the top-K functions $\{f_1, f_2, ..., f_k | f_i \in P\}$ ranked by the similarity scores.

The major challenge of BCSA is introduced by the compilation process. It produces unreadable machine code, discards types, and variable names, and optimizes the structures like control flow graphs (CFGs). Thus, a line of BCSA research extracts semantic features for accurate similarity comparison. Conventional works like [19, 31, 58, 80, 86] use program static/dynamic analysis techniques to extract semantic features like input-output relations or symbolic expressions. These methods, despite their accuracy, generally suffer from high overhead and are not widely used in practice.

Research success in deep learning, particularly in representation learning [15], has inspired a line of BCSA works with largely enhanced performance [28, 29, 35, 50, 62, 82, 96, 99, 100]. In short, deep-learning techniques facilitate transforming high-dimensional data into lower-dimensional representations, which are often referred to as "embedding vectors" in a latent space. This way, matching complex high-dimensional data can be reduced to computing their "distance" in the latent space.

The pipeline of DNN-based BCSA is largely inspired by the contemporary research in natural language processing (NLP). High-levelly speaking, a query binary function f_q is viewed as a sequence of instructions, where BCSA treats instructions and instruction sequences as words and sentences for representation learning and matching [28, 35, 62, 82, 96, 99, 100]. Fig. 1 presents a common pipeline. ① Given an executable, BCSA first disassembles the input into assembly instructions and further recovers the function-level CFG. ② Then, the instruction sequence of each basic block in the query function is treated as a sentence. Instruction embeddings are produced by word embedding techniques like Word2vec [64], and block embeddings are produced with pooling layers or recurrent neural networks (RNN) like HBMP [73]. ③ Since each binary function is structured, recent works [96, 99, 100] frequently use graph

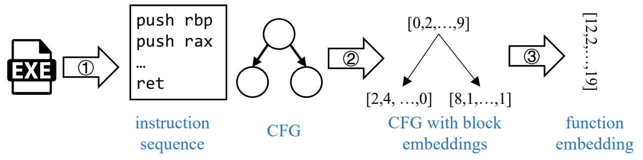


Figure 1: The common pipeline of DNN-based BCSA.

neural networks (GNN) to compute the query function's embedding vector directly over its CFG. In comparison to earlier works (e.g., SAFE [62]) which treat a binary function merely as an instruction sequence (i.e., a whole block) and ignore its CFG structure, GNN-enhanced works often achieve notably better accuracy. Last, to decide the similarity between the query function and a function in the database, algorithms like cosine distance are used over their embedding vectors (omitted in Fig. 1).

Performance Degradation of DNN-based Applications. While no existing works have systematically studied and repaired the performance degradation of DNN-based BCSA, we introduce several key issues that generally impede DNN-based applications.

Distribution shifting. The performance of DNN-based applications likely drops when inputs are out of the model training set's distribution. This is often referred to as out-of-distribution (OOD). Note that it is a common challenge when integrating DNN in real-world applications. In particular, our preliminary observation has shown that in the context of DNN-based BCSA, inputs can be built with different compilers, linkers, and optimization settings, resulting in detection failures. To cope with this challenge, one may have to frequently re-train the employed DNN models with new input samples. However, such an approach requires a huge effort, and real-world inputs are generally unpredictable [92].

GNN Specific Issues. GNN exhibits highly encouraging performance to learn structural data [57, 61]. GNN is extensively employed in program analysis applications, including BCSA. Nevertheless, GNNs are hard to train and prone to under-reaching, over-smoothing, or over-squashing problems. *Under-reaching* is a typical GNN hurdle induced by the number of layers K in a GNN network. This hyper-parameter K has limited the number of hops reachable from any node in the graph through message passing. Thus, any nodes in the graph cannot be aware of nodes that are farther away than K hops. While increasing the number of propagation layers can resolve the under-reaching problem, it may cause the *over-smoothing* problem, which arises due to how GNNs are trained [11]. During training, GNN updates node representations by aggregating themselves and their neighbors' representations through message passing. When the number of iterations for message passing is sufficiently large, the neighbor aggregation strategy likely smooths node representations over a graph, causing them to converge to similar vectors. Moreover, *Over-squashing* is another inherent problem for GNNs. It happens when the input exhibit specific topological properties [11, 76]. As a result, the model performs poorly when handling such kind of inputs, which usually involves long-range interaction.

Input Repairing. With the usage of DNNs in various fields, there is a growing interest in detecting and repairing failure-inducing inputs. For example, with the help of triplet loss [71], Xiao et al. [92] propose to automatically recognize unexpected image inputs, which are mapped to similar yet more regulated samples in the training data. Besides repairing images, DietCodeBert [107] is designed over source code inputs, and it uses auxiliary models to identify source code pieces with critical semantics. Li et al. [53] leverage a black-box majority voting phase to repair outputs of code completion models. Compared with offline model re-training or fine-tuning, online detection and repairing of failure-inducing inputs offer several advantages. First, it reduces the burden and cost of re-training/fine-tuning when the underlying DNN models are already in use. Moreover, model re-training/fine-tuning is infeasible for various scenarios; for instance, the model or original training data is not accessible, or no proper devices (e.g., GPUs) are available for re-training. In contrast, input repairing can be performed on the fly, and it does not require the model or training data. Moreover, even if the model is accessible and re-trainable, we anticipate that input repairing can offer orthogonal and likely synergistic improvement. This work, for the first time, presents an effective and tailored input repairing framework for BCSA and its enabled downstream applications. See further comparison with relevant works in Sec. 4.

3 PRELIMINARY STUDY

To explore the failure patterns of de facto DNN-based BCSA, we manually examine the matching results of three SoTA BCSA models and analyze all the failure cases.[1] We select two evaluated and well-performing BCSA tools from a recent survey [61]: SAFE [62] and GNN-Talos [51, 61]. [2] As noted in Sec. 2, GNN-Talos leverages GNN to take CFG-level embedding into account, whereas the earlier work, SAFE, treats a binary function as a sentence. We also evaluated a commercial BCSA tool (name blinded) using GNN.

We evaluate these three BCSA tools using FFmpeg, a popular open-source multimedia framework. We perform cross-optimization BCSA, where the 23,705 binary functions in FFmpeg are split into a query set (9,397 functions) and a target set (14,308 functions) stored in the database. In short, while each tool exhibits reasonable BCSA performance in this setting, they make mistakes from time to time. At this step, we randomly pick 200 failure samples made by each BCSA tool and discuss their failure patterns as follows.

3.1 Failure Patterns of SAFE

Technically, SAFE is an RNN-based BCSA model which treats assembly functions as sentences and uses a self-attentive sentence encoder to learn the function representation. After manually reviewing the failure instances from SAFE, we discover that *attention overfitting* is the major cause of failures for the SAFE model. We notice that the model frequently focuses on the function prologues and also stack-relative instructions. In fact, about 50% of the failure instances gain the most significant attention at their first basic blocks, incorrectly resulting in high similarities between query functions and the functions with similar prologues in the database. In general, we observe that most failure instances are due to incorrect attention over various mundane instructions (e.g., stack operations) in an

[1]A matching failure occurs when the function with the highest similarity score (top-1 matching) is wrong.
[2]We denote GNN in Marcelli et al. as GNN-Talos.

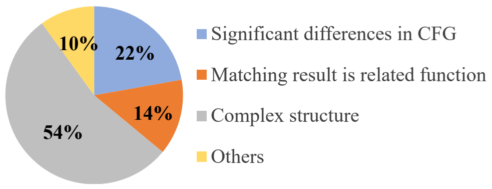


Figure 2: Failure pattern distribution.

assembly function. To clarify, while detecting similar functions on the basis of prologues is a common heuristic for writing function signatures and Yara rules [9, 16], these rules usually employs other global features like strings to reduce false positive matches. However, the attention mechanism of SAFE falsely and merely focuses on those mundane routines like the function prologues.

3.2 Failure Patterns of GNN-Based Techniques

By manually analyzing the failure instances of two GNN-based tools (GNN-Talos and the commercial tool), we identify three common failure patterns as following. Due to the limited space, we leave samples of each pattern on our webpage [7] for reference, and we report the distribution of failure patterns in Fig. 2.

① Significant differences in CFGs: If the graph structures between the query function and the ground truth are vastly different, GNN often prioritizes the graph structure over the node attributes, even if they are similar at the instruction level (i.e., node attributes), resulting in failures. We categorize an assembly function and its ground truth as pattern ① if one's CFG has 1.5 times more nodes than the other. As expected, compiler optimizations like function inlining are major root causes of such failures.

② Functions with similar CFGs: Some functions with similar CFGs and functionality gain greater scores than the correct match occasionally, such as those utility functions (e.g., `cbs_h265_write_sei_buffering_period` and `cbs_h265_write_sei_pic_timing`) with similar functionality and only slight difference. As a heuristic, we identify those functions by measuring if their names' common subsequence is over 50%.

③ Complex structure: The majority of failure cases fall in this category. In short, functions may have complex CFGs and long-range distances between nodes on the CFG. Such complex structures would presumably result in the inherent problems of GNNs, including under-reaching, over-squashing, and over-smoothing (as introduced in Sec. 3). For instance, basic blocks containing the return instructions often have extremely long distances from the entry block of the function, resulting in the under-reaching and over-squashing issues.

④ Others: Some failure patterns are hard to generalize. For example, the CFG of a query function is relatively simple yet matched to a function with a vastly different CFG. This is hard to interpret on our end, given that explaining DNN model predictions is still an open problem. We refer to these types of issues as Others.

Explanation. As shown in Fig. 2, ③ and ① are the two major failure patterns for GNN-based BCSA models. To efficiently generate embedding vectors, GNN-based approaches often limit the steps for message passing. Therefore, the node information cannot propagate thoroughly when a function's CFG is complex, resulting in low-quality embedding due to the under-reaching problem mentioned in Sec. 2. By analyzing the topology, we observe that cases belonging to pattern ③ usually have long distance between basic blocks, resulting in under-reaching and over-squashing problems. Some GNN-based BCSA models increase the number of layers for message passing to avoid under-reaching, but they in turn likely suffer from the over-smoothing issue [11].

4 FRAMEWORK DESIGN

Following the empirical observations in Sec. 3, we design BINAUG, a BCSA input repairing framework that improves the accuracy of DNN-based BCSA solutions. Given a query binary function, BINAUG aims to provide a lightweight approach to re-ranking the top-K results returned from a DNN-based BCSA tool.[3]

Limitations of Existing Works. As already reviewed in Sec. 2, recent works, InputReflector and CCTest, launch input repairing to improve the performance of DNN-based applications. However, InputReflector is specifically designed for image classification, while BCSA mainly relies on the expressiveness of code embeddings yielded by DNN models. CCTest improves the results of the code completion system through source code transformations, which are unsuitable for BCSA since CCTest's transformation candidates (e.g., variable names) are unavailable in binary code. Also, result re-ranking methods are often tailored for information retrieval systems and are not applicable to the BCSA scenario [20, 66].

Framework Overview. BINAUG subsumes two variants of input repairing methods, based on whether the BCSA embedding model is accessible or not. The white-box setting is applicable when the embedding model internal, including its structure and parameters, is accessible. In contrast, users can merely access the embedding vectors in the black-box setting. For both white-box and black-box settings, the query binary and the database being queried (i.e., P of Sec. 2 but not the training dataset of the BCSA model) are always accessible. This setting is reasonable and aligned with typical usage scenarios where users upload a query binary through API to a remote BCSA service and receive the top-K matched functions.

Following the standard practice, the BCSA first provides the top-K most similar functions in the function pool P for a query binary function f_q. With the initial matching results, BINAUG re-ranks the results by calculating $ReRank(f_q, p_i)$ between the query function and the i-th ranked function as follows:

$$ReRank(f_q, p_i) = (1 - \alpha)Sim(f_q, p_i) + \alpha Sim_{aug}(f_q, p_i) \quad (1)$$

where $ReRank(f_q, p_i)$ (Eq. 1) is defined as the linear combination between the original similarity score ($Sim(f_q, p_i)$) and the augmented similarity score $Sim_{aug}(f_q, p_i)$. We elaborate on computing the augmented similarity scores for both white-box and black-box settings in Sec. 4.1 and Sec. 4.2, respectively. Our exploration (see details in Sec. 6.6) across multiple SoTA BCSA models shows that the optimal values for α ranges from 0.4 to 0.6. We thus recommend setting $\alpha = 0.5$ for general use.

[3]Our evaluation shows that for SoTA BCSA models, configuring $K = 10$ usually offers a good balance between enhanced accuracy and incurred cost; see details in Sec. 6.6.

4.1 White-box Input Repairing

Attention-Based BCSA Models. When BCSA models employ the standard attention mechanism [77], we classify it as an attention-based model. As aforementioned, modern BCSA solutions often treat binary code (e.g., assembly basic blocks or instruction sequences) as natural language sentences and apply the attention mechanism to learn the semantic similarity. In other words, attention-based models are common in contemporary BCSA works, e.g., SAFE which is evaluated in this work.

As uncovered in Sec. 3.1, models with attention layers often falsely focus on a function's prologue and other stack-related instructions. However, those instructions are generally mundane, and they do not necessarily encode meaningful semantics of a function. This results in the *overfitting* problem of attention-based models.

To alleviate the overfitting issue, we propose a novel mutation to generate new embeddings for a pair of functions under comparison. Given two assembly functions f_1 and f_2 and an attention-based BCSA model, we first apply the longest common subsequence (LCS) to obtain an instruction sequence $I = [i_1, i_2, \ldots, i_n]$ that is commonly focused (i.e., each instruction is associated with a high attention score; we set the threshold as 0.1 as most attention scores are in the range of 10^{-4} to 10^{-12}) in f_1 and f_2 by the attention model. Next, we mutate the original instruction sequence by masking an instruction $i_j \in I$ and collect the embedding vector over the mutated input from the BCSA model. To "mask" i_j, since there is no token for masking of the original model, we replace its instruction with the `NOP` instruction which does nothing during execution. After iteratively mutating every $i \in I$, we collect embeddings over all mutations. Then, we average the mutated embeddings of f_1 (as well as that of f_2).[4] We then use the new embeddings for re-ranking.

Instead of iteratively masking every $i \in I$, one may wonder about the feasibility of masking all instructions of I at a time. Although technically feasible, masking all instructions of I results in meaningless instruction sequences and low-quality embeddings, while masking one instruction each time introduces small and acceptable noises. Intuitively, this method generalizes the inputs and thus mitigates overfitting. And since every attended instruction is removed once for the following comparison, we expect to alleviate its influence and force the attention-based BCSA model to focus on uncommon instructions in f_1 and f_2 when generating embeddings.

GNN-Based Models. Instead simply treating the assembly instructions as a sequence of tokens, recent works have shown that the structure information of assembly programs can be largely beneficial for more accurate BCSA. Thus, the mainstream of recent BCSA tools leverages GNNs to capture program structural-level information. We denote them as GNN-based BCSA models. In particular, all our evaluated SoTA BCSA tools, except SAFE, employ GNNs.

We observe that control-flow graph (CFG) is the most frequently-used program structure [37] by GNNs. As shown in Fig. 1, after generating the node feature vectors with NLP techniques like Word2vec [64], SoTA BCSA tools [28, 29, 50, 96, 100] employ GNNs to produce function-level embeddings. Inspired by solutions to the classic network alignment problem (also known as "graph matching problem" [47]), we propose an GNN-specific input repairing scheme. Overall, network alignment aims to match corresponding nodes between graphs and has been widely-applied to social network analysis [44, 56] and protein analysis [45]. Generally, a network can be aligned on the basis of graph topology or node features, or both features together. We focus on node features, since over-smoothing is a problem that generally affects the expressiveness of node embeddings. Moreover, node alignment shall avoid triggering the over-squashing and under-reaching problems, since this procedure does not require the message passing phase. We however assume that GNN has correctly captured the topological information of the graph in most scenarios. This is because de facto GNN architectures are at most powerful as the Weisfeiler-Lehman algorithm [87], which is a computationally efficient heuristic to perform graph isomorphism tests [12], as shown by Xu et al [94]. Most SoTA GNN designs go beyond this baseline [88] and well captures the topology of the graph.

[4]To clarify, "averaging" is a basic and useful trick [2, 40] to compute the arithmetic average of multiple embedding vectors in natural language processing.

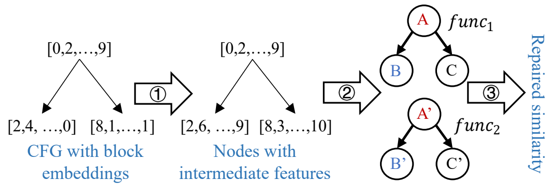


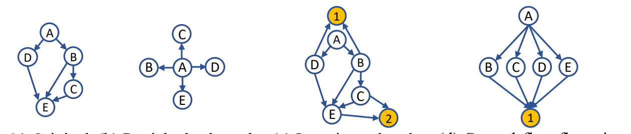
Figure 3: Illustrating GNN-based white-box repairing.

To ease presentation, we name our white-box, GNN-specific input repairing method as Hungarian algorithm (HA). Fig. 3 presents the pipeline of HA. After initializing basic blocks' embeddings (as shown in Fig. 1), we first perform ① an iteration of message passing. This way, we ensure that the intermediate feature of each node captures the information of its neighbors, whereas the limited iteration (only one iteration) eliminates over-smoothing. Next, we employ ② the standard Hungarian algorithm [5] to compute the optimal node alignment of two matched functions. Last, the repaired similarity (i.e., $Sim_{aug_{1,2}}$) is computed by ③ averaging the cosine similarities between the matched nodes.

4.2 Black-box Input Repairing

Motivation. In black-box scenarios, we can only access the top-K matched functions and the associated scores to quantify the similarity between each top-K function and the query function. Although white-box methods are not applicable, we aim to propose methods to repair the query input and overcome hurdles in BCSA. Recall in Sec. 3.2, failure patterns ③ and ① (the two major failure patterns) primarily root from the overly complex and vastly different inputs. The intuition is that if we "regulate" the CFG with a set of transformations, we may likely make visually distinct CFGs more similar. More importantly, if we can properly increase node connectivity, we enhance the message passing between nodes, improve the expressiveness of the node embeddings, and inherently alleviate the challenge imposed by complex and largely different CFGs.

Overview. We propose three transformations directly over the query functions. Inspired by graph rewiring [76] and graph augmentation methods [27, 108], we design all three transformations over the query function CFG. In general, these methods "regulate"



(a) Original (b) Partial wheel graph (c) Inserting subnodes (d) Control-flow flattening

Figure 4: Black-box graph transformations.

the structural representation of the query function and a database function under comparison, thus eliminating the influence of potential OOD, enhancing the node connectivity, and presumably easing the difficulty of comparing two functions originated from the same source code. The derived new similarity score, $Sim_{aug_{1,2}}$, will be used in Eq. 1 to update the final similarity score $ReRank_{1,2}$. We detail each method below, whose holistic view is shown in Fig. 4.

Partial Wheel Graph. We notice that a recent work [17] has measured the expressiveness of GNN using Dirichlet energy. In short, it is shown that transforming graphs into a supernode structure is equivalent to dropping edges, which can increase Dirichlet energy and alleviate the issues exposed in our empirical observation.

With this regard, we transform the original CFG to a new graph with a supernode. Specifically, we connect the first node (i.e., the entry of a function) with all other basic blocks, forming a special wheel-like structure. We present a case in Fig. 4(b) (transformed from Fig. 4(a)), where the first basic block is the supernode.

Inserting Subnodes. Recall in Sec. 2, message passing is the critical phase for encoding graphs. However, this process can be time-consuming; thus, existing BCSA tools limit the steps for message passing, resulting in the under-reaching problem when the CFG is complex. To alleviate this problem, we propose to insert subnodes (i.e., yellow nodes of Fig. 4(c)) between CFG nodes to enhance the reachability of message passing compared to the original CFG. Moreover, adding extra edges and supernodes are also profitable means for solving over-squashing [11, 36, 70].

Control-Flow Flattening. This transformation is essentially derived from software obfuscation methods [46], where the whole CFG is transformed into a huge "switch-like" structure. Moreover, it is noted that this transformation can be implemented as a combination of wheel transformation and subnode insertion. An example is presented in Fig. 4(d). We first apply the wheel transformation on the original CFG, then a new node that connects with all blocks except the entry block is inserted. For the sake of simplicity, we call the newly-inserted node as "dispatcher node."

5 EVALUATION SETUP

We implement BinAug using PyTorch (ver. 1.13.0) and extract features with IDA Pro [39], a commercial and popular reverse engineering tool [28, 29, 99, 100]. We use a Ryzen 3970X 32-core server with 256GB memory and a RTX 3090 GPU for experiments. The codebase of BinAug is available at [3] for research usage and extension.

Datasets. We reuse the two datasets of a previous BCSA study [61]. Dataset-1 is the training dataset that contains seven open-source projects. We followed its building method and compiled binaries with two compiler families (`gcc` and `clang`) and five optimization options (i.e., `O0-O3` and `Os`). Dataset-2 consists of ten open-source libraries and is also used by Trex [68]. Since Dataset-2 is compiled with `gcc` only, we additionally extend the dataset by compiling it with `clang`; thus, we can evaluate cross-compiler setting with Dataset-2. We also adopt a common approach [28, 99, 100] to removing simple functions (i.e., functions with less than five basic blocks). At last, there are 490K and 424K functions of two datasets, respectively. More details of the above two datasets are available in [61]. Besides, we use FFmpeg compiled by `gcc` with different optimizations as our validation dataset; this validation dataset consists of 23,705 functions (aligned with Sec. 3) to select the best model for testing. Overall, all binaries in our experiment are selected from real-world software in various domains, including mathematical computing, data compression, data encryption, database, and so on. We thus believe that our experiment is reasonably close to real-world BCSA usage scenarios. And we ensure the accuracy of our study and the credibility of our findings to a great extent.

Evaluation Targets. Marcelli et al. [61] study SoTA BCSA solutions and demonstrate that GNN-based BCSA outperforms other solutions in most situations. To evaluate the performance of BinAug, we also choose GNN-based works from [61], i.e., GNN-Talos [51], GNN-Structure2vec [23], and CodeCMR/BinaryAI [99, 100]. For GNN-Structure2vec, we consider two works: Gemini [96], a pioneer of applying GNN to BCSA with manually-engineered node features, and Structure2vec [63], producing feature vectors of basic blocks with assembly code embeddings. CodeCMR [100] is a commercial tool. We implement our own version with GGNN [52] and Set2Set [79] pooling layer based on its paper (denoted as Set2set). Additionally, we also include GAT [78], another SoTA GNN-based approach.

For approaches without a GNN model, we evaluate SAFE [62], a variant of the self-attentive sentence embedding model [55]. We also include Asm2Vec [28] which produces function-level embeddings using a PV-DM model [48], and takes into account the function CFG using a random walk-based method. Since Asm2Vec is not based on attention or GNN, we applied BinAug's black-box repairing strategies on Asm2Vec. We have also included an anonymous, DNN-based commercial BCSA solution available on the market to access BinAug's black-box methods in a real-world application scenario.

Two recent BCSA works [51, 82] are not evaluated due to their inefficiency. In particular, GMN, a variant of GNN-Talos proposed by Li et al. [51] does not generate function embeddings but computes the similarity of a pair of functions with a neural network; thus, the time cost for comparison becomes infeasible when the database to be searched is large. We also exclude jTrans [82] since it uses a large and expensive Bert model [26] to produce function embeddings.

Feature Extraction. As shown in the typical BCSA pipeline in Fig. 1, we need to perform both assembly instruction embedding and function-level embedding. Thus, besides deciding the target models for generating function-level embeddings, we also need to embed assembly code, except Gemini, which relies on manually-selected features. We use PalmTree [50], the SoTA instruction embedding framework, to embed assembly code for GNN-based models. Following the solution of CodeCMR [100], we train an end-to-end HBMP [73] model to generate basic block embeddings. Then, GNNs can be used to produce function-level embeddings.

For SAFE, our preliminary studies show that the *Precision*@1 performance and re-ranking performance of BinAug is similar when using either Word2vec or PalmTree as the instruction embeddings. Nevertheless, because of the performance overhead for re-encoding the instruction embeddings with PalmTree, we decide not to replace Word2vec with PalmTree. Asm2Vec uses a customized PV-DM model to learn function-level embeddings with self-supervised learning. It provides an API for use, and is treated as black-box in our evaluation.

Training Setting. For the training setup, given the popularity of the 64-bit x86 architecture, we use binary functions compiled with `gcc -O0` for 64-bit x86 as the anchor throughout training. Following a typical learning setting, we train the model until the loss stabilized, and we select the best model for evaluation based on results over our validation setting, i.e., performing search between FFmpeg compiled with `gcc -O0` and `gcc -O3`. For hyper-parameters used in model training, we follow the configurations reported in each tool's paper or settings in the released code repositories.

Evaluation Metrics. We use *Precision@K* as the evaluation metric. This is a standard metric for evaluating information retrieval tasks, where a query function f_q is correctly determined if the ground truth is within the top-K retrieved functions. Following the definition from Sec. 2, *Precision@K* is defined as follows:

$$Precision@K = \frac{\text{Number of query's correct match is within top-}K}{\text{Total number of issued queries}} \quad (2)$$

To deliver a comprehensive and challenging evaluation of BinAug, we set $K = 1$, the hardest setting for BCSA in our evaluations.

6 EVALUATION

Sec. 6.1 reports that BinAug can enhance the accuracy of BCSA tools regarding various challenging settings. Sec. 6.2 shows BinAug can improve critical real-world downstream applications of BCSA. Sec. 6.3 reports the overhead of BinAug, while Sec. 6.4 presents further study over the effectiveness of BinAug. Sec. 6.5 reports combining black- and white-box methods, and we discuss the effect of BinAug's hyper-parameters in Sec. 6.6.

6.1 BCSA Repair Performance

To evaluate the effectiveness of BinAug, we use three settings (XO, XO+XC, XO+XC+XB) from [61]. In particular, (1) XO denotes a cross-optimization setting where BCSA searches similar functions compiled with different compiler optimizations but the same compiler; (2) XO+XC denotes searching functions compiled with different optimizations and compilers; (3) XO+XC+XB is a more challenging, cross-bitness setting as it additionally compares 32-bit vs. 64-bit binaries. All three settings subsume typical scenarios of BCSA. In Table 1, we present the averaged *Precision*@1 of BCSA models, and their improvement after applying different repairing methods of BinAug, along with values of standard deviation. We now discuss the results in detail.

White-Box Methods. As shown in Table 1, our white-box input repairing strategies (i.e., LCS and Hungarian algorithm) successfully improve the *Precision*@1 for all BCSA works across all three settings. On average, the improvement of Hungarian algorithm (HA) is 6.35pt. For GNN-based BCSA models with learned instruction embeddings, we observe a trend where the improvement is inversely proportional to the original matching results. Nevertheless, BinAug can still bring considerable improvement for SoTA BCSA works as their accuracies are far from the perfect. We present an in-depth analysis in Sec. 6.4.

For SAFE, the attention-based model, BinAug enhances the *Precision*@1 by an average of 7.03pt. Unlike applying HA to GNN-based models, BinAug achieves constantly high improvement for different settings. Having that said, BinAug offers the least improvement with the XO+XC+XB setting. Recall that the attention-based repair relies on the LCS of two functions' instruction sequences to perform matching. However, there may be no common instructions shared by two functions with different bitnesses since they have different instruction sets and registers (e.g., there is no 64-bit register `rbp` in 32-bit binary code). See further discussion in Sec. 6.4.

Black-Box Methods. Although most black-box methods offer general improvements, the best repair method of a specific model may vary. For instance, inserting subnodes raises the *Precision*@1 of GNN-Talos model over 2pt across three comparison settings, but it is not the best black-box repair for Set2Set. This phenomenon is reasonable as a certain issue can have severer influence on a specific model than another model. We interpret that over-squashing has a considerable impact on GNN-Talos since shortening long distances by inserting subnodes (IS) can consistently improve its *Precision*@1, whereas over-smoothing is likely a significant issue of Set2Set as partial wheel graph (PWG) and control-flow flattening (CFF) bring higher improvement compared with inserting subnodes. Also, while CFF offers the best black-box repairing for Set2Set, it achieves negative improvement for the other models. We interpret this is due to over-squashing; see further discussions in Sec. 6.4.

Compared with the improvement between BinAug's white- and black-box methods, we can tell the white-box methods always bring better performance. The overall improvement of white-box methods (6.46pt) is nearly tripled than that of black-box methods (2.38pt). This is reasonable, given that white-box repairing offers more guided and DNN internal-aware enhancement. Having that said, our white-box repairing methods are practically applicable with moderate cost (see Sec. 6.3), and BinAug users (in this case, mainly the BCSA service providers) can employ them to improve their results without expensive retraining or fine-tuning. Also, in case only black-box methods are available, users, in this case mainly the BCSA API users, can still leverage the lightweight black-box methods to improve the results of the employed BCSA models.

Overall, the results suggest the effectiveness of BinAug on repairing BCSA results. For white-box scenarios, the HA manifests the best performance. This illustrates the importance of considering the intermediate node alignment, and also indicate the potential of BinAug in boosting other code similarity tasks on the basis of GNNs. As for black-box models, we observe that the best method frequently differs depending on the specific models. In domain-specific scenarios where only black-box methods are applicable, we recommend users to first launch validation experiments to decide the best repair method for their models.

Table 1: Averaged results of XO, XO+XC and XO+XC+XB. We mark best black-box repair method .

Model	Setting	XO		XO+XC		XO+XC+XB		**Total**
		Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.	Avg.
GNN-Talos	*Precision@1*	63.36%	9.53	74.87%	7.42	73.59%	9.6	70.61%
	HA-W	+6.84pt	2.09	+6.37pt	1.69	+6.49pt	2.35	+6.57pt
	PWG-B	+2.38pt	1.86	-0.76pt	1.72	-0.42pt	1.77	+0.40pt
	CFF-B	+1.37pt	1.85	-0.57pt	1.01	-0.31pt	1.14	+0.16pt
	IS-B	+2.33pt	1.6	+2.03pt	1.06	+2.32pt	1.06	+2.23pt
Structure2vec	*Precision@1*	60.49%	8.94	70.32%	7.03	67.3%	12.83	66.0%
	HA-W	+9.44pt	2.16	+10.53pt	2.14	+11.04pt	4.1	+10.34pt
	PWG-B	+3.1pt	1.26	+4.49pt	1.97	+1.42pt	0.52	+3.0pt
	CFF-B	-7.4pt	2.74	-15.59pt	4.04	-14.36pt	2.37	-12.45pt
	IS-B	+0.57pt	1.23	+0.4pt	1.23	+4.77pt	1.46	+1.91pt
Set2Set	*Precision@1*	72.39%	8.95	84.89%	4.72	83.49%	6.0	80.25%
	HA-W	+4.7pt	1.89	+2.74pt	1.33	+2.81pt	1.03	+3.41pt
	PWG-B	+1.03pt	1.2	+0.09pt	1.16	+0.39pt	0.73	+0.50pt
	CFF-B	+1.54pt	1.3	+1.04pt	0.89	+0.98pt	1.2	+1.19pt
	IS-B	+0.86pt	1.25	+0.75pt	0.68	+0.79pt	0.76	+0.80pt
GAT	*Precision@1*	59.26%	8.91	69.28%	7.43	67.03%	11.37	65.19%
	HA-W	+7.55pt	2.17	+8.74pt	1.51	+8.41pt	2.31	+8.23pt
	PWG-B	-0.01pt	0.03	0.0pt	0.0	0.0pt	0.0	-0.003pt
	CFF-B	-5.04pt	2.52	-13.38pt	3.57	-11.97pt	4.09	-10.13pt
	IS-B	+2.84pt	1.4	+3.92pt	1.08	+3.48pt	1.88	+3.41pt
Gemini	*Precision@1*	53.54%	11.18	46.07%	16.12	42.25%	15.20	47.29%
	HA-W	+2.93pt	2.58	+2.95pt	3.11	+3.77pt	5.60	+3.22pt
	PWG-B	+0.28pt	2.82	-1.46pt	3.08	-2.9pt	5.92	-1.36pt
	CFF-B	+0.56pt	2.67	+1.15pt	2.69	+0.66pt	4.74	+0.79pt
	IS-B	-5.19pt	4.09	-4.19pt	2.60	-4.02pt	6.00	-4.47pt
asm2vec	*Precision@1*	37.96%	7.86	26.53%	15.80	15.77%	14.37	26.75%
	PWG-B	+0.60pt	2.01	+0.24pt	2.41	+0.51pt	1.94	+0.45pt
	CFF-B	+0.84pt	1.86	+0.34pt	2.14	+0.20pt	1.96	+0.46pt
	IS-B	+1.80pt	2.11	+2.61pt	2.57	+1.06pt	2.77	+1.82pt
Commercial solution	*Precision@1*	70.30%	6.43	70.99%	6.55	62.03%	7.55	67.77%
	PWG-B	+1.46pt	1.49	+0.32pt	1.36	+0.94pt	1.58	+0.91pt
	CFF-B	+0.24pt	1.80	-0.28pt	1.82	+0.53pt	1.37	+0.16pt
	IS-B	+1.28pt	1.00	+2.48pt	1.09	+2.59pt	1.03	+2.12pt
SAFE	*Precision@1*	54.26%	8.97	37.85%	11.89	37.47%	12.01	43.19%
	LCS-W	+5.42pt	3.09	+13.18pt	3.24	+2.48pt	2.75	+7.03pt
Total	W	+6.15pt	-	+7.41pt	-	+5.83pt	-	+6.46pt
	HA-W	+6.29pt	-	+6.27pt	-	+6.50pt	-	+6.35pt
	LCS-W	+5.42pt	-	+13.18pt	-	+2.48pt	-	+7.03pt
	B	+1.95pt	-	+2.53pt	-	+2.66pt	-	+2.38pt
	PWG-B	+1.26pt	-	+0.42pt	-	-0.01pt	-	+0.56pt
	CFF-B	-1.13pt	-	-3.90pt	-	-3.47pt	-	-2.83pt
	IS-B	+0.64pt	-	+1.14pt	-	+1.57pt	-	+1.12pt

1. **Setting** abbreviations: Hungarian algorithm (HA), partial wheel graph (PWG), control-flow flattening (CFF), inserting subnodes (IS), longest common subsequence (LCS).
2. The postfix **W** or **B** means the method is white- or black-box, respectively.

Table 2: Averaged results of function database search. We mark best black-box repair methods .

Model	Setting	Poolsize			
		128	512	1000	10000
GNN-Talos	*Precision@1*	78.12%	63.36%	55.50%	23.34%
	HA-W	+3.51pt	+5.38pt	+5.30pt	+2.66pt
	PWG-B	-0.58pt	+1.58pt	+1.47pt	+0.88pt
	CFF-B	+0.39pt	+1.11pt	+1.08pt	+0.45pt
	IS-B	+1.43pt	+2.52pt	+1.87pt	+1.07pt
Structure2vec	*Precision@1*	74.09%	61.82%	52.68%	22.07%
	HA-W	+7.23pt	+7.18pt	+8pt	+3.65pt
	PWG-B	+2.34pt	+2.57pt	+3.03pt	+1.25pt
	CFF-B	-6.05pt	-7.91pt	-6.84pt	-2.72pt
	IS-B	+1.24pt	+0.34pt	+0.67pt	+0.48pt
Set2Set	*Precision@1*	83.53%	70.33%	63.03%	28.06%
	HA-W	+2.86pt	+4.36pt	+4.07pt	+2.13pt
	PWG-B	+0.78pt	+1.17pt	+0.85pt	+0.62pt
	CFF-B	+0.33pt	+1.76pt	+1.04pt	+0.86pt
	IS-B	+0.13pt	+0.85pt	+0.7pt	+0.59pt
GAT	*Precision@1*	74.61%	61.10%	51.58%	21.34%
	HA-W	+5.01pt	+5.88pt	+6.33pt	+3.0pt
	PWG-B	0pt	0pt	0pt	0pt
	CFF-B	-6.32pt	-4.82pt	-3.9pt	-1.07pt
	IS-B	+1.89pt	+2.25pt	+2.47pt	+1.21pt
Gemini	*Precision@1*	62.87%	54.22%	45.52%	20.65%
	HA-W	+3.21pt	+3.11pt	+4.20pt	+4.30pt
	PWG-B	+0.08pt	+0.22pt	-0.06pt	-0.14pt
	CFF-B	+1.56pt	+1.45pt	+1.71pt	+1.36pt
	IS-B	-5.58pt	-5.85pt	-5.65pt	-4.11pt
Asm2Vec	*Precision@1*	50.19%	43.49%	40.07%	31.55%
	PWG-B	+2.23pt	+1.47pt	+1.36pt	+1.84pt
	CFF-B	+2.72pt	+1.40pt	+1.37pt	+1.69pt
	IS-B	+2.45pt	+1.55pt	+1.77pt	+1.77pt
Commercial solution	*Precision@1*	75.61%	66.40%	62.05%	33.98%
	PWG-B	0pt	+0.13pt	+0.2pt	+0.83pt
	CFF-B	0pt	+0.10pt	+0.26pt	+0.87pt
	IS-B	0pt	+0.49pt	+0.65pt	+1.30pt
SAFE	*Precision@1*	54.89%	43.00%	35.58%	14.32%
	LCS	+7.49pt	+6.71pt	+5.83pt	+1.94pt
Total	W	+4.89pt	+5.43pt	+5.62pt	+2.46pt
	B	+1.53pt	+1.80pt	+1.72pt	+1.27pt

1. **Setting** abbreviations: Hungarian algorithm (HA), partial wheel graph (PWG), control-flow flattening (CFF), inserting subnodes (IS), longest common subsequence (LCS).
2. The postfix **W** or **B** means the method is white- or black-box, respectively.

6.2 Enhancing Downstream Applications

We consider two representative tasks, function database search and binary software component analysis (B-SCA).

Function Database Search. Following Sec. 6.1, we now evaluate BINAUG in a function database search task. This task often supports real-world applications of BCSA, such as code clone detection. We create function pools with various sizes following the setting in [82] and using functions of Dataset-2. Table 2 presents the results of the function database search. We observe that both of our white-box and black-box input repairing strategies exhibit similar behaviors compared with the results in Table 1. This suggests that BINAUG's performance is consistent across different scenarios. It is reasonable because the function database search (one to many search) deems a nature extension of the pair-wise comparison (one to one search) launched in Sec. 6.1. Overall, the results suggest that BINAUG is effective in augmenting important applications like code clone detection under different settings and for different models.

Binary Software Component Analysis (B-SCA). B-SCA [74, 98] is a popular downstream application of BCSA in the security community. It involves analyzing arbitrary executable files to identify potential Open Source Software (OSS) components reused in them. This way, we are able to identify possible vulnerabilities propagating from upstream OSS components to the analyzed executable. In short, B-SCA requires mapping each function in the executable to the most similar function in the function database, and then aggregates the results to generate a list of possible OSS components. However, a large function database is required to provide accurate results in a typical B-SCA setting, which can result in precision issues due to similar modules across different libraries or common control-flow structures shared by library utility functions and others [74, 98].

We implement the B-SCA analysis pipeline based on the setting in [90]. Given that the performance of similarity detection is the key component for B-SCA, we use the two best BCSA models (GNN-Talos and Set2Set) to support B-SCA. Accordingly, we use the white-box repairing (Hungarian algorithm) and the best black-box repairing methods (inserting subnode for GNN-Talos and control-flow flattening for Set2Set) to uncover the potential "upper-bound" improvement that can be offered by BINAUG. We select 14 different

Table 3: Results of B-SCA before and after using BinAug.

Model	Setting	Precision (%)	Recall (%)	F1 (%)
GNN-Talos	Default	19.9	28.4	23.4
	BinAug (white-box)	27.9	32.2	29.9
	BinAug (black-box)	25.5	22.8	24.1
Set2Set	Default	31.6	42.2	36.1
	BinAug (white-box)	45.9	43.1	44.5
	BinAug (black-box)	40.9	46.6	43.6
Commercial	Default	24.3	35.0	28.7
	BinAug (black-box)	36.6	37.0	36.8

Table 4: Overhead of white-box repairing.

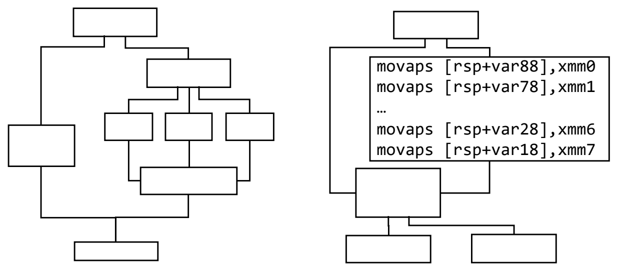
TOP-K	5	10	50	100
GNN-Talos	2%	3%	13%	23%
Set2Set	1%	2%	9%	16%

binaries covering different functionalities (ranging from cryptographic computation to database application) as the test inputs for B-SCA analysis and identify their reused OSS components. We create the function database with the latest version of 169 different common OSS libraries. Full details of the setup are provided at [4].

Table 3 presents the encouraging results of B-SCA before and after using BinAug, which notably increases the F1 scores around 8pt, a frequently-used measurement for evaluating SCA works [74, 90, 98]. We notice that BinAug brings significant improvement (around 12pt) on the precision of B-SCA, resulting in the increases of F1 scores. Our manual analysis reveals that BinAug can reduce many wrongly matched functions (i.e., false positive cases), resulting in the rises of precisions. For instance, `mod_pagespeed` is an Apache module depending on 51 OSS libraries. After applying the white-box repairing of BinAug, the number of false positive cases is reduced from 37 to 13. Overall, compared with black-box methods, the white-box method can constantly achieve higher precisions and F1 scores. This observation aligns with our findings in Sec. 6.1. Nevertheless, BinAug can still improve the Commercial BCSA tool's F1 score and precision for over 8pt and 12pt, respectively, with our black-box (subnode insertion) method.

6.3 Cost of BinAug

In this section, we assess the overhead of BinAug. Given a query function, Table 4 reports the overhead (as a fraction of total analysis time) for white-box methods. We use 111,592 query functions from the benchmark used in the B-SCA experiment of Sec. 6.2. For the white-box settings, BinAug brings 19.5% additional overhead on average, and we observe that the overhead of BinAug is proportional to the number of similar functions (K) considered; this is reasonable, as given the top-K matched functions for a query function, BinAug launches K pair-wise comparisons, where in each comparison, BinAug performs relatively expensive node alignment based on the intermediate representations in BCSA. Having that said, the absolute cost is still small: when $K = 100$, the total extra slowdown is 6918.7 seconds, meaning that the average cost for repairing one query function is only 0.062 ($\frac{6918.7}{111592}$) seconds. When implementing BinAug, we carefully optimize the intermediate computations (e.g., [8]), thus effectively reducing the overhead.



(a) `elf_vxworks_finish_dynamic_entry` (b) `_bfd_error_handler`

Figure 5: Illustration of mismatched function CFGs.

Despite the values of K in "top-K," black-box methods take a one-off effort to mutate the input function and generate the mutation's embedding. Then, re-ranking the top-K similar functions is performed by computing K `cosine` distances among embedding vectors of inputs and top-K functions. These steps are much faster compared with node alignment required by the white-box method. We omit the detailed results in Table 4 as the extra overhead is negligible in our evaluations.

6.4 Successes and Failures of BinAug

This section inspects the effectiveness of BinAug in addressing failure patterns observed in Sec. 3. To do so, for each BCSA model, we analyze 40 randomly-picked success and failure cases for every white-box and black-box setting, resulting in a total of 2,160 cases.[5] We present the analysis results below.

Success: Attention-Based Models. As elaborated in Sec. 4.1, intuitively, we mask stack operations to force the model focusing on instructions conveying meaningful and hopefully distinguishable functionality of a binary function. To confirm this, we randomly study 40 success cases. Overall, in 90% of the instances, BinAug successfully forces the model to focus on instructions that have low attention scores (under 10^{-4}) before repairing, resulting in over 7% increases of *Precision*@1 in Table 2. Here, we report that our experiment results support the design motivation in Sec. 4.1.

Success: GNN-Based Models. To quantify how BinAug fixes the error patterns observed in Sec. 3, we randomly pick and study 40 success cases when applying the Hungarian algorithm to each GNN-based BCSA model. In total, 14.5% of the successfully repaired cases belong to pattern ①. One such case is in Fig. 5: when querying the function database with function `_bfd_error_handler` (Fig. 5(b)) compiled with `gcc -O3`, the initial top-matched result (Fig. 5(a)) is incorrect. It is evident that the second block of `_bfd_error_handler` is filled with SSE instructions, while the matched function has no SSE instructions. Additionally, the control flow structure has significant differences. The incorrect match was due to over-smoothing, where the expressiveness of the node embeddings has been lowered and becomes mostly indistinguishable. We observe that after repairing, this incorrect matching is fixed with a much lower score.

[5]In line with Table 1, we have in total 27 black-box and white-box settings across all models; thus, we analyze $(40 + 40) \times 27 = 2160$ cases in total.

For the complex structure (pattern ③) failures, we successfully repair 73% cases, whose rationals are aligned with the above discussion. Moreover, we see that 4% of the cases belong to pattern ④ ("Others"), and the remaining 8.5% belong to pattern ②. Overall, the distribution of successfully-repaired cases correlates to the failure patterns observed in Sec. 3. This suggests that Binaug can effectively fix *common hurdles* faced by SoTA BCSA tools.

In sum, our GNN-based repairing approach is seen as effective to align nodes in the CFGs. As discussed in Sec. 4.1, we perform node alignment using the intermediate node representation. Compared to the input node representation and the node representation extracted from last layers of the network, the representation extracted from the intermediate layer of the encoder has captured the network topology with minimal effects from the over-smoothing problem. Using the intermediate layer representation, we repair the results through the averaged similarity of the matched nodes.

Success: Black-Box Methods. We examine randomly selected success matches made by BCSA models after repairing. In short, 51% of the successfully repaired case belongs to the complex structure (pattern ③), where 23.9% and 15% of the cases belong to patterns ① and ②, respectively. Overall, this trend matches with our expectation. Black-box repairing is design to regulate the input and provide "shortcut" for message passing, hence reducing the challenges imposed by complex and unregulated input structures. Moreover, we see that 9.6% of the cases belong to pattern ④ ("Others"), among which 59.2% of cases come from the commercial BCSA tool we benchmark. We believe this could be due to the difference in features used in the commercial tool, although it is obscure to confirm.

Failure: Attention-Based Models. Most failures are comparison between binaries of different bitnesses (32-bit vs. 64-bit). As already explained in Sec. 6.1, the different instruction sets result in Binaug's failures; see further discussion and future work in Sec. 7.

Failure: GNN-Based Models. In BCSA models based on GNNs, we find that Binaug failures are due to the fact that true positive matches are out of top-K. To solve this, we explore adjusting K by setting it to larger values (15 and 20). Nevertheless, this introduces minimal impact on Binaug's overall performance. Upon manual inspection of these cases, we find that the function query could have a significantly different graph topology than the correct answer. For instance, the `gcc-O3` version of `mp_mul_internal` has 109 basic blocks, while the function in the `gcc-O0` version only has 12 basic blocks. We conclude that these cases belong to the pattern ① we observed in Sec. 3.2, but it is generally difficult to solve.

Failure: Black-Box Methods. As shown in Table 1 and Table 2, the performance of BCSA may decrease notably when applying the control-flow flattening repair (except for the Set2Set case). With manual exploration, we believe this is presumably due to over-squashing. In short, control-flow flattening creates a bottleneck-like structure between the function entry block and the newly-inserted dispatcher node. As noted in previous work [76], this structure might reduce the expressiveness of the GNN model and result in performance degradation. While we believe that the Set2Set graph pooling layer [79] explains why the Set2Set model can alleviate this problem, exploring the root cause of this is hard, given explainable AI is still an open problem. We leave it as future work for selecting proper neural network architectures specifically for BCSA.

Table 5: Evaluating the synergy effect of Binaug.

Model	Setting	XO		XO+XC		XO+XC+XB	
		Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.
GNN-Talos	*Precision*@1	63.36%	9.53	74.87%	7.42	73.59%	9.6
	Synergy	+6.33pt	2.55	+5.69pt	2.19	+5.82pt	2.06
Structure2vec	*Precision*@1	60.49%	8.94	70.32%	7.03	67.3%	12.83
	Synergy	+9.44pt	2.16	+10.53pt	2.14	+10.38pt	3.68
Set2Set	*Precision*@1	72.39%	8.95	84.89%	4.72	83.49%	6.0
	Synergy	+4.63pt	2.03	+2.47pt	1.47	+2.50pt	1.45
GAT	*Precision*@1	59.26%	8.91	69.28%	7.43	67.03%	11.37
	Synergy	+7.36pt	2.69	+9.61pt	1.79	+8.02pt	2.30
Gemini	*Precision*@1	53.54%	11.18	46.07%	16.12	42.25%	15.20
	Synergy	+2.45pt	1.98	+3.53pt	1.87	+3.92pt	2.98
Average	Synergy	+6.04pt	-	+6.37pt	-	+6.13pt	-
Reference	White-box	+6.15pt	-	+7.41pt	-	+5.83pt	-
	Black-box	+2.08pt	-	+2.53pt	-	+2.44pt	-

6.5 Combining Black- and White-Box Methods

This section explores the potential synergy effects of our proposed black- and white-box repair methods. For each BCSA model, we combine the best-performing black-box and white-box methods according to the results in Sec. 6.1. Table 5 presents the result. Overall, compared with the white-box method, we find that the combination of white- and black-box methods can hardly further improve the accuracy.

We believe that the effect of our black-box repair methods is dominated by the white-box repair, which is intuitive. To explore this "synergy effect," we first transform the CFG of a query function with black-box methods. Then, after generating nodes' embeddings and performing an iteration of message passing, we employ Hungarian algorithm (i.e., the white-box repair) for node alignment. Recall that our black-box repair methods insert nodes and edges but keep the instruction sequence of each basic block unchanged. Hence, the initial representation of each node also remains unchanged. After an iteration of message passing, subnode insertion produces unchanged intermediate representations as it does not modify original edges, while the other two black-box repair methods remove original edges, which cannot facilitate the follow-up node alignment in the white-box method.

6.6 Hyper-Parameter Analysis

In the re-ranking process of Binaug (Sec. 4), there are two hyper-parameters, i.e., α and K. α decides the contribution of the re-ranked results to the final ranking, and K decides the number of results being considered in top-K. For previous evaluations, the default setting is $\alpha = 0.5$ and $K = 10$. Here, we study different values of α and K using the function database search application noted in Sec. 6.2. Due to limited space, this section reports the key findings, and the complete results are available on our webpage [6].

Specifically, we vary α from 0 to 1 and set $K = 10$; then, we evaluate the changed *Precision*@1 of models after using Binaug. We report that Binaug's optimal improvements, for *all models*, is achieved when $\alpha \in [0.4, 0.6]$. Thus, we interpret that our current setting, $\alpha = 0.5$, is reasonably good. Moreover, when changing K from 1 to 100, we record Binaug's performance. The values of improved *Precision*@1 for *all models* rise fast when $K < 10$, but they all saturate when $K > 20$. In this study, we set $K = 10$ to reach a balance between overhead and accuracy.

7 DISCUSSION

Threats to Validity. One threat to the validity of our study is whether BinAug is applicable to other BCSA tools. To address this, we have designed BinAug, an input repairing tool, that is mostly orthogonal to specific implementation details of BCSA tools. In addition, we have evaluated the effectiveness of BinAug across a set of popular BCSA tools with varying details. This alleviates the threat to generalization.

BinAug is designed to improve the accuracy of BCSA and likely benefits varying downstream applications of BCSA. In Sec. 6.2, we assess the effectiveness of BinAug's performance in enhancing the results of code clone detection and software component analysis. Technically speaking, BinAug can also be applied to other closely related areas, such as vulnerability detection in binary code.

Another possible threat to the validity of our results is over-specialization, meaning that our observations from the Sec. 3 may only apply to specific benchmark configurations. To address this concern, we use three non-overlapping datasets for conducting the preliminary study, training BCSA models, and evaluating BinAug. This shall mitigate the risk and illustrate BinAug's generalization as well. The effectiveness of BinAug on those large-scale benchmarks (each benchmark has over ten thousands cases, and in total we use 18 large open-source projects) indicates that our observations are not limited to specific settings and is likely to generalize to varying scenarios in the wild. We also study how different hyper-parameters of BinAug affect its performance in Sec. 6.6.

Future Work. As noted in Sec. 6.3, BinAug introduces overhead when it is required to mask instructions or generate many mutations for repairing inputs. While the overhead is acceptable for most cases, it may be prohibitive for large-scale usage. To alleviate this, one may explore only repairing inputs whose top-K matching scores are generally low or only repairing inputs with special patterns [95] that are prone to matching failures. Second, as mentioned in Sec. 6.4, the improvement of BinAug significantly decreases when repairing the cross-bitness matching of attention-based models. We envision that lifting the binary code to a platform-independent intermediate representation (IR) before using models is a potential solution. Also, both BCSA tools and BinAug heavily depend on the output quality of the underlying binary reverse engineering and analysis platform. If the binary executable is heavily obfuscated, de facto reverse engineering tools may fail to recover the correct CFG or assembly code, impeding BCSA tools and BinAug.

8 RELATED WORK

Binary Similarity Analysis. Besides eight evaluated tools, there are many other BCSA works. DeepBinDiff [29] generates instruction-level embedding with a pre-trained Word2Vec model [64]. Then, it produce embedding of a basic block by averaging the embeddings of the basic block's instructions. DeepBinDiff is a block-level matching technique which is not suitable for our framework. IMF-SIM [86] collect execution traces from functions and compute the similarity score of two binary functions by machine learning algorithms. Compared with computing cosine similarity of function-level embeddings, IMF-SIM is hard to scale up. Genius [33] and VulSeeker [35] encode basic blocks with manually-selected features, while recent works [50, 62, 80, 81] demonstrate semantic-aware techniques usually perform better in BCSA task.

DNN-Testing. With the increasing prevalence of machine learning model deployment in society, it is crucial to test deep neural networks (DNN) [105] to uncover defects. To address the complex testing oracle problems, metamorphic testing [22] has widely applied to generate test cases and achieve success in discovering defects of DNN models in various domains, such as image classification [30, 42, 75, 84, 102–104], NLP [18, 43, 59, 101], autonomous driving systems [65], and code completion systems [53]. For testing ML-based BCSA, input generation is challenging since randomly perturbing the binary code can easily break its functionality. Leveraging the reassembleable disassembling technique [83, 85] is a practical solution. Moreover, Wong et al. [89] test several SoTA GNN-based BCSA models with dataset augmented by obfuscation techniques, which effectively mitigates adversarial attacks.

9 CONCLUSION

This paper studies the limitations of de facto DNN-based BCSA tools and summarizes failure patterns. We then accordingly design a novel input repairing tool, BinAug, to enhance the performance of existing BCSA tools. Experimental results show that BinAug significantly improves the performance of SoTA BCSA tools across different benchmarks and tasks.

ACKNOWLEDGEMENTS

This work was supported in part by CCF-Tencent Open Research Fund and a RGC ECS grant under the contract 26206520. We are grateful to the anonymous reviewers for their valuable comments.

REFERENCES

[1] [n. d.]. 3.11 Options That Control Optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.
[2] [n. d.]. Average embedding vectors. https://datascience.stackexchange.com/questions/107462/why-does-averaging-word-embedding-vectors-exctracted-from-the-nn-embedding-laye.
[3] [n. d.]. binaug. https://github.com/wwkenwong/BinAug/.
[4] [n. d.]. BSCA. https://sites.google.com/view/binaug/bsca/.
[5] [n. d.]. Hungarian algorithm. https://en.wikipedia.org/wiki/Hungarian_algorithm.
[6] [n. d.]. Parameter Analysis. https://sites.google.com/view/binaug/parameter-analysis.
[7] [n. d.]. Result Website. https://sites.google.com/view/binaug.
[8] [n. d.]. Speedy cosine similarity computation in PyTorch. https://stackoverflow.com/questions/50411191/how-to-compute-the-cosine-similarity-in-pytorch-for-all-rows-in-a-matrix-with-re.
[9] [n. d.]. Yara. http://virustotal.github.io/yara/.
[10] 2014. BinDiff. https://www.zynamics.com/bindiff.html.
[11] Uri Alon and Eran Yahav. 2021. On the Bottleneck of Graph Neural Networks and its Practical Implications. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=i80OPhOCVH2
[12] László Babai and Ludek Kucera. 1979. Canonical labelling of graphs in linear average time. *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)* (1979), 39–46.
[13] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, behavior-based malware clustering.. In *NDSS*, Vol. 9. 8–11.

[14] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics *(NIPS 2018)*.

[15] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. 2012. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35 (2012), 1798–1828.

[16] Michael Brengel and Christian Rossow. 2021. {YARIX}: Scalable {YARA-based} malware intelligence. In *30th USENIX Security Symposium (USENIX Security 21)*. 3541–3558.

[17] Chen Cai and Yusu Wang. 2020. A Note on Over-Smoothing for Graph Neural Networks. *ICML Workshop: Graph Representation Learning and Beyond, 2020.* (2020).

[18] Jialun Cao, Meiziniu Li, Yeting Li, Ming Wen, S. C. Cheung, and Haiming Chen. 2020. SemMT: A Semantic-Based Testing Approach for Machine Translation Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31 (2020), 1 – 36.

[19] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 678–689.

[20] Di Chen, Shanshan Zhang, Jian Yang, and Bernt Schiele. 2020. Norm-Aware Embedding for Efficient Person Search. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020), 12612–12621.

[21] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *24th {USENIX} security symposium ({USENIX} security 15)*. 659–674.

[22] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong ....

[23] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative Embeddings of Latent Variable Models for Structured Data. In *International Conference on Machine Learning*.

[24] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries *(PLDI)*.

[25] Yaniv David and Eran Yahav. 2014. Tracelet-based Code Search in Executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 349–360.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (2019), 4171–4186.

[27] Kaize Ding, Zhe Xu, Hanghang Tong, and Huan Liu. 2022. Data Augmentation for Deep Graph Learning: A Survey. *SIGKDD Explor. Newsl.* 24, 2 (dec 2022), 61–77. https://doi.org/10.1145/3575637.3575646

[28] S. H. Ding, B. M. Fung, and P. Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *IEEE S&P*.

[29] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DEEPBINDIFF: Learning Program-Wide Code Representations for Binary Diffing. (2020).

[30] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, Jagadeesh Chandra J. C. Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018).

[31] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 303–317.

[32] 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.

[33] 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.

[34] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs *(ICICS)*.

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

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

[37] Irfan Ul Haq and Juan Caballero. 2021. A survey of binary code similarity. *ACM Computing Surveys (CSUR)* 54, 3 (2021), 1–38.

[38] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 63–72.

[39] SA Hex-Rays. 2014. IDA Pro: a cross-platform multi-processor disassembler and debugger.

[40] Yong Jin Kim (https://stats.stackexchange.com/users/188593/yong-jin kim). [n. d.]. What does average of word2vec vector mean? Cross Validated. arXiv:https://stats.stackexchange.com/q/318882 https://stats.stackexchange.com/q/318882 URL:https://stats.stackexchange.com/q/318882 (version: 2017-12-15).

[41] 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.

[42] Zhenlan Ji, Pingchuan Ma, Yuanyuan Yuan, and Shuai Wang. 2023. CC: Causality-Aware Coverage Criterion for Deep Neural Networks. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), 1788–1800.

[43] Mingyue Jiang, Tsong Yueh Chen, and Shuai Wang. 2022. On the effectiveness of testing sentiment analysis systems with metamorphic testing. *Inf. Softw. Technol.* 150 (2022), 106966.

[44] Giorgios Kollias, Shahin Mohammadi, and Ananth Y. Grama. 2012. Network Similarity Decomposition (NSD): A Fast and Scalable Approach to Network Alignment. *IEEE Transactions on Knowledge and Data Engineering* 24 (2012), 2232–2243.

[45] Oleksii Kuchaiev and Natasa Przulj. 2011. Integrative network alignment reveals large regions of global network similarity in yeast and human. *Bioinformatics* 27 10 (2011), 1390–6.

[46] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *IEEE S&P*.

[47] Eugene L Lawler. 1963. The quadratic assignment problem. *Management science* 9, 4 (1963), 586–599.

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

[49] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*. 184–195.

[50] Xuezixiang Li, Qu Yu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. (2021).

[51] 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.

[52] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Gated graph sequence neural networks. (2016).

[53] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. Cctest: Testing and repairing code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

[54] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. (2018).

[55] Zhouhan Lin, Minwei Feng, Cícero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. 2017. A Structured Self-attentive Sentence Embedding. (2017).

[56] Liang Liu, Bo Qu, Bin Chen, Alan Hanjalic, and Huijuan Wang. 2017. Modeling of Information Diffusion on Social Networks with Applications to WeChat. *ArXiv* abs/1704.03261 (2017).

[57] Linhao Luo, Gholamreza Haffari, and Shirui Pan. 2023. Graph sequential neural ode process for link prediction on dynamic and sparse graphs. In *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining*. 778–786.

[58] 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 Trans. Softw. Eng.* 43, 12 (Dec. 2017), 1157–1177.

[59] Pingchuan Ma, Shuai Wang, and Jin Liu. 2020. Metamorphic Testing and Certified Mitigation of Fairness Violations in NLP Models. In *IJCAI*. 458–465.

[60] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, X. Zhang, and Ananth Y. Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018).

[61] 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.

[62] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Proceedings of 16th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[63] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. *Proceedings 2019 Workshop on Binary Analysis Research* (2019).

[64] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. (2013).

[65] Qi Pang, Yuanyuan Yuan, and Shuai Wang. 2021. MDPFuzz: testing models solving Markov decision processes. *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2021).

[66] Changhua Pei, Yi Zhang, Yongfeng Zhang, Fei Sun, Xiao Lin, Hanxiao Sun, Jian Wu, Peng Jiang, Junfeng Ge, Wenwu Ou, and Dan Pei. 2019. Personalized re-ranking for recommendation. *Proceedings of the 13th ACM Conference on Recommender Systems* (2019).

[67] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/3132747.3132785

[68] 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).

[69] Danfeng Qin, Stephan Gammeter, Lukas Bossard, Till Quack, and Luc Van Gool. 2011. Hello neighbor: Accurate object retrieval with k-reciprocal nearest neighbors. *CVPR 2011* (2011), 777–784.

[70] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.

[71] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 815–823.

[72] Xiaohui Shen, Zhe L. Lin, Jonathan Brandt, Shai Avidan, and Ying Wu. 2012. Object retrieval and localization with spatially-constrained similarity measure and k-NN re-ranking. *2012 IEEE Conference on Computer Vision and Pattern Recognition* (2012), 3013–3020.

[73] Aarne Talman, Anssi Yli-Jyrä, and Jörg Tiedemann. 2019. Sentence embeddings in NLI with iterative refinement encoders. *Natural Language Engineering* 25, 4 (2019), 467–482.

[74] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. 2022. LibDB: An Effective and Efficient Framework for Detecting Third-Party Libraries in Binaries. *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)* (2022), 423–434.

[75] Yongqiang Tian, Shiqing Ma, Ming Wen, Yepang Liu, S. C. Cheung, and X. Zhang. 2021. To what extent do DNN-based image classification models make unreliable inferences? *Empirical Software Engineering* 26 (2021).

[76] Jake Topping, Francesco Di Giovanni, Benjamin Paul Chamberlain, Xiaowen Dong, and Michael M. Bronstein. 2022. Understanding over-squashing and bottlenecks on graphs via curvature. (2022).

[77] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[78] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. *International Conference on Learning Representations* (2018). https://openreview.net/forum?id=rJXMpikCZ

[79] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2016. Order matters: Sequence to sequence for sets. (2016).

[80] Huaijin Wang, Pingchuan Ma, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. sem2vec: Semantics-aware Assembly Tracelet Embedding. *ACM Transactions on Software Engineering and Methodology* 32 (2022), 1 – 34.

[81] 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 (2023), 226–250.

[82] 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. (2022). https://doi.org/10.1145/3533767.3534367

[83] Huaijin Wang, Shuai Wang, Dongpeng Xu, X. Zhang, and Xiao Liu. 2022. Generating Effective Software Obfuscation Sequences With Reinforcement Learning. *IEEE Transactions on Dependable and Secure Computing* 19 (2022), 1900–1917.

[84] Shuai Wang and Zhendong Su. 2020. Metamorphic Object Insertion for Testing Object Detection Systems. In *ASE*.

[85] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *USENIX Security*.

[86] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 319–330.

[87] Boris Weisfeiler and Andrei Leman. 1968. THE REDUCTION OF A GRAPH TO CANONICAL FORM AND THE ALGEBRA WHICH APPEARS THEREIN. *NTI, Series*, 2(9):12–16.

[88] Asiri Wijesinghe and Qing Wang. 2022. A New Perspective on "How Graph Neural Networks Go Beyond Weisfeiler-Lehman?". In *International Conference on Learning Representations*.

[89] Wai Kin Wong, Huaijin Wang, Pingchuan Ma, Shuai Wang, Mingyue Jiang, Tsong Yueh Chen, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Deceiving Deep Neural Networks-Based Binary Code Matching with Adversarial Programs. *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2022), 117–128.

[90] Seunghoon Woo, Sung-Hwuy Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), 860–872.

[91] Yan Xiao, Ivan Beschastnikh, David S. Rosenblum, Changsheng Sun, Sebastian G. Elbaum, Yun Lin, and Jin Song Dong. 2021. Self-Checking Deep Neural Networks in Deployment. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), 372–384.

[92] Yan Xiao, Yun Lin, Ivan Beschastnikh, Changsheng Sun, David Rosenblum, and Jin Song Dong. 2022. Repairing Failure-inducing Inputs with Input Reflection. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (2022).

[93] Xiaofei Xie, Wenbo Guo, L. Ma, Wei Le, Jian Wang, Lingjun Zhou, Yang Liu, and Xinyu Xing. 2021. RNNRepair: Automatic RNN Repair via Model-based Analysis. In *International Conference on Machine Learning*.

[94] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks? (2019).

[95] 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. (2023).

[96] 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 *CCS*.

[97] Hui Wang Ya Liu. 2018. Tracking Mirai variants. https://www.virusbulletin.com/conference/vb2018/abstracts/tracking-mirai-variants/.

[98] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. ModX: binary level partially imported third-party library detection via program modularization and semantic matching. In *Proceedings of the 44th International Conference on Software Engineering*. 1393–1405.

[99] 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. (2020).

[100] 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. In *Neural Information Processing Systems*.

[101] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2021. Enhancing Deep Neural Networks Testing by Traversing Data Manifold. *arXiv preprint arXiv:2112.01956* (2021).

[102] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Unveiling Hidden DNN Defects with Decision-Based Metamorphic Testing. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (2022).

[103] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2023. Revisiting neuron coverage for dnn testing: A layer-wise and distribution-aware criterion. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1200–1212.

[104] Yuanyuan Yuan, Shuai Wang, Mingyue Jiang, and Tsong Yueh Chen. 2021. Perception Matters: Detecting Perception Failures of VQA Models Using Metamorphic Testing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 16908–16917.

[105] J Zhang, Mark Harman, Lei Ma, and Yang Liu. 2019. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering* 48 (2019), 1–36.

[106] Xiaohui Zhang, Yuanjun Gong, Bin Liang, Jianjun Huang, Wei You, Wenchang Shi, and Jian Zhang. 2022. Hunting bugs with accelerated optimal graph vertex matching. *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022).

[107] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: simplifying programs for pre-trained models of code. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022).

[108] Tong Zhao, Yozen Liu, Leonardo Neves, Oliver J. Woodford, Meng Jiang, and Neil Shah. 2020. Data Augmentation for Graph Neural Networks. In *AAAI Conference on Artificial Intelligence*.

[109] Zhun Zhong, Liang Zheng, Donglin Cao, and Shaozi Li. 2017. Re-ranking Person Re-identification with k-Reciprocal Encoding. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), 3652–3661.

[110] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).