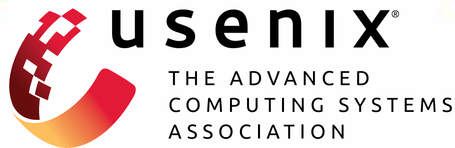

Improving ML-based Binary Function Similarity Detection by Assessing and Deprioritizing Control Flow Graph Features

Jialai Wang, *Tsinghua University;* Chao Zhang, *Tsinghua University and Zhongguancun Laboratory;* Longfei Chen and Yi Rong, *Tsinghua University;* Yuxiao Wu, *Huazhong University of Science and Technology;* Hao Wang, Wende Tan, and Qi Li, *Tsinghua University;* Zongpeng Li, *Tsinghua University and Quancheng Labs*

https://www.usenix.org/conference/usenixsecurity24/presentation/wang-jialai

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Improving ML-based Binary Function Similarity Detection by Assessing and Deprioritizing Control Flow Graph Features

Jialai Wang[*], Chao Zhang[*‡§], Longfei Chen[*], Yi Rong[*], Yuxiao Wu[†], Hao Wang[*], Wende Tan[*], Qi Li[*], and Zongpeng Li[*♣§]

[*]*Tsinghua University*, [†]*Huazhong University of Science and Technology*,
[‡]*Zhongguancun Laboratory*, [♣]*Quancheng Labs*

{wang-jl22, clf23, rongy19, hao-wang20}@mails.tsinghua.edu.cn, wyxhustcse@hust.edu.cn, twd2.me@gmail.com, {chaoz, qli01, zongpeng}@tsinghua.edu.cn

Abstract

Machine learning-based binary function similarity detection (ML-BFSD) has witnessed significant progress recently. They often choose control flow graph (CFG) as an important feature to learn out of functions, as CFGs characterize the control dependencies between basic code blocks. However, the exact role of CFGs in model decisions is not explored, and the extent to which CFGs might lead to model errors is unknown. This work takes a first step towards assessing the role of CFGs in ML-BFSD solutions both theoretically and practically, and promotes their performance accordingly. First, we adapt existing explanation methods to interpreting ML-BFSD solutions, and theoretically reveal that *existing models heavily rely on CFG features*. Then, we design a solution δCFG to manipulate CFGs and practically demonstrate the lack of robustness of existing models. We have extensively evaluated δCFG on 11 state-of-the-art (SOTA) ML-BFSD solutions, and find that the models' results would flip if we manipulate the query functions' CFGs but keep semantics, showing that *most models have bias on CFG features*. Our theoretic and practical assessment solutions can also serve as a robustness validator for the development of future ML-BFSD solutions. Lastly, we present a solution to utilize δCFG to augment training data, which helps deprioritize CFG features and enhance the performance of existing ML-BFSD solutions. Evaluation results show that, MRR, Recall@1, AUC and F1 score of existing models are improved by up to 10.1%, 12.7%, 5.1%, and 27.2% respectively, proving that *reducing the models' bias on CFG features could improve their performance*.

1 Introduction

Binary function similarity detection (BFSD) takes a pair of functions (usually in assembly or intermediate form) as input, and computes a similarity score between them [10, 38]. BFSD has proven effective in a broad spectrum of applications, such as known vulnerability discovery [12, 21, 29], malware lineage [2, 3, 30, 52], software plagiarism detection [43, 44, 58], patch analysis [27, 32, 68], and software supply chain analysis [25]. Given recent advances in machine learning (ML), SOTA BFSD are mostly ML-based [21, 37, 38, 46, 62–64, 67, 72], and we call them ML-BFSD solutions for brevity. Instead of heavily relying on manually specified features, such solutions design ML models to embed target binary functions into a latent vector space, where geometric metrics are adopted to compute function similarity.

We notice many ML-BFSD solutions [21, 38, 46, 67, 72] develop their models over CFGs, which characterize control dependencies within functions and help ML-BFSD solutions interpret function semantics. However, reliance on CFGs could impact the learning of semantics. For instance, solutions [38, 72] that use GNNs distribute instructions among different graph nodes based on control dependencies depicted by CFGs. While GNNs effectively perceive relationships between instructions within directly connected basic blocks, they struggle to learn relationships between instructions located in distantly separated blocks. This limitation indicates that CFG-based learning could restrict models' ability to learn function semantics. Since model decisions are influenced by both semantic and CFG features, the impact on learning semantics could lead to models' over-reliance on CFG features.

We also note several works [28, 39] raise the concern that some approaches heavily rely on CFG features, thereby underestimating the significance of semantics. Unfortunately, these works mention the over-reliance on CFGs issue based purely on experience or intuition, and the problematic approaches they point out are typically non-machine learning methods. Few studies explore the exact role of CFG features on decisions of ML-BFSD solutions. Furthermore, it is unknown to what extent the side effects introduced by CFGs could lead to model errors. Modern developers are still unaware of this issue. For instance, recent works [23, 33, 45, 66] still heavily rely on CFGs and are prone to errors. Given this, it is imperative to quantify the exact role of CFG features and systematically evaluate the severity of the over-reliance issue, serving as a

[§] Corresponding authors.

robustness validator for future work. More importantly, we should also provide a solution to mitigate the over-reliance problem and improve model performance.

We take the first step towards assessing the role of CFG features in ML-BFSD solutions. First, we theoretically interpret the importance of CFG features in ML-BFSD models, and show that *existing models heavily rely on CFG features*. Then, through practical experiments, we evaluate the robustness of existing models on CFGs. Specifically, we design a solution δCFG to manipulate CFGs, while keep function semantics intact. After manipulation, we evaluate existing models' performance on the modified functions. Results show existing models tend to make mistakes once the CFGs change, suggesting that *existing models have bias on CFG features*. Containing theoretical interpretation and practical experiments, our work serves as *a robustness validator* for developers. For example, they can use our work to determine whether a model exhibits over-reliance on CFG features, thereby guiding further improvements. Without a validator, even if developers recognize the over-reliance issue and make adjustments, it remains challenging to verify that the adjustment is effective. Further, we analyze why existing models are prone to such bias, and show δCFG can be used to augment training data and *improve models through deprioritizing CFG features*.

First, we analyze the importance of CFG features, by adapting existing explanation methods to ML-BFSD solutions. This is not a trivial task. In traditional scenarios, *e.g.*, computer vision, a single pixel can be treated as a feature, and explanation methods can be employed to calculate the importance of each pixel. Thus the importance of different features on model decisions is intuitive. However, in BFSD scenarios, it is inappropriate to model each byte as a feature, as binary code has higher level of semantics (*e.g.*, opcode, operands, CFGs, call graphs). Given this challenge, we design function features that are human-readable, which can represent both CFGs and function semantics. Based on this, explanation methods can intuitively reflect the importance of different features. We adapt approximation-based explanation methods to 11 representative ML-BFSD solutions, and reveal that CFG features are significantly more important than other features on most of these solutions. In other words, most of the existing ML-BFSD solutions heavily rely on CFG features.

In addition to the theoretical analysis on CFG importance, we further evaluate the robustness of existing models on CFG features with quantitative experiments. Particularly, in real-world software development, function CFGs can undergo changes due to various factors such as code refactoring. Such natural CFG transformations present unique challenges to ML-BFSD solutions. To simulate and assess these real-world, natural changes in CFGs, we design a solution δCFG, which can manipulate CFGs while maintaining function semantics. Given a pair of dissimilar (or similar) functions, δCFG can make them have similar (or dissimilar) CFGs. As a result, we have constructed hard-to-detect samples, including (1) different functions with identical CFGs and (2) same function with different CFGs, and evaluated 11 representative solutions on them. Evaluation results show that most existing models make mistakes on these samples. In other words, CFG features are not reliable and may cause bias to ML-BFSD solutions.

Further, we analyze why these solutions heavily rely on CFG features, with two potential reasons identified. (1) We identify design flaws in these ML-BFSD solutions that impede the learning of semantics. (2) We reveal that there is a bias when constructing training datasets. Existing ML-BFSD solutions construct training sets without considering function pairs with identical CFGs but different semantics, inducing the over-reliance on CFG features. Thus, we leverage δCFG to augment training data and enhance the performance of ML-BFSD solutions. Empirical results show δCFG can considerably enhance model performance, improving MRR, Recall@1, AUC and F1 score by up to 10.1%, 12.7%, 5.1%, and 27.2% respectively. We further explain the enhanced models, revealing that the over-reliance on CFG features has been deprioritized.

In summary, this work makes the following contributions.

- We take the first step towards assessing the role of CFG features in ML-BFSD solutions, with theoretic explanation and quantitative evaluation, and provide a robustness validator for developers to identify whether their models suffer from the over-reliance issue.
- We are the first to adapt explanation methods to the BFSD scenario for theoretic analysis, revealing that existing models heavily rely on CFG features.
- We present a CFG manipulation solution δCFG and conduct practical experiments, which shows that most models have bias on CFG features.
- We also present a data augmentation solution based on δCFG, which helps deprioritizes CFG features and promote existing models' performance.

2 Background and Related Work

2.1 ML-BFSD Solutions

Definition. ML-BFSD solutions aim to compute the similarity of two binary functions (raw bytes), measured with a similarity score from 0 (no similarity) to 1 (identical). Two functions compiled from two source code functions that are the same or similar to some extent (*e.g.*, one is a patched version of the other) should acquire high similarity scores [45,64]. Compared with Non-ML-BFSD solutions [14,15,29,49,53], ML-BFSD solutions [9,21,36,39,55,61,69,72,74] generally offer higher accuracy, require less human involvement, and have generalizes better. We present an intuitive example describing the workflow of ML-BFSD solutions in Figure 1.

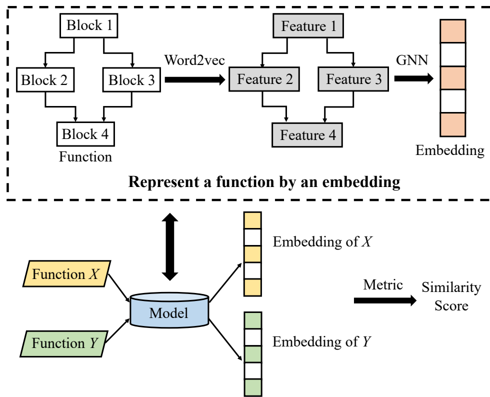

Figure 1: The general workflow of ML-BFSD solutions.

ML-BFSD solutions represent binary functions as embeddings through adopting ML models, and then apply specific metrics to compute the similarity score between functions. In practical applications, a given function is compared with all the functions in a function pool. Similarity scores are then used to rank functions in the pool, identifying the closest matches to the given function.

ML-BFSD solutions studied by us. CFGs encode function structures, depicting control dependencies between basic blocks. We note numerous ML-BFSD solutions treat CFGs [6, 45, 64] as critical components. However, the exact role of CFGs and the side effects of over-reliance on CFGs are not investigated. In this work, we conduct studies on 11 representative ML-BFSD solutions, which are proven to be top-performing, open-sourced and widely adopted [64]. Our work is model-agnostic, suitable for evaluating solutions regardless of whether they employ CFGs. To validate this point, beyond assessing solutions that use CFGs (the first eight [8, 16, 33, 38, 46, 50, 67, 72]), we also selected three solutions that don't use CFGs for evaluation (the last three, namely Safe [47], Trex [52] and jTrans [64]).

2.2 Explanation Methods

Explanation methods [41, 57, 70, 71] have been widely used in interpreting model behaviors. We aim to utilize explanation methods to study the role of CFGs. We hope that explanation methods can explicitly determine the importance of CFG features in model decisions. In this regard, we refer to approximation-based explanation methods [7, 22, 54, 56, 73].

Approximation-based approaches. We focus on local explanation approaches [7, 22, 54, 56, 73] which are the most prevailing form in the explanation domain [24, 40]. Such local approximation-based approaches concentrate on explaining the decision of target models for a given input instance. These approaches use interpretable models to locally approximate decision boundaries of target models in the vicinity of the input model. According to the interpretable models, these approaches can provide the importance score of each feature. Specifically, given an instance $\mathbf{x}$ with d features, *i.e.*, $\mathbf{x} = [x_1, x_2, ..., x_d]$, they perturb feature values of $\mathbf{x}$ under specific strategy to generate numerous variants (*i.e.*, local instances in the vicinity of $\mathbf{x}$) as training data. With training data, they adopt interpretable models f (e.g., linear models) to locally approximate decision boundaries of target models. When the approximation finishes, they utilize f to form a vector $\mathbf{a} = [a_1, a_2, ..., a_d]$, where a_i is treated as the contribution (importance score) of x_i to the decision of target models at $\mathbf{x}$.

Why choose approximation-based approaches? We choose approximation-based approaches for three reasons. First, such methods approximate target models with interpretable models, which can explicitly measure feature importance, helping understand the role of CFGs. Second, such approaches are model-agnostic. ML-BFSD solutions to be explained contain different types of models, such as GNNs and Transformers. We need a model-agnostic method that can be applied to explain all these models. Third, such methods are popular in explaining models deployed in function-related scenarios [13, 22, 40], which are closely related to our tasks.

There are other types of explanation approaches, such as backpropagation-based [11, 59, 60], perturbation-based [4, 17, 18], and GNN-based [41, 42, 42, 65, 71] approaches. We neglect these approaches since many of them are tailored for the image domain. Additionally, some of them focus on specific network architectures, *e.g.*, GNNs, and are not model-agnostic. For example, CFGExplainer [26] specifically supports GNN-based classifiers. In contrast, our task targets a broader range of models such as RNNs and Transformers. More importantly, we focus on exploring the role of CFG features. It is essential to separate CFG features from semantic features to ascertain the role of CFGs. CFGExplainer can only identify significant function sub-parts encompassing many instructions, which merge CFG features with semantic features, obscuring the role of CFG features.

2.3 Divergence from Related Work

Related work on model evaluation. A recent study [45] evaluates a batch of BFSD solutions, aiming at fair and meaningful comparison among them. It fails to consider the role of CFGs, and is considerably different from our work. They argue many solutions are tailored to slightly different objectives, depending on the context or specific scenarios, and as such, a direct comparison of BFSD solutions is infeasible. Thus, they focus on establishing a uniform dataset, and re-implement many BFSD solutions to conduct comparison with uniform metrics, such as recall and area under curve (AUC). Even though the study tries to observe the performance of BFSD solutions under different settings, for instance, by using different GNN variants and diverse feature engineering methods, they do not consider exploring the role of CFGs. In contrast, we focus on exploring the role of CFGs, model enhancement,

and mitigation on the over-reliance on CFG features.

Limited understanding on CFG over-reliance. Although several studies [28, 39] came across over-reliance on CFGs, the approaches they identify are non-ML models, which are rather different from ML-based models. For example, Liu *et al.* [39] comment that some non-machine learning approaches [20, 48] based on graph isomorphism heavily rely on CFGs. Furthermore, these studies often briefly acknowledge over-reliance based on experience or intuition, without a systematic evaluation of CFG's exact role. This makes it hard to effectively caution subsequent developers about the over-reliance. In line with this observation, later studies [38, 46, 72] tested by us still suffer from this issue. Our work fills this gap. It serves as a model-agnostic robustness validator, poised to be applied to future models, aiding in discerning their potential over-reliance on CFG features. Otherwise, even if developers refine their models, efficacy of such refinement is elusive.

Table 1: Human-readable features.

Type	Feature Name
Semantic Features	Call Instructions (Call)
	Jump Instructions (Jump)
	Arithmetic Instructions (Arith)
	Data Transfer Instructions (Data-Tran)
	Other Instructions (Other)
CFG Features	No. of Nodes (Nodes)
	No. of Edges (Edges)
	Graph Similarity (Graph-Sim)

3 Explainer: Explaining ML-BFSD Solutions

To understand the role of CFGs in ML-BFSD solutions, we implement an explanation module `Explainer`, which refers to prior art [7, 22, 54, 56, 73] and adapts them to the BFSD domain. The basic idea is to locally approximate a target model with an interpretable ML model, then present the importance of different features. Given a model F adopted by the target ML-BFSD solution, an instance (a function) $\mathbf{x}$, we aim to acquire the importance score of each feature in $\mathbf{x}$ to the decision of F at $\mathbf{x}$. (1) we first specify features to represent functions, supporting `Explainer` to score the importance of each feature. (2) Then, we generate local instances in the vicinity of $\mathbf{x}$, to locally approximate F with an interpretable model M. M explicitly represents the importance score of each feature, and the role of CFGs becomes intuitive. We introduce the above two points in Sections 3.1 and 3.2, respectively, and present the explanation results in Section 3.3.

3.1 Feature Specification

We are the first to apply explanation approaches to the BFSD domain, and existing explanation approaches lack feature specifications, particularly regarding the features depicting CFGs. This hinders `Explainer` from scoring the importance of each feature. We note some explanation methods [22] directly score function bytes, which is pointless in our task, since CFGs are hard to directly characterize using bytes, and the importance of CFGs is lacking in these explanation methods. To overcome this challenge, we specify human-readable features to represent functions. Each feature represents a component affecting decisions of ML-BFSD solutions, and can be scored by `Explainer` to intuitively reflect its importance. We specify eight features in Table 1, the first five are used to depict function semantics, namely *semantic features*, and the last three to characterize CFGs, namely *CFG features*.

Semantic features. We design semantic features to depict function semantics in two stages. *Stage I*: Instead of employing function bytes directly, we disassemble them into instructions, each viewed as an individual semantic feature. The motivation is two-fold: (1) Instructions, in contrast to bytes, are more human-readable, enhancing direct understanding of feature importance. (2) Instructions capture the entirety of a function's semantics without loss. However, acquiring global explanation results typically requires analyzing many functions consisting of a variety of instructions. Given the expansive nature of the instruction space and the myriad combinations of opcodes and operands, directly enumerating the importance scores of instructions becomes impractical.

Stage II: To address the above challenge, we categorize instructions into five common types: call, jump, arithmetic, data transfer, and others, collectively denoted as $T = \{T_1, T_2, T_3, T_4, T_5\}$. We first aggregate the importance scores of each type's instructions. A challenge arises from the potential variation in the number of instructions among these types, which could lead to biased perceptions of importance. We normalize these aggregated scores by dividing them by the total number of instructions within the respective type, thereby ensuring an equitable depiction of importance. Formally, if I_k is the set of instructions of type T_k and $s(i)$ indicates the importance score for instruction i, then the score for type T_k is $S_{T_k} = \frac{\sum_{i \in I_k} s(i)}{|I_k|}$. Using this approach, we compute scores for these five instruction types, presenting them as semantic feature scores to users.

CFG features. We then specify CFG features to depict characteristics of CFGs. We first consider the number of nodes and edges to represent the basic attributes of CFGs. Depicting CFGs is equivalent to depicting graphs, and these two features cannot distinguish graphs with the same number of nodes and edges. We further specify graph similarity, which reflects the difference between two graphs, and could distinguish graphs with the same size. We choose the widely-adopted Weisfeiler-Lehman optimal assignment kernel [34] to compute CFG similarity. For each local instance $\mathbf{x}'$ in the vicinity of $\mathbf{x}$, we compute the graph similarity between the CFGs of $\mathbf{x}'$ and $\mathbf{x}$, and treat it as a feature.

After feature specification, we represent $\mathbf{x}$ by d-dimension features, i.e., $\mathbf{x} = [x_1, x_2, ..., x_d]$. Here, $\mathbf{x}[1 : d-3]$ represents

all the instructions (stage I) within $\mathbf{x}$, while $\mathbf{x}[d-2:d]$ denotes CFG features. Explanation methods can then assign importance scores to each feature. Note that when presenting scores of semantic features to users, we compute scores for the five instruction types (stage II) based on the scores of each individual instruction and then present these scores to users.

3.2 Local Approximation

We then locally approximate decision boundaries of target models. (1) we first generate local instances in the vicinity of $\mathbf{x}$. (2) Then, we label these instances, constructing the training set. (3) We train interpretable models to locally approximate decision boundaries.

Local instance generation. The basic idea for the generation is to perturb $\mathbf{x}$ to generate local instances in the vicinity of $\mathbf{x}$ [22, 56]. Perturbations target both semantic and CFG features. To this end, we randomly delete instructions in $\mathbf{x}$ for the generation. From a semantic perspective, any instruction deletion naturally results in perturbation of semantic features. On the CFG side, if a jump instruction is deleted, it affects the number of CFG edges. Furthermore, if all instructions within a basic block are coincidentally deleted, it leads to the removal of that particular basic block, which then alters the number of CFG nodes. Both scenarios influence graph similarity. During the generation of local instance $\mathbf{x}'$, `Explainer` randomly deletes q of instructions, where q is uniformly sampled from $(0, \tau]$. Here we set the threshold τ to restrict the perturbations, to ensure $\mathbf{x}'$ is in the vicinity of $\mathbf{x}$. Each local instance is generated by applying a direct perturbation to the original $\mathbf{x}$, not iteratively on previously perturbed versions. Under this strategy, `Explainer` generates $\mathcal{N}$ unique local instances, which are utilized as training data.

Labeling training data. Then, we label the generated local instances for approximating decision boundaries. Given a local instance $\mathbf{x}'$, existing approximation-based methods label it with $F(\mathbf{x}')$, which is unsuitable in our scenarios. The main reason is that similarity detection models are trained in a contrastive learning manner, dramatically different from a supervised learning manner. Here, $F(\mathbf{x}')$ is a multi-dimension feature vector, which cannot be directly treated as a label.

Similarity detection models make decisions by computing the similarity between function feature vectors. Inspired by this, we propose labeling $\mathbf{x}'$ by computing the similarity between $\mathbf{x}'$ and $\mathbf{x}$. Specifically, we label $\mathbf{x}'$ with $D(F(\mathbf{x}'), F(\mathbf{x}))$, where D represents the cosine similarity function. Such labels reflect the impacts of feature perturbations on model decisions. Intuitively, a high $D(F(\mathbf{x}'), F(\mathbf{x}))$ implies $\mathbf{x}'$ and $\mathbf{x}$ are similar, and the corresponding perturbations on features do not significantly affect model decisions, and vice versa.

Model development. After labeling training data, we train interpretable models to locally approximate decision boundaries. Previous approximation-based methods assume local decision boundaries to be either linear [56], or non-linear [22]. This work applies both types of methods to interpret ML-BFSD solutions, since local boundaries vary in different solutions. Specifically, (1) for the linear hypothesis, we train a linear regression model f to locally approximate decision boundaries [56]. Recall we represent $\mathbf{x}$ as $[x_1, x_2, ..., x_d]$. After approximation, $f(\mathbf{x}) = \sum_{i=1}^{d}(w_i * x_i)$, where w_i is the coefficient of f that corresponds to the feature x_i, and the coefficient presents the importance of x_i. (2) For the non-linear hypothesis, we construct a mixture regression model to locally approximate F [22], and score feature importance.

3.3 Explanation Results

Experiment setup. To assess the role of CFGs, we adopt `Explainer` to explain 11 representative ML-BFSD solutions. Eight of them leverage CFGs: Genius [16], Asm2Vec [8], Gemini [67], GMN [38], GraphEmb [46], OrderMatters [72], XBA [33], and DEXTER [50]. The remaining three, Safe [47], Trex [52], and jTrans [64], do not. We randomly select 5,000 functions from open-source projects. For each function $\mathbf{x}$, we generate 1,000 local instances around it, and set the threshold $\tau = 0.5$. Then we apply both linear and non-linear models to the local approximation, as mentioned in Section 3.2. (1) Linear model. For each x and its local instances, we follow LIME [56] to train a Ridge regression model for the approximation. (2) Non-linear model. Similarly, we follow LEMNA [22] to train a Gaussian mixture model.

Metrics. As mentioned in Section 3.2, given a function $\mathbf{x}$, `Explainer` computes the importance scores of each feature. Rather than showing explanation results for each function in sequence, we evaluate overall results based on two metrics. First, we compute the **average importance score** of each feature. The average score of feature i is defined as $\frac{sum(x_i)}{num(\mathbf{x})}$, where $sum(x_i)$ is the sum score of feature i across all functions, and $num(\mathbf{x})$ is the number of functions. Second, we compute the **top-1 rate** of semantic features and CFG features, which are considered as the most important features. The top-1 rate of semantic features is defined as: $\frac{num(semantics)}{num(\mathbf{x})}$, where $num(semantics)$ is the number of functions whose highest scores are acquired by one of the semantic features. The top-1 rate of CFG features is defined as: $1 - \frac{num(semantics)}{num(\mathbf{x})}$.

Results. Overall, the results of average importance score and top-1 rate indicate CFG features are much more important than semantic features on most solutions. Thus, we conclude most of them heavily rely on CFG features. In detail, Table 2 shows average importance score. On most ML-BFSD solutions, both LIME and LEMNA yield the highest scores for CFG features. For example, in DEXTER, according to LIME, Graph-Sim acquires the highest score 0.163, while the importance scores of Call, Jump, Arith, Data-Tran, and Other are just 0.117, 0.126, 0.109, 0.108, and 0.116. Figure 2 and 3 present the results of top-1 rate for LIME and LEMNA, and we find that on most ML-BFSD solutions, CFG features

Table 2: Evaluation results of average importance scores on each similarity detection solution.

Explanation Method	BFSD Solutions	Average Score								CFG features score the highest?
		Semantic Features					CFG Features			
		Call	Jump	Arith	Data-Tran	Other	Nodes	Edges	Graph-Sim	
LIME	Genius	0.071	0.098	0.032	0.051	0.073	0.127	0.119	**0.144**	Yes
	Asm2Vec	0.063	0.088	0.046	0.055	0.085	0.116	0.124	**0.216**	Yes
	Gemini	0.056	0.109	0.054	0.050	0.065	0.142	0.143	**0.381**	Yes
	GMN	0.058	0.062	0.074	0.067	0.062	0.156	0.138	**0.384**	Yes
	GraphEmb	0.079	0.107	0.073	0.072	0.113	0.155	0.121	**0.278**	Yes
	OrderMatters	0.095	0.074	0.110	0.083	0.093	0.171	0.141	**0.234**	Yes
	XBA	0.154	0.118	0.100	0.103	0.108	0.119	0.108	**0.190**	Yes
	DEXTER	0.117	0.126	0.109	0.108	0.116	0.152	0.119	**0.163**	Yes
	SAFE	0.102	0.119	0.149	0.115	**0.152**	0.129	0.095	0.140	No
	Trex	0.115	0.118	0.128	0.122	**0.136**	0.130	0.122	0.130	No
	jTrans	0.127	**0.171**	0.108	0.124	0.129	0.117	0.106	0.126	No
LEMNA	Genius	0.088	0.096	0.074	0.082	0.088	0.118	0.109	**0.179**	Yes
	Asm2Vec	0.085	0.104	0.085	0.112	0.088	0.121	0.114	**0.182**	Yes
	Gemini	0.080	0.146	0.079	0.070	0.097	0.111	0.125	**0.291**	Yes
	GMN	0.090	0.097	0.113	0.109	0.102	0.152	0.104	**0.233**	Yes
	GraphEmb	0.108	0.147	0.102	0.103	0.155	0.111	0.100	**0.174**	Yes
	OrderMatters	0.061	0.105	0.072	0.080	0.093	0.168	0.143	**0.223**	Yes
	XBA	0.147	0.131	0.110	0.113	0.121	0.108	0.124	**0.186**	Yes
	DEXTER	0.120	0.124	0.113	0.114	0.122	0.146	0.123	**0.152**	Yes
	SAFE	0.108	0.126	0.158	0.124	**0.169**	0.103	0.091	0.121	No
	Trex	0.118	0.121	0.131	0.126	**0.133**	0.127	0.121	0.123	No
	jTrans	0.132	**0.162**	0.112	0.127	0.136	0.110	0.103	0.118	No

achieve a significantly higher top-1 rate than semantic features. This indicates CFG features have the greatest influence on most solutions, confirming the importance of CFGs in model decisions. Taking DEXTER as an example, according to LIME, the top-1 rate of CFG features is 2.9× higher than that of semantic features. We analyze the reason behind such over-reliance in Section 5.3. SAFE, Trex, and jTrans don't show a strong dependence on CFGs, which aligns with their non-utilization of CFGs. This alignment underscores `Explainer`'s capability to effectively assess the importance of CFG features.

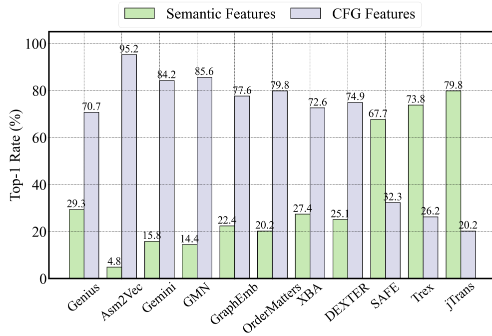

Figure 2: LIME shows the top-1 rate of semantic and CFG features. On most ML-BFSD solutions, CFG features outperform semantic features.

4 δCFG: Assessing the Impact of CFGs

After `Explainer` reveals the importance of CFG features, we aim to explore the impacts of CFGs on decisions of ML-BFSD

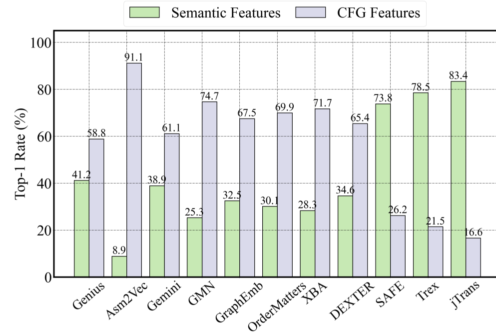

Figure 3: LEMNA shows the top-1 rate of semantic and CFG features. On most ML-BFSD solutions, the top-1 rate of CFG features is higher than that of semantic features.

solutions. We note that CFGs have a dynamic nature. Activities such as code refactoring [19] introduce changes in CFGs, mirroring the fluidity and evolution of software codebase. For instance, common refactoring techniques like method extraction, loop simplification, and dead code removal can substantially alter a CFG without changing function semantics. These changes are organic and are part of the life cycle of software development, aiming at improving code readability, maintainability, and performance. However, since ML-BFSD solutions heavily rely on CFG features, such changes might considerably affect model decisions.

We hope to explore the impact of CFG changes by manipulating CFGs of a given pair of functions to make them identical or different, without changing function semantics. In detail, (1) given an ML-BFSD solution model F, a pair of randomly selected functions **<x, y>** with different CFGs, we manipulate

their CFGs to make them identical. The modified functions are denoted as $<\mathbf{x}', \mathbf{y}'>$, and they have identical CFGs. We then investigate the impact of the modification on their similarity score, by comparing $D(F(\mathbf{x}'), F(\mathbf{y}'))$ and $D(F(\mathbf{x}), F(\mathbf{y}))$. (2) Conversely, given a randomly selected function $\mathbf{x_2}$, which has identical CFG as $\mathbf{x}$, $\mathbf{x}'$ has different CFG from $\mathbf{x_2}$. We compare $D(F(\mathbf{x}'), F(\mathbf{x_2}))$ and $D(F(\mathbf{x}), F(\mathbf{x_2}))$ to assess the impact when CFGs become different.

Furthermore, based on the manipulated functions, we can explore errors caused by the over-reliance on CFG features, which indicates that CFG changes in real-world software development can threaten ML-BFSD solutions. For example, we compare the similarity scores of the function pairs that have the same CFGs but different semantics against those with different CFGs but the same semantics. More specifically, we re-sample function pair $<\mathbf{x}, \mathbf{y}>$ that has different semantics, and function pair $<\mathbf{x}, \mathbf{x_2}>$ that has identical semantics (e.g., $\mathbf{x_2}$ is a compilation variant of $\mathbf{x}$). Then, after manipulation, if the similarity score $D(F(\mathbf{x}'), F(\mathbf{y}'))$ is higher than $D(F(\mathbf{x}'), F(\mathbf{x_2}))$, it indicates an error in the model, since the former with different semantics is claimed more similar than the latter with identical semantics. Such errors can jeopardize downstream tasks that use ML-BFSD solutions. Taking malware clustering as an example, the analysts adopt ML-BFSD solutions to identify common functions across different malware samples according to their similarity [1, 30]. In this case, the wrong behaviors of ML-BFSD solutions could introduce irrelevant functions, compromising the accuracy of clustering.

4.1 Methodology

We propose δCFG to manipulate function CFGs to be identical or different. The main challenge lies in automating the manipulation while preserving function semantics: (1) the manipulation should be automated, since comprehensive evaluation and exploring model errors requires manipulating abundant function pairs. The cost of human intervention is unacceptable. (2) As δCFG needs to verify the importance of CFGs, the manipulation should preserve the original semantics, which avoids introducing altered semantics that affect model decisions. Once the manipulation of CFGs is made, any change in model decisions can be attributed to CFGs.

We note that obfuscation techniques [5, 31] can also modify CFGs without altering semantics. However, these techniques are not appropriate in our scenario. Firstly, they introduce drastic code changes that are not common in software evolution. Secondly, they typically make CFGs different, and it is challenging to make CFGs identical.

To address the above challenge, we propose a solution δCFG. We show the basic idea in Figure 4, and the detailed workflow in Algorithm 1. δCFG contains two algorithms: the *basic block match algorithm*, and the *edge match algorithm*. Using these two algorithms enables δCFG to automatically manipulate CFGs of function pairs, and make them identical, without disturbing the semantics. Moreover, these algorithms can also manipulate CFGs to be different. Consider a pair $<\mathbf{x}, \mathbf{x_2}>$ that has identical CFGs. If we select another function $\mathbf{y}$ with a distinct CFG and apply our algorithms, we obtain a pair $<\mathbf{x}', \mathbf{y}'>$ with identical CFGs. This inherently makes CFGs of $\mathbf{x}'$ and $\mathbf{x_2}$ different. To this end, we primarily introduce how to make the CFGs of function pairs identical, with the divergence of CFGs emerging as an inherent consequence.

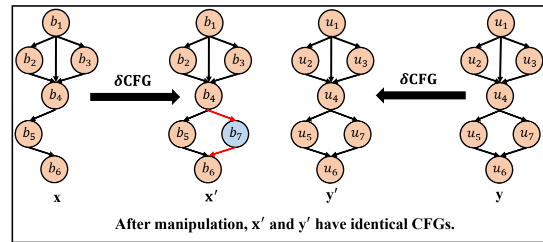

Figure 4: Basic idea of δCFG. Red edges and blue basic blocks are introduced by δCFG.

In this context, V represents the set of basic blocks, and E denotes the set of edges. To generate identical CFGs for $\mathbf{x}$ and $\mathbf{y}$, we essentially establish two mappings between their modified counterparts, $\mathbf{x}'$ and $\mathbf{y}'$. The first is a basic block mapping, $f_v : V_{\mathbf{x}'} \rightarrow V_{\mathbf{y}'}$. That is, there is a one-to-one correspondence between each basic block in $V_{\mathbf{x}'}$ and each basic block in $V_{\mathbf{y}'}$. We use the basic block match algorithm to establish this mapping. The second is an edge mapping, $f_e : E_{\mathbf{x}'} \rightarrow E_{\mathbf{y}'}$. We establish the one-to-one correspondence between each edge in $E_{\mathbf{x}'}$ and each edge in $E_{\mathbf{y}'}$, by using the edge match algorithm.

Basic block match algorithm. To establish the basic block mapping, we match each basic block of $\mathbf{x}$ with that of $\mathbf{y}$, resulting in a one-to-one correspondence. Overall, we first match the entry block of $\mathbf{x}$ with that of $\mathbf{y}$. Then, using these two entry blocks as starting points, we use a breadth-first search (BFS) strategy to match the remaining blocks across both functions.

Given that CFGs are directed graphs, when matching basic blocks, it is logical to match blocks at similar depths within the two CFGs. Thus, we employ BFS, which enables match progress from earlier depths to later depths within the CFGs. Our BFS-driven algorithm begins by matching the entry blocks b_{entry} and u_{entry} ($b \in \mathbf{x}$ and $u \in \mathbf{y}$). As BFS unfolds, we match the direct successors of each previously matched pair, aiming to match blocks at similar depths in their respective CFGs.

In detail, for each pair (b_i, u_i) that has been matched during BFS traversal, we identify their direct successor sets as S_{b_i} and S_{u_i}. We remove basic blocks from S_{b_i} and S_{u_i} that are already matched, resulting in refined sets M_{b_i} and M_{u_i}. The question left is how to match basic blocks within these two sets. We leverage the principle that basic blocks with similar connectivity patterns within CFGs should match. That is, blocks with a larger number of direct successors should be matched with each other, and vice versa. To this end, the basic blocks in M_{b_i}

and M_{u_i} are sorted in descending order based on their number of direct successors and matched accordingly. When blocks have identical direct successor counts, we count their paths toward the final block in CFGs and prioritize matching blocks with more paths. If the number of paths is equal, we randomly match blocks. We also recognize size disparities between M_{b_i} and M_{u_i} can occur. That is $|M_{b_i}| \neq |M_{u_i}|$. To solve this problem, we introduce $k = |M_{b_i}| - |M_{u_i}|$ empty basic blocks to $\mathbf{y}$ if $k > 0$, or infuse $-k$ empty blocks to $\mathbf{x}$ if $k < 0$. This strategy ensures that each block from $\mathbf{x}$ and $\mathbf{y}$ is matched with a corresponding counterpart, whether it is a genuine basic block or an introduced empty one. This iterative process proceeds until all blocks within both $\mathbf{x}$ and $\mathbf{y}$ find their counterpart.

Upon completing block mapping, the algorithm introduces empty basic blocks, transitioning the block sets for functions $\mathbf{x}$ and $\mathbf{y}$ from $V_{\mathbf{x}}$ and $V_{\mathbf{y}}$ to $V_{\mathbf{x}'}$ and $V_{\mathbf{y}'}$ respectively. Now, both $V_{\mathbf{x}'}$ and $V_{\mathbf{y}'}$ contain an identical number of blocks. Assuming each now consists of m blocks, we can represent them as $V_{\mathbf{x}'} = \{b_1, b_2, \dots, b_m\}$ and $V_{\mathbf{y}'} = \{u_1, u_2, \dots, u_m\}$. Here, b_i in $V_{\mathbf{x}'}$ corresponds directly to u_i in $V_{\mathbf{y}'}$. The block mapping will be used when establishing the following edge mapping.

Edge match algorithm. We design this algorithm to establish the edge mapping. The basic idea is to conduct a bijection between edges of $\mathbf{x}$ and $\mathbf{y}$. We iterate through each edge in $\mathbf{x}$, and if its corresponding edge is absent in $\mathbf{y}$, we add the required edge to $\mathbf{y}$. This process is then reciprocated for $\mathbf{y}$.

To traverse each edge in $\mathbf{x}$, we iterate through each basic block in $V_{\mathbf{x}'} = \{b_1, b_2, \dots, b_m\}$, and record the direct successors of each basic block with a corpus $S_{\mathbf{x}'} = \{S_{b_1}, S_{b_2}, ..., S_{b_m}\}$. Each $S_{b_i} \in S_{\mathbf{x}'}$ denotes direct successors of b_i, indicating there are directed edges from b_i to each basic block in S_{b_i}. These directed edges should all have corresponding edges in $\mathbf{y}$. Recalling the basic block mapping, we note that b_i matches u_i, and all basic blocks in S_{b_i} also match the basic blocks in $V_{\mathbf{y}'}$, denoted as S_{u_i}. To this end, for each $u_i \in V_{\mathbf{y}'}$, we check if there are directed edges from u_i to each basic block in S_{u_i} one by one. If certain edges are missing, they are appended to $\mathbf{y}$. After traversing each edge in $\mathbf{x}$, we reciprocate the process for $V_{\mathbf{y}'}$. After performing the bidirectional matching on both $\mathbf{x}$ and $\mathbf{y}$, the edge match algorithm completes a comprehensive mapping for all edges between the two functions.

After adopting the two match algorithms, δCFG results in $\mathbf{x}'$ and $\mathbf{y}'$. The CFGs of them become identical, as shown in Figure 4. We note that δCFG introduces empty basic blocks, which do not compromise the semantics. However, the introduced edges might perturb original control flows and compromise original semantics. To solve this problem, we manipulate branch conditions, such as setting corresponding conditions to be always true. This guarantees functions maintain their original control flows, steering clear of the newly added branches, and thus preserving original semantics. We introduce details in Section 4.2. Additionally, observing the characteristics of the two algorithms, we can discern the CFG change on $\mathbf{x}$ (or $\mathbf{y}$) depends on the CFG of $\mathbf{y}$ (or $\mathbf{x}$). As long as we select target functions with diverse CFGs, we can achieve diverse CFG changes. This implies that our δCFG can simulate a wide variety of CFG changes that occur during software development.

Algorithm 1: The workflow of δCFG

```
Input: x = (V_x, E_x), y = (V_y, E_y);
Output: the manipulated functions x' = (V_x', E_x'), y' = (V_y', E_y');
Function δCFG:
    V_x' = V_x, V_y' = V_y, E_x' = E_x, E_y' = E_y;
    f_v = Basic-block-match(V_x', V_y');
    Edge-match(f_v, V_x', V_y', E_x', E_y');
    return x' = (V_x', E_x'), y' = (V_y', E_y');
Function Basic-block-match(V_x', V_y'):
    f_v = {}; // basic block mapping
    f_v[b_entry] = u_entry; // match entry blocks of x and y
    Q = Queue().push((b_entry, u_entry)); // first in first out queue
    BFS(Q, f_v, Vis, V_x', V_y');
    return f_v;
Function BFS(Q, f_v, V_x', V_y'):
    if Q.isempty() then
        return;
    b_i, u_i = Q.pop();
    Vis.append((b_i, u_i));
    N_b = [], N_u = []; // empty list to record the number of
        direct successors
    M_bi, M_ui = Remove(GetSuccessors(b_i, u_i)); // remove
        successors that have been matched
    for b in M_bi do
        N_b.append(|GetSuccessors(b)|);
    Sort all basic blocks in M_bi in descending order based on N_b;
    for u in M_ui do
        N_u.append(|GetSuccessors(u)|);
    Sort all basic blocks in M_ui in descending order based on N_u;
    k = |M_bi| - |M_ui|;
    if k > 0 then
        for j = 1 to |M_ui| do
            f_v[M_bi[j]] = M_ui[j];
            Q.push((M_bi[j], M_ui[j]));
        for j = |M_ui| + 1 to |M_ui| + k do
            u_empty = CreateBasicblock();
            V_y' = V_y' ∪ u_empty;
            f_v[M_bi[j]] = u_empty;
            Q.push((M_bi[j], u_empty));
    else
        for j = 1 to |M_bi| do
            f_v[M_bi[j]] = M_ui[j];
            Q.push((M_bi[j], M_ui[j]));
        for j = |M_bi| + 1 to |M_bi| - k do
            u_empty = CreateBasicblock();
            V_x' = V_x' ∪ u_empty;
            f_v[u_empty] = M_ui[j];
            Q.push((u_empty, M_ui[j]));
    BFS(Q, f_v, V_x', V_y');
Function Edge-match(f_v, V_x', V_y', E_x', E_y'):
    for b in V_x' do
        u = f_v[b];
        for s in GetSuccessors(b) do
            s' = f_v[s];
            E_y' = E_y' ∪ (u, s');
    for u in V_y' do
        b = f_v[u];
        for s in GetSuccessors(u) do
            s' = f_v[s];
            E_x' = E_x' ∪ (b, s');
```

4.2 Implementation

δCFG requires source code of functions, which is easily available. δCFG manipulates CFGs on LLVM intermediate representation (IR) [35]. After δCFG changes CFGs on IR, we use clang to compile the IR to binary. Here, we introduce key implementation details. For the basic block match and edge match algorithms, they need to create basic blocks, determine successors of basic blocks, and add edges. LLVM offers functions to implement all these operations. More specifically, we apply `Create` to create empty basic blocks, and `getSuccessor` to determine successors of basic blocks. The crux is how to add edges.

Edge addition. Given a basic block $b_i \in V_{\mathbf{x}'}$ with $|S_{b_i}|$ direct successors before edge addition, we suppose the number of direct successors increases to M after the addition ($|S_{b_i}| < M$). (1) When $M = 1$, it is an indication of a situation that requires the addition of a single edge. To facilitate this, we directly employ the `CreateBr` function to create this edge. (2) When $M = 2$, it signifies a conditional jump situation, *i.e.*, two edges need to be added. This scenario requires the use of the `CreateCondBr` function to introduce the two edges. (3) When $M > 2$, this implies a switch case scenario (our analysis reveals that at the IR level, a switch case would uniquely result in more than two branches). For adding multiple edges in such a scenario, we utilize function `SwitchInst::Create` to introduce a switch-case construct.

Semantics preservation amid edge addition. When a basic block already has direct successors, *i.e.*, $|S_{b_i}| \geq 1$, the addition of edges implies replacing the previous branch instruction with a new one, and introducing new branches, which may compromise function semantics.

For $|S_{b_i}| = 1$, it indicates b_i originally has a direct successor, and the existing branch instruction is a direct jump instruction. Given the fact that $M > |S_{b_i}|$, when $M = 2$, we need to substitute the existing branch instruction with a conditional jump. To maintain semantics, we ensure the path from b_i to b_j is consistently executed by setting the corresponding condition to be always true. When $M > 2$, we substitute the existing branch instruction with a switch-case construct. Then, we designate the preferred path (b_i to b_j) as the default. By manipulating the corresponding switch condition, we ensure that the default branch is invariably taken.

For $|S_{b_i}| = 2$, it indicates b_i originally has two direct successors, and the existing branch instruction is a conditional jump instruction. Given that $M > |S_{b_i}| = 2$, we substitute the branch instruction with a switch-case construct. Recognizing that the condition of the original branch evaluates to boolean values true or false, we employ the `CreateZExt` to map true to 1 and false to 0. These mapped values then serve as switch variables for the switch-case construct. Specifically, the original path taken when the condition is true is now associated with case 1 in the switch-case construct. Similarly, the original path taken when the condition is false is associated with case 0. Any other branches in the switch-case construct are designed not to be taken, thus preserving the original semantics.

For $|S_{b_i}| > 2$, it implies b_i has multiple direct successors, and the existing branch instruction is a switch-case construct. Accordingly, we add $M - |S_{b_i}|$ branches by introducing new cases to a switch-case construct. We employ the `addCase` for this purpose. By carefully setting case values, we ensure that these new branches remain untaken, thereby maintaining the original semantics. Here, how to set case values is the crux. More specifically, assuming S_{b_i} contains n basic blocks ($|S_{b_i}| = n$), given a switch variable *var*, there are n case values, represented as $C = \{C_0, C_1, ..., C_{n-1}\}$. Each case value C_j corresponds to a basic block b'_j in $S_{b_i} = \{b'_0, b'_1, ..., b'_{n-1}\}$. During each execution, the switch variable *var* will take on one of the case values, subsequently executing the corresponding basic block. To maintain the original semantics, when adding $M - n$ new cases, it's essential to ascertain that these new case values do not overlap with the existing ones in C. Thus, these new case values will never equal *var* and the newly added branches will never be executed. This approach maintains the original semantics.

The question left is how to ensure that the newly added case values do not overlap with C. The main challenge lies in ensuring that the newly added case values do not overlap with C. It's noteworthy that each function may have a unique set of C. To automate the addition of non-overlapping case values, we replace all existing case values in C with a continuous sequence $\widetilde{C}$ ranging from 0 to $n-1$, i.e., $\widetilde{C} = \{0, 1, ..., n-1\}$. That is, now each case value $\widetilde{C}_j$ corresponds to basic block b'_j. Concurrently, we adjust the switch variable *var* to match this new sequence, yielding var_{new}. The adjustment is realized through the operation: $var_{new} = \bigvee_{1 \leq j \leq n-1} \texttt{CreateSExt}(var == C_j) \wedge j$, where LLVM function `CreateSExt` extends a signed value to 32 bits. To this end, when $var = C_j$, var_{new} equals to $\widetilde{C}_j$, and b'_j will be executed. Thus, the case value replacement will not compromise the original semantics. With this replacement in place, when new case values arise, we can straightforwardly set their values greater than $n-1$, automating the addition of case values. As the new case values exceed $n-1$, they do not overlap with $\widetilde{C}$, without compromising the original semantics.

5 Evaluation on δCFG

We adopt δCFG to answer four research questions (RQs):

- **RQ1:** What is the impact on model decisions when a pair of functions with identical CFGs are modified to have different CFGs? (Section 5.1)
- **RQ2:** What is the impact on model decisions when a pair of functions with different CFGs are modified to have identical CFGs? (Section 5.2)

- **RQ3:** How come these ML-BFSD solutions rely heavily on CFGs? (Section 5.3)
- **RQ4:** Can our δCFG be utilized to mitigate the over-reliance and enhance the performance of ML-BFSD solutions? (Section 5.4)

5.1 Change Identical CFGs to Be Different

Overall, results reveal that for the majority of the ML-BFSD solutions, the similarity scores for function pairs with identical CFGs experience a substantial decrease when their CFGs become different. More importantly, these changes in CFGs could result in model errors. That is, functions that are semantically identical might be incorrectly identified as dissimilar.

Experimental setup. For each ML-BFSD solution **F**, we randomly select 5,000 function pairs with identical CFGs and semantics to investigate the impact of modifying CFGs. During the selection of each function **x**, we traverse all possible combinations of compilation variants, such as (O0, O1) and (O0, O2), until we identify a pair of variants whose CFGs are identical. If no such pair is found, the function is not selected. Ultimately, we collect 5,000 function pairs with identical CFGs and semantics, each denoted by $<\mathbf{x}, \mathbf{x_2}>$.

Evaluation metrics. We use decrease ratio and error rate (ER) to evaluate the impact when CFGs become different. (1) Similarity scores will decrease when CFGs of pairs change from being identical to different. We evaluate the decrease in their similarity score by *decrease ratio*: $\frac{\mathbf{F(x,x_2)-F(x',x_2)}}{\mathbf{F(x,x_2)}}$. (2) We further measure *ER* caused by CFG changes. Recalling scenarios where ML-BFSD solutions are used, given a **x**, the task is to identify the closest counterpart $\mathbf{x_2}$ from a pool of candidate functions. We mimic real-world scenarios where **x** has been changed to $\mathbf{x'}$ due to CFG changes such as code refactoring, and test whether **F** can still identify $\mathbf{x_2}$ as the most similar function despite the CFG changes.

For each function **x**, we create four function pools with sizes of 16, 32, 64 and 128, respectively, with functions randomly selected from open-source projects. We guarantee that, within each pool, the similarity score between **x** and $\mathbf{x_2}$ ranks at the top. Subsequently, upon computing similarity scores between $\mathbf{x'}$ and all functions in the pool (including $\mathbf{x_2}$) using **F**, we rank all functions based on these scores. We evaluate whether the pair $<\mathbf{x'}, \mathbf{x_2}>$ maintains its top rank despite the CFG changes. Any deviation from the top rank is deemed an error. Thus, $ER = \frac{num(error)}{num(all)}$, where $num(error)$ is the number of errors, and $num(all)$ is the total number of function pairs, *i.e.*, 5,000, declared in our setup.

Results. Figure 5 shows the decrease ratios observed. ML-BFSD solutions that rely on CFGs are sensitive to CFG changes, reflecting high decrease ratios with many function pairs experiencing a decline in similarity scores by 40% ∼ 60% or more than 60%. Taking OrderMatters as an example,

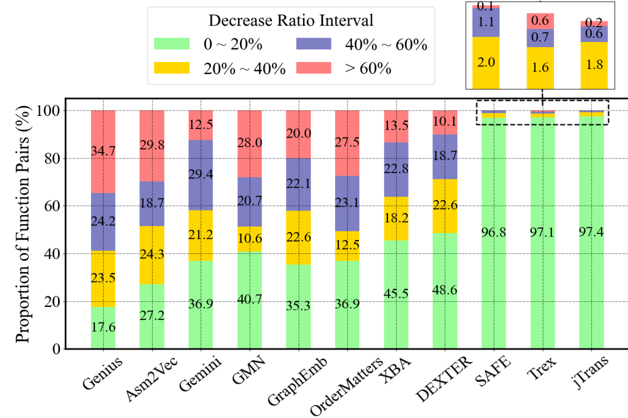

Figure 5: Evaluation results of decrease ratios. In non-CFG dependent solutions, most function pairs exhibit decrease ratios within the 0 ∼ 20% interval (represented by the green bars).

23.1% function pairs witness a decrease of 40% ∼ 60%, and 27.5% experience a decrease exceeding 60%.

In contrast, in non-CFG dependent solutions, namely SAFE, Trex, and jTrans, a significant majority of function pairs exhibit decrease ratios within the 0 ∼ 20% interval, while only a minor fraction of function pairs show higher decrease ratios. Such results indicate these solutions are insensitive to CFG changes. For instance, in jTrans, 97.4% of function pairs show decrease ratios within 20%, while only 2.6% of function pairs exceed 20%.

The ER results in Table 3 highlight that a majority of solutions demonstrate a substantial ER, even at the smallest pool size of 16. This suggests that when CFGs of function pairs differ, many models often misjudge them as dissimilar, despite their identical semantics. To elaborate, with a pool size of 16, most solutions' ER is still above 40%. As we increase the pool size from 16 to 128, the ER for the majority further escalates significantly.

Summary. Most solutions exhibit high decrease ratios and ER due to the over-reliance on CFGs, revealing a notable model limitation. In contrast, SAFE, Trex, and jTrans diverge in performance compared to others, displaying much lower ratios and ER. This aligns with our explanation results that these three do not heavily rely on CFGs.

5.2 Change Different CFGs to Be Identical

We further consider scenarios where function pairs with different CFGs are changed to have identical CFGs. On most ML-BFSD solutions, we observe that such changes induce a significant similarity score increase, despite different function semantics. Furthermore, a considerable number of function pairs are misjudged as similar, with similarity scores rising to the top position.

Table 3: Evaluation of ER when CFGs become different.

BFSD Solutions	ER (%)			
	pool size = 16	pool size = 32	pool size = 64	pool size = 128
Genius	62.7	67.9	73.3	75.1
Asm2Vec	31.7	37.4	43.8	48.0
Gemini	40.1	49.2	57.4	65.3
GMN	42.2	47.6	54.7	64.1
GraphEmb	38.7	45.6	52.1	60.2
OrderMatters	57.8	65.1	70.7	75.2
XBA	52.0	62.5	70.6	76.0
DEXTER	46.5	56.3	63.7	70.6
SAFE	1.4	1.8	2.0	2.5
Trex	1.3	1.5	2.2	4.1
jTrans	1.1	1.5	2.4	3.0

Experimental setup. For each solution **F**, we randomly select 5,000 function pairs with different semantics from open-source projects. Each function pair <**x**, **y**> is chosen such that $\mathbf{F}(\mathbf{x},\mathbf{y})$, their similarity score, is lower than the score between **x** and its compilation variants, ensuring that the semantics of <**x**, **y**> are different and can be correctly distinguished by **F**. Then, δ`CFG` manipulates each pair <**x**, **y**> into <**x**′, **y**′>. Here, **x**′ and **y**′ have identical CFGs but different semantics.

Evaluation metrics for RQ2. We use increase ratio, error rate (ER), and top-1 rate to evaluate for the evaluation. When function pair CFGs are changed to be identical, (1) we assess the improvement of their similarity scores by measuring *increase ratio*: $\frac{\mathbf{F}(\mathbf{x}',\mathbf{y}')-\mathbf{F}(\mathbf{x},\mathbf{y})}{\mathbf{F}(\mathbf{x},\mathbf{y})}$. (2) Following the ER metric defined in Section 5.1, we construct four function pools of sizes 16, 32, 64, and 128 in a similar way. Each pool encompasses a randomly selected compilation variant of **x**, denoted as $\widetilde{\mathbf{x}}$. Additionally, each pool includes **y**′ to emulate a scenario in which the CFG of **x** is modified to be identical to a function (i.e., **y**′) in pools. We ensure that, prior to the modification, the similarity score between **x** and its compilation variant $\widetilde{\mathbf{x}}$ is the highest within the pool. After the CFG modification, we compute similarity scores between **x**′ and all functions in the pool (including $\widetilde{\mathbf{x}}$), and rank all functions based on these scores. If the similarity score between **x**′ and $\widetilde{\mathbf{x}}$ no longer holds the top rank, we regard it as an error. Then, we calculate the ER in the same manner as described in Section 5.1. (3) We further underscore the extent to which identical CFGs can contribute to misjudging functions as similar. We measure the proportion of pairs that, when modified to possess identical CFGs, achieve the top-1 position in terms of similarity scores within their respective pools. To this end, we introduce the metric *top-1 rate*: $\frac{num(top\text{-}1)}{num(all)}$. Here, $num(top\text{-}1)$ represents the count of pairs for which the similarity score of **x**′ and **y**′ ranks first within the pool. Meanwhile, $num(all)$ signifies the total number of function pairs investigated.

Results. Figure 6 presents the results of increase ratios. For the majority of solutions, manipulating CFGs to be identical leads to significant increases in similarity scores for function pairs with different semantics. Specifically, over 56.9% of function pairs (100% − 43.1% on OrderMatters) register increase ratios above 40%, with at least 42.4% surpassing 60%. Table 4 further underscores a major shortcoming in ML-BFSD solutions when evaluating function pairs with identical CFGs. These models frequently misidentify them as similar, a clear discrepancy from their true different semantics. Remarkably, the ER for a bulk of solutions breaches the 50% mark at a modest pool size of 16, and this exceeds 70% when the pool size extends to 128. Moreover, Table 5 shows that considerable function pairs, upon becoming identical CFGs, achieve the top-1 similarity scores within their respective function pools. On most solutions, the top-1 rate remains above 20% even as the pool size increases to 128.

Summary. Except for SAFE, Trex and jTrans that do not rely on CFGs, other solutions show pronounced increase ratios, ER, and top-1 rate. Such results illuminate the stark challenges these models grapple with when tasked with recognizing semantic dissimilarities amidst identical CFGs, validating the pitfalls of over-reliance on CFGs.

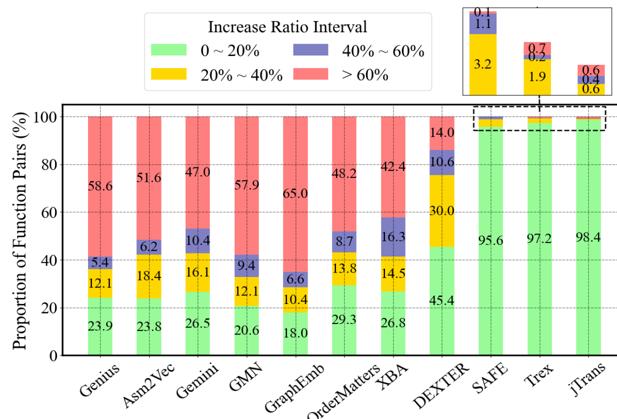

Figure 6: Evaluation results of increase ratios.

Table 4: Evaluation of ER when CFGs become identical.

BFSD Solutions	ER (%)			
	pool size = 16	pool size = 32	pool size = 64	pool size = 128
Genius	72.1	82.1	85.3	88.4
Asm2Vec	51.0	53.5	56.1	59.1
Gemini	52.5	58.6	63.1	71.2
GMN	50.8	61.6	67.1	71.9
GraphEmb	43.5	49.8	55.2	62.2
OrderMatters	69.2	74.3	78.1	81.3
XBA	59.8	69.1	76.5	79.6
DEXTER	58.9	64.3	68.6	74.0
SAFE	2.0	3.2	3.4	4.4
Trex	1.6	2.1	2.4	3.1
jTrans	1.5	1.7	2.6	3.1

5.3 Interpreting CFG Over-reliance

After validating ML-BFSD solutions heavily relying on CFG features, we conclude two potential reasons that cause the

Table 5: Evaluation results of top-1 rate.

BFSD Solutions	Top-1 rate (%)			
	pool size = 16	pool size = 32	pool size = 64	pool size = 128
Genius	39.1	35.8	30.5	25.6
Asm2Vec	29.2	27.5	23.2	20.8
Gemini	37.4	34.8	31.6	28.6
GMN	38.5	34.6	33.5	31.7
GraphEmb	24.5	23.7	22.2	21.0
OrderMatters	36.2	31.3	26.4	22.0
XBA	23.2	20.7	18.8	18.0
DEXTER	34.6	29.2	24.0	21.7
SAFE	5.5	4.2	3.5	2.3
Trex	4.6	2.3	1.9	1.7
jTrans	2.0	1.9	1.6	1.5

over-reliance: (1) Many ML-BFSD solutions suffer from design flaws that compromise the learning of semantics. (2) The training set has a bias. Based on the above analysis, we clarify the reasoning between semantics and CFG features.

Design flaw analysis. We find that these ML-BFSD solutions have design flaws that undermine the learning of semantics, resulting in the over-reliance on CFG features.

First, some methods [16, 50, 67] manually specify features, potentially resulting in information loss. For instance, both Gemini and DEXTER neglect the order of instructions when defining manual features, which leads to information loss. Second, instruction relationships are not well captured by many ML-BFSD solutions. Genius learns intra-block semantics but not inter-block relations. Asm2Vec only partially learns relationships through a random walk, which covers a small number of basic blocks. In ML-BFSD solutions utilizing GNNs [33, 38, 46, 67, 72] , GNNs assign instructions to various graph nodes based on control dependencies depicted by CFGs, where each node corresponds to a basic block. However, GNNs typically excel at learning relationships between instructions within basic blocks that are directly connected. For instructions located in distant blocks that are not directly connected, it is generally less effective for GNNs to capture relationships between them. This implies that learning based on CFGs could limit models' understanding of instruction context. We note that Pei et al. [51] explore the potential of utilizing large language models (LLMs) for representation. Given their exceptional representation capabilities, using LLMs for representation may become a future trend.

Bias of training set. We note many solutions use a biased training set, resulting in over-reliance on CFGs. These solutions aim to train models to identify function pairs with identical semantics as similar (denoted as $\langle \mathbf{x}, \widetilde{\mathbf{x}} \rangle$) and function pairs with different semantics as dissimilar (denoted as $\langle \mathbf{x}, \mathbf{y} \rangle$). Here $\mathbf{x}$ and $\mathbf{y}$ are semantically distinct functions taken from the function set constructed by the ML-BFSD solutions, and $\widetilde{\mathbf{x}}$ is a compilation variant of $\mathbf{x}$. However, the distribution of function pairs has a bias, and thus causes models to heavily rely on CFGs. Specifically, function pairs can be generally categorized into four types. (1) Type 1: different CFGs and different semantics. (2) Type 2: different CFGs but same semantics. (3) Type 3: same CFGs and same semantics. (4) Type 4: same CFGs but different semantics. We find that numerous pairs belong to types 1-3, while only a small amount of them belong to type 4. Note that, only function pairs in types 3 and 4 have the same CFGs. However, type 3 is much more common than type 4, which can cause models to tend to identify functions with the same CFGs as similar, namely, over-reliance on CFG features.

Table 6: The proportion of four types of function pairs.

Repetition Count	Proportion (%)			
	Type 1	Type 2	Type 3	Type 4
Repetition #1	49.71	40.49	9.51	0.29
Repetition #2	49.73	42.46	7.54	0.27
Repetition #3	49.75	39.39	10.61	0.25
Repetition #4	49.72	40.31	9.69	0.28
Repetition #5	49.68	39.33	10.67	0.32

Specifically, these solutions use the BinaryCorp-3M training set from jTrans [64], which compiles tens of thousands of open-source projects and establishes this uniform and large-scale dataset containing approximately 3,000,000 functions. We randomly select 400,000 function pairs from the training set, and observe the proportion of the four types of function pairs. We repeat each experiment five times. Table 6 presents the proportion of the four types of function pairs, and the proportion of type 4 is much lower than that of others. Only function pairs in types 3 and 4 have identical CFGs, but type 3 is $27.9 \sim 42.4\times$ more than type 4, which leads to overfitting and over-reliance on CFGs. That is, models are prone to identify function pairs with the same CFGs as semantically equivalent.

Reasoning between semantics and CFG features. Semantics are determined by both instructions themselves and their contexts, i.e., relationships between instructions. Thus, ML-BFSD solutions need to learn not only the instructions but also their contexts. However, CFGs can only aid models in learning a portion of (not all) the context of instructions and may even limit the model's learning of instruction context. On the other hand, modeling the full context, i.e., an instruction sequence, may yield different embeddings by different ML-BFSD solutions. But this is not an issue. Their embeddings differ because they reside in different latent spaces. These differing latent spaces can be aligned through projection.

5.4 Model Performance Improvement

After analyzing the reasons behind the over-reliance on CFGs, we find δ`CFG` can be used to mitigate this issue by alleviating training set bias. Consequently, δ`CFG` helps ML-BFSD solutions better learn function semantics, improving model performance. We use δ`CFG` to manipulate function CFGs, thus increasing the number of function pairs with identical CFGs but different semantics. Subsequently, we use the manipulated functions generated by δ`CFG`, as well as the original training

Table 7: Comparison of improvement in MRR after fine-tuning with and without augmented data (denoted as **clean**), expressed as Δ MRR. The results validate the effectiveness of using δCFG for fine-tuning the models.

BFSD Solutions	Δ MRR (%)													
	O0,O3		O1,O3		O2,O3		O0,Os		O1,Os		O2,Os		Average	
	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean
Gemini	**9.0**	0.0	**3.5**	0.2	**1.6**	0.2	**7.0**	0.2	**5.5**	-0.1	**4.8**	0.0	**5.2**	0.1
GMN	**10.1**	0.3	**5.0**	0.2	**1.4**	0.2	**9.8**	0.1	**4.0**	-0.1	**4.1**	0.3	**5.7**	0.2
GraphEmb	**3.7**	-0.1	**2.8**	0.2	**1.0**	0.3	**3.6**	-0.1	**2.9**	0.1	**3.1**	0.1	**2.9**	0.1
OrderMatters	**1.4**	0.1	**1.3**	-0.1	**0.9**	0.1	**3.1**	0.1	**1.3**	-0.1	**1.8**	0.1	**1.6**	0.1
XBA	**0.4**	0.1	**0.3**	-0.1	**0.2**	0.1	**1.1**	0.1	**0.8**	0.1	**0.4**	0.1	**0.5**	0.1
DEXTER	**1.4**	-0.2	**4.7**	-0.1	**1.9**	0.2	**2.0**	0.3	**4.5**	0.1	**1.4**	0.3	**2.7**	0.1

Table 8: Comparison of improvement in Recall@1 after fine-tuning with and without augmented data (denoted as **clean**), expressed as Δ Recall@1. The results validate the effectiveness of δCFG.

BFSD Solutions	Δ Recall@1 (%)													
	O0,O3		O1,O3		O2,O3		O0,Os		O1,Os		O2,Os		Average	
	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean
Gemini	**10.8**	-0.1	**4.4**	0.2	**1.9**	0.1	**9.3**	0.0	**6.7**	0.1	**5.6**	0.1	**6.5**	0.1
GMN	**12.7**	0.3	**7.1**	-0.1	**1.8**	0.2	**12.7**	0.1	**6.2**	0.3	**5.9**	0.1	**7.7**	0.2
GraphEmb	**5.2**	0.1	**4.3**	0.2	**1.5**	0.3	**5.4**	0.2	**4.2**	0.3	**4.3**	0.2	**4.2**	0.3
OrderMatters	**1.8**	0.0	**1.6**	-0.1	**1.2**	0.1	**3.5**	0.2	**1.7**	0.1	**2.5**	0.0	**2.1**	0.1
XBA	**0.6**	0.0	**0.4**	-0.1	**0.3**	0.1	**1.6**	0.1	**1.3**	0.2	**0.7**	0.0	**0.8**	0.1
DEXTER	**1.1**	-0.3	**7.0**	0.2	**3.4**	-0.2	**3.3**	0.3	**7.1**	0.3	**1.8**	0.3	**3.9**	0.1

Table 9: Comparison of improvement in AUC after fine-tuning with and without augmented data (denoted as **clean**), expressed as Δ AUC.

BFSD Solutions	Δ AUC (%)													
	O0,O3		O1,O3		O2,O3		O0,Os		O1,Os		O2,Os		Average	
	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean
Gemini	**1.0**	0.0	**0.6**	0.1	**0.4**	0.0	**0.5**	0.0	**0.5**	0.0	**0.4**	-0.1	**0.6**	0.0
GMN	**5.1**	0.0	**3.7**	-0.1	**2.8**	0.0	**2.9**	0.0	**2.7**	0.0	**2.5**	0.0	**3.3**	0.0
GraphEmb	**3.8**	0.0	**2.4**	0.0	**1.6**	0.0	**1.8**	0.0	**1.6**	-0.1	**1.7**	0.0	**2.2**	0.0
OrderMatters	**0.4**	0.0	**0.5**	0.1	**0.5**	0.1	**0.6**	-0.1	**0.4**	0.0	**0.6**	0.0	**0.5**	0.0
XBA	**0.2**	0.0	**0.2**	0.0	**0.1**	-0.1	**0.3**	0.0	**0.2**	0.0	**0.1**	-0.1	**0.2**	0.0
DEXTER	**0.4**	0.0	**0.7**	0.1	**0.1**	0.0	**0.6**	0.0	**0.3**	0.0	**0.3**	0.0	**0.4**	0.0

Table 10: Comparison of improvement in F1 score after fine-tuning with and without augmented data (denoted as **clean**), expressed as Δ F1 score.

BFSD Solutions	Δ F1 Score (%)													
	O0,O3		O1,O3		O2,O3		O0,Os		O1,Os		O2,Os		Average	
	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean	δCFG	clean
Gemini	**5.5**	0.0	**0.1**	0.0	**0.2**	-0.1	**5.6**	0.1	**0.3**	0.0	**0.2**	0.1	**2.0**	0.0
GMN	**27.2**	0.0	**10.8**	-0.1	**2.0**	0.0	**18.6**	0.0	**11.3**	0.0	**11.0**	0.0	**13.5**	0.0
GraphEmb	**3.7**	0.1	**5.0**	0.1	**1.1**	0.0	**4.0**	0.1	**5.1**	0.0	**4.7**	0.0	**3.9**	0.1
OrderMatters	**16.9**	0.1	**5.1**	0.0	**0.6**	0.0	**15.7**	0.0	**5.9**	-0.1	**5.0**	0.0	**8.2**	0.0
XBA	**0.2**	0.0	**0.2**	0.0	**0.1**	0.0	**0.2**	0.0	**0.1**	0.0	**0.2**	0.0	**0.2**	0.0
DEXTER	**14.9**	0.2	**8.9**	0.0	**1.2**	-0.1	**8.0**	-0.2	**8.5**	0.0	**4.0**	0.2	**7.6**	0.0

set to fine-tune ML-BFSD solutions, enhancing ML-BFSD solutions.

We use the training set from BinaryCorp-3M, and randomly select 50,000 function pairs exhibiting differing semantics and employ δCFG to generate manipulated pairs. Then, we curate a fine-tuning dataset by integrating these manipulated pairs (50,000) into the training set. Using this enhanced set, we fine-tune the top six performing solutions with 5 epochs: Gemini, GMN, GraphEmb, OrderMatters, XBA and DEXTER. Finally, we evaluate model performance on the test set (no augmented data) from BinaryCorp-3M, with pool size of 32.

Performance and semantics enhancement with δCFG.

Table 11: Change in importance scores: increase for semantic features and decrease for CFG features in enhanced models.

Explanation Method	BFSD Solutions	Δ Average Score							
		Semantic Features					CFG Features		
		Call	Jump	Arith	Data-Tran	Other	Nodes	Edges	Graph-Sim
LIME	Gemini	0.008	0.007	0.006	0.004	0.011	-0.005	-0.006	-0.083
	GMN	0.041	0.049	0.032	0.038	0.047	-0.031	-0.008	-0.174
	GraphEmb	0.022	0.019	0.021	0.026	0.009	-0.017	-0.003	-0.118
	OrderMatters	0.014	0.007	0.002	0.005	0.008	-0.011	-0.008	-0.024
	XBA	0.012	0.006	0.004	0.006	0.007	-0.008	-0.013	-0.035
	DEXTER	0.016	0.005	0.008	0.009	0.017	-0.012	-0.010	-0.066
LEMNA	Gemini	0.011	0.010	0.008	0.004	0.003	-0.009	-0.001	-0.041
	GMN	0.027	0.028	0.005	0.007	0.022	-0.018	-0.025	-0.096
	GraphEmb	0.006	0.003	0.004	0.008	0.012	-0.005	-0.004	-0.019
	OrderMatters	0.016	0.004	0.013	0.006	0.015	-0.011	-0.043	-0.093
	XBA	0.015	0.003	0.012	0.004	0.012	-0.013	-0.026	-0.088
	DEXTER	0.009	0.002	0.006	0.007	0.013	-0.008	-0.004	-0.012

We assess model performance using four metrics commonly employed in previous literature [45,46,64]: MRR, Recall@1, AUC, and F1 score. Table 7, 8, 9 and 10 highlight performance enhancements across all models upon applying δ`CFG`, with the highest observed boosts of 10.1% in MRR, 12.7% in Recall@1, 5.1% in AUC and 27.2% in F1 score. On average, MRR, Recall@1, AUC and F1 score experience an uplift of 0.5% $\sim$ 5.7%, 0.8% $\sim$ 7.7%, 0.2% $\sim$ 3.3%, and 0.2% $\sim$ 13.5% respectively. Conversely, models fine-tuned solely with the BinaryCorp-3M training set (denoted as **clean**) often stagnate and even regress in performance. Further, we use our `Explainer` to explain the enhanced models, revealing that the importance of semantic features is improved. According to Table 11, the enhanced models exhibit increased importance scores for semantic features and reduced scores for CFG features, signifying a better understanding of function semantics and reduced over-reliance on CFG features.

Robustness evaluation. The above results validate the enhancement of models through the application of δ`CFG`. Such enhancement enables models to generate new function representations. To evaluate the robustness of new representations, we increase instruction separation by augmenting the number of basic blocks between instructions. This methodology allows us to assess models' robustness to such changes and validate their improved capability in learning relationships between instructions in distinct basic blocks.

We employ the same set of function pairs as declared in Section 5.1, each consisting of semantically equivalent functions with identical CFGs, denoted as **<x, x$_2$>**. For each pair, we randomly select k basic blocks within **x** to split using LLVM's function `splitBasicBlock`, with k randomly chosen from the set {1,2}. We then reconnect the divided basic blocks with direct jump instructions to maintain the original control flow. After the split, we introduce a stochastic element by randomly determining whether to incorporate additional branches between the newly formed basic blocks. For example, if a basic block b is split into b_1 and b_2, with a direct transition from b_1 to b_2, we randomly decide whether to introduce a new successor node b_3 to b_1 and switch the direct jump to a conditional jump. These modifications effectively increase instruction separation.

We compare the performance of models enhanced by δ`CFG` against the original models without enhancements (denoted as **baseline**). For comparisons, we use the decrease ratio and ER metrics as defined in Section 5.1. Lower values in both metrics indicate greater robustness, affirming models' superior ability to learn relationships between instructions.

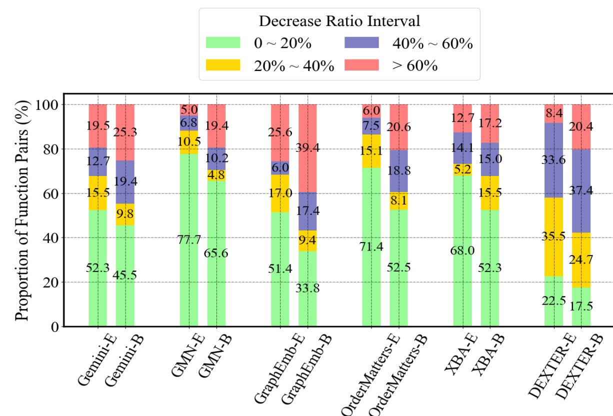

Figure 7: Comparison of decrease ratios between enhanced (-E) and baseline (-B) models.

Figure 7 and Table 12 show that function pairs analyzed using enhanced models exhibit lower decrease ratios and ER compared to those analyzed with baseline models. Such results validate the improved robustness of enhanced models. In Figure 7, a greater proportion of function pairs evaluated using enhanced models maintain decrease ratios within the 0 $\sim$ 20% interval. In contrast, function pairs analyzed with baseline models more frequently fall into higher decrease ratio intervals, namely 40% $\sim$ 60% and above 60%. For instance, in GMN-E (enhanced GMN), only 11.8% function pairs show

Table 12: Comparison of ER between enhanced and baseline models.

BFSD Solutions	ER (%)							
	pool size = 16		pool size = 32		pool size = 64		pool size = 128	
	δCFG	baseline	δCFG	baseline	δCFG	baseline	δCFG	baseline
Gemini	**30.6**	36.0	**36.8**	42.8	**40.9**	47.4	**46.1**	51.6
GMN	**23.0**	33.5	**26.4**	37.3	**29.2**	41.7	**32.8**	46.1
GraphEmb	**23.2**	47.1	**28.3**	51.4	**33.7**	54.6	**38.7**	57.3
OrderMatters	**13.5**	29.6	**18.6**	35.5	**24.2**	43.3	**28.8**	50.5
XBA	**47.6**	51.5	**53.9**	57.4	**58.8**	61.5	**62.0**	65.2
DEXTER	**40.4**	49.3	**43.2**	55.5	**47.9**	60.1	**53.1**	62.8

decrease ratios exceeding 40%. In comparison, GMN-B (the baseline) results in 29.6% function pairs experiencing decrease ratios above 40%. This indicates greater variability in similarity scores when computed using the baseline model. In Table 12, across all function pool sizes , enhanced models consistently achieve lower ER. Overall, the above results indicate that enhanced models are less affected by increased instruction separation, thereby affirming their better robustness and improved ability to discern instruction relationships.

6 Conclusion

This work is the first to investigate the role of CFG features in ML-based BFSD, serving as a robustness validator for model developers. We first designed `Explainer` to explain 11 representative ML-BFSD solutions and reveal that CFG features are overly relied on. Then, we designed δ`CFG` to explore the impact of CFG features on model decisions, and discovered numerous errors in ML-BFSD solutions by manipulating function CFGs. We further analyzed the reasons behind such over-reliance on CFG features, and discovered design flaws in solutions, as well as a serious bias in training data. In addition, we found that ML-BFSD solutions could also be used to enhance model performance. We conducted fine-tuning with δ`CFG` on all ML-BFSD solutions studied by us, significantly improving model performance as well as de-prioritizing the over-reliance issue.

Acknowledgments

This work was supported in part by China Key R&D Program 2021YFB2701000 and 2022YFB2901300; by Quancheng Lab QCLZD202304 & SYS202201; by National Natural Science Foundation of China 61972224; by Beijing Science Foundation IS23057; by China Unicom Guangdong; and by NSFOCUS 20222910019.

References

[1] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. *FOSSIL*: A resilient and efficient system for identifying FOSS functions in malware binaries. *ACM Trans. Priv. Secur.*, 2018.

[2] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the Network and Distributed System Security Symposium,*, 2009.

[3] Silvio Cesare, Yang Xiang, and Wanlei Zhou. Control flow-based malware variantdetection. *IEEE Transactions on Dependable and Secure Computing*, 2013.

[4] Chun-Hao Chang, Elliot Creager, Anna Goldenberg, and David Duvenaud. Explaining image classifiers by counterfactual generation. In *International Conference on Learning Representations, ICLR*, 2019.

[5] Christian Collberg. The tigress c obfuscator. https://tigress.wtf/introduction.html. Accessed: 2023-06-08.

[6] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, 2016.

[7] Amit Dhurandhar, Karthikeyan Natesan Ramamurthy, and Karthikeyan Shanmugam. Is this the right neighborhood? accurate and query efficient model agnostic explanations. In *Advances in Neural Information Processing Systems*, 2022.

[8] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *IEEE Symposium on Security and Privacy, SP*, 2019.

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

[10] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[11] Gabriel G. Erion, Joseph D. Janizek, Pascal Sturmfels, Scott M. Lundberg, and Su-In Lee. Improving performance of deep learning models with axiomatic attribution priors and expected gradients. *Nat. Mach. Intell.*, 2021.

[12] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *Network and Distributed System Security Symposium*, 2016.

[13] Ming Fan, Wenying Wei, Xiaofei Xie, Yang Liu, Xiaohong Guan, and Ting Liu. Can we trust your explanations? sanity checks for interpreters in android malware analysis. *IEEE Trans. Inf. Forensics Secur.*, 2021.

[14] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *Eighth International Conference on Software Security and Reliability*, 2014.

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

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

[17] Ruth Fong, Mandela Patrick, and Andrea Vedaldi. Understanding deep networks via extremal perturbations and smooth masks. In *IEEE/CVF International Conference on Computer Vision, ICCV*, 2019.

[18] Ruth C. Fong and Andrea Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. In *IEEE International Conference on Computer Vision*, 2017.

[19] Martin Fowler and Kent Beck. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.

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

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

[22] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. LEMNA: explaining deep learning based security applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2018.

[23] Yixin Guo, Pengcheng Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. Exploring GNN based program embedding technologies for binary related tasks. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC*. ACM, 2022.

[24] Dongqi Han, Zhiliang Wang, Wenqi Chen, Ying Zhong, Su Wang, Han Zhang, Jiahai Yang, Xingang Shi, and Xia Yin. Deepaid: Interpreting and improving deep learning-based anomaly detection in security applications. In *ACM SIGSAC Conference on Computer and Communications Security*, 2021.

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

[26] Jerome Dinal Herath, Priti Prabhakar Wakodikar, Ping Yang, and Guanhua Yan. Cfgexplainer: Explaining graph neural network-based malware classification from control flow graphs. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*. IEEE, 2022.

[27] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Cross-architecture binary semantics understanding via similar code comparison. In *IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016.

[28] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Binary code clone detection across architectures and compiling configurations. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC*. IEEE Computer Society, 2017.

[29] He Huang, Amr M Youssef, and Mourad Debbabi. Binsequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, 2017.

[30] Jiyong Jang, Maverick Woo, and David Brumley. Towards automatic software lineage inference. In *Proceedings of the 22th USENIX Security Symposium*, 2013.

[31] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection*, 2015.

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

[33] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. Improving cross-platform binary analysis using representation learning via graph alignment. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2022.

[34] Nils M Kriege, Pierre-Louis Giscard, and Richard Wilson. On valid optimal assignment kernels and applications to graph classification. *Advances in neural information processing systems*, 2016.

[35] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization.*, 2004.

[36] Quoc V. Le and Tomás Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML*, 2014.

[37] Xuezixiang Li, Yu Qu, and Heng Yin. Palmtree: Learning an assembly language model for instruction embedding. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2021.

[38] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *Proceedings of the 36th International Conference on Machine Learning, ICML*, 2019.

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

[40] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. Deep learning for android malware defenses: a systematic literature review. *ACM Journal of the ACM*, 2022.

[41] Ana Lucic, Maartje A. ter Hoeve, Gabriele Tolomei, Maarten de Rijke, and Fabrizio Silvestri. Cf-gnnexplainer: Counterfactual explanations for graph neural networks. In *International Conference on Artificial Intelligence and Statistics*, 2022.

[42] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. Parameterized explainer for graph neural network. In *Proceedings of the International Conference on Neural Information Processing Systems*, 2020.

[43] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

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

[45] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *USENIX Security Symposium*, 2022.

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

[47] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. SAFE: self-attentive function embeddings for binary similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment International Conference*, 2019.

[48] Jiang Ming, Meng Pan, and Debin Gao. ibinhunt: Binary hunting with inter-procedural control flow. In *International Conference on Information Security and Cryptology*, pages 92–109. Springer, 2012.

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

[50] J. Patrick-Evans, M. Dannehl, and J. Kinder. Xfl: Naming functions in binaries with extreme multi-label learning. In *IEEE Symposium on Security and Privacy (SP)*, 2023.

[51] Kexin Pei, Weichen Li, Qirui Jin, Shuyang Liu, Scott Geng, Lorenzo Cavallaro, Junfeng Yang, and Suman Jana. Exploiting code symmetries for learning program semantics, 2024.

[52] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. In *IEEE Transactions on Software Engineering*, 2022.

[53] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *IEEE Symposium on Security and Privacy*, 2015.

[54] Gregory Plumb, Denali Molitor, and Ameet Talwalkar. Model agnostic supervised local explanations. In *Advances in neural information processing systems*, 2018.

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

[56] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[57] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-precision model-agnostic explanations. In *Proceedings of the AAAI conference on artificial intelligence*, 2018.

[58] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009.

[59] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *IEEE International Conference on Computer Vision, ICCV*, 2017.

[60] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML*, 2017.

[61] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 2014.

[62] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. Clap: Learning transferable binary code representations with natural language supervision, 2024.

[63] Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xi Xiao. Cebin: A cost-effective framework for large-scale binary code similarity detection, 2024.

[64] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. jtrans: jump-aware transformer for binary code similarity detection. In *International Symposium on Software Testing and Analysis*, 2022.

[65] Dana Warmsley, Alex Waagen, Jiejun Xu, Zhining Liu, and Hanghang Tong. A survey of explainable graph neural networks for cyber malware analysis. In *IEEE International Conference on Big Data*, 2022.

[66] Fengliang Xia, Guixing Wu, Guochao Zhao, and Xiangyu Li. Simcge: Simple contrastive learning of graph embeddings for cross-version binary code similarity detection. In Cristina Alcaraz, Liqun Chen, Shujun Li, and Pierangela Samarati, editors, *Information and Communications Security - 24th International Conference, ICICS*. Springer, 2022.

[67] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.

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

[69] Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. Codee: A tensor embedding scheme for binary code search. *IEEE Transactions on Software Engineering*, 2021.

[70] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. CADE: detecting and explaining concept drift samples for security applications. In *USENIX Security Symposium*, 2021.

[71] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. In *Advances in neural information processing systems*, 2019.

[72] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI*, 2020.

[73] Xingyu Zhao, Wei Huang, Xiaowei Huang, Valentin Robu, and David Flynn. Baylime: Bayesian local interpretable model-agnostic explanations. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, 2021.

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