Pretraining on Call Graphs: When Binary Analysis Tasks Profit From Context

Samuel Valenzuela
Ludwig-Maximilians-Universität München
Munich Center for Machine Learning
Center for Digital Technology and Management
Munich, Germany
samuel.valenzuela@lmu.de

Johannes Kinder
Ludwig-Maximilians-Universität München
Munich Center for Machine Learning
Munich, Germany
johannes.kinder@lmu.de

Abstract

Binary function embedding models are trained to encode the semantics of binary code in such a way that they can be generalized to a variety of reverse engineering tasks, such as binary code search, vulnerability detection, or malware classification. While many models only take the function in question as contextual input, there have been successful attempts to improve function embeddings by leveraging information from the call graph. In this study, we dissect the implications of these embedding refinements. We conduct experiments using a range of graph-based models on the embeddings generated by two state-of-the-art binary function embedding models. Integrating inter-procedural context, we show that improvements on binary code similarity detection (BCSD) will not necessarily generalize to downstream tasks, neither of semantic nor of syntactic nature. More generally, we find that optimizing for semantic similarity tasks correlates with worse performance on syntactic tasks. By conducting an explanatory analysis on the dataset, we find that the call graph-based enhancements significantly enhance the robustness of embeddings, particularly in scenarios where the initial models struggle. Furthermore, we observe that the added context is more beneficial for namespace-related functions than for those focused on individual logic, confirming that the call graph can be leveraged most effectively in context-dependent scenarios.

CCS Concepts

• **Security and privacy → Software reverse engineering**; • **Computing methodologies → Neural networks**.

Keywords

Binary analysis, Binary code similarity detection, Call graphs, Explainability, Function embeddings, Graph neural networks

ACM Reference Format:
Samuel Valenzuela and Johannes Kinder. 2026. Pretraining on Call Graphs: When Binary Analysis Tasks Profit From Context. In *34th IEEE/ACM International Conference on Program Comprehension (ICPC '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3794763.3794795

ICPC '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *34th IEEE/ACM International Conference on Program Comprehension (ICPC '26), April 12–13, 2026, Rio de Janeiro, Brazil*, https://doi.org/10.1145/3794763.3794795.

1 Introduction

Binary code analysis is a subfield of program analysis dealing with situations in which the original source code of a compiled program is not available to the reverse engineers. Applications range from vulnerability detection [11, 22] and malware classification [30] to more general reverse engineering tasks such as binary code similarity detection (BCSD) [21, 23, 35, 36, 39, 41] and function labeling [4, 8, 14, 17, 28]. Additionally, syntactic binary analysis tasks such as compiler provenance recovery [31]—i.e., identifying aspects including the compiler version or optimization level—help in understanding and working with the code [16].

A particularly challenging aspect of low-level code comprehension is that two pieces of binary code can differ greatly in their structure and instruction sequences, yet implement the same functionality and thus be semantically equivalent. This also applies to binary programs compiled from the same source when using different compiler options or optimization levels. Crucially, binaries are typically stripped of debug information like variable names that may offer helpful clues regarding the code semantics.

In light of these intricacies, deep learning approaches have gained wide adoption for binary analysis tasks in recent years to help reverse engineers navigate machine code more effectively. As disassembled binary code can be read in a similar manner to sequential text, many approaches are inspired by advances in the NLP domain and make use of the way code is structured. For instance, distant instructions are oftentimes closely related through jumps or function calls. Different approaches have been adopted to model these relationships. While recent Transformer-based models capture control flow by sharing parameters between embeddings [35, 36], other methods employ graph-based models on different program representations such as the control or data flow graph [11, 22, 39, 41].

Tuning graph nodes on their surrounding context has proven to be beneficial not only on an instruction level, but also on a function level [12]. Particularly given that the Transformer-based state of the art does not scale well to incorporate the entire program context, architectures have been emerging which extract information from the call graphs for different types of tasks including BCSD [12, 21, 37, 40], function labeling [17, 40], and compiler provenance recovery [12, 16]. Taking BCSD as an example, the intuition behind leveraging the call graphs to improve the quality of binary function embeddings is that the embeddings of a function f and a function g should approach each other if g calls function h and f is semantically equivalent to g but inlines h. Given that the average function in our dataset calls 4.4 distinct internal functions (i.e., excluding external library calls) the number of considered instructions is almost tripled

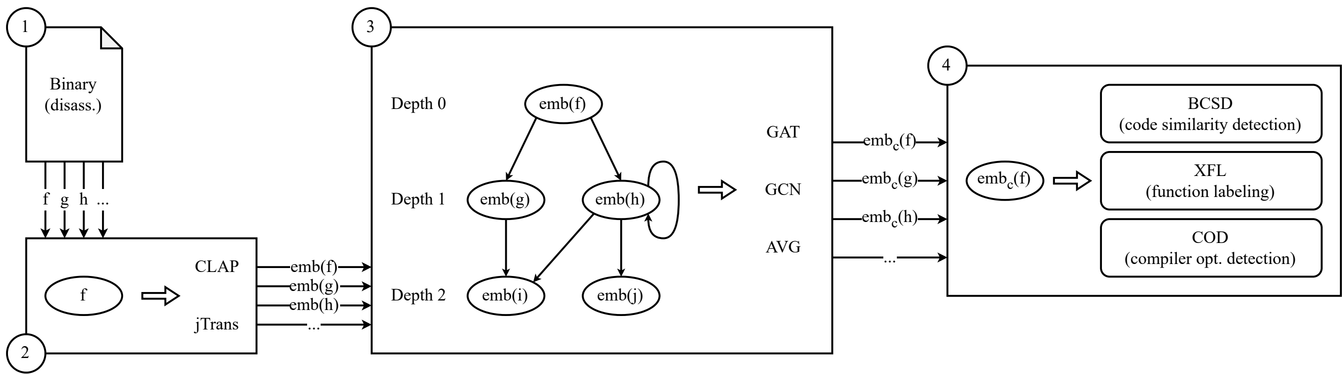

Figure 1: Overview of our workflow. First, a binary is disassembled and all functions are extracted. These are then processed by a binary function embedding model. The resulting embeddings for each function populate the nodes of the call graph and are updated using a graph-based model. The context-enhanced embeddings are then used as an alternative input to different machine learning tasks.

from 84.3 to 239.0 in a single step and highlights why models can benefit from this additional context.

While the aforementioned approaches successfully incorporate call graphs into the model training for various tasks, little attention has been paid to the application of the resulting embeddings to downstream tasks without task-specific embedding refinements. More generally, there is no literature to our knowledge that systematically examines how such semantic tasks affect the model's retention of technical details and as such how generalizable the model is not only to other semantic tasks, but also to syntactic tasks such as compiler provenance recovery. Furthermore, this line of work has mainly shown *that* the consideration of call graphs improves model performance, but there are no comprehensive studies on *how* or for which types of functions this approach is especially fruitful.

In this paper, we explore how call graphs can be leveraged in the pretraining stage using BCSD. Inspired by related work [12, 37], we use two variants of graph neural networks (GNNs) in order to leverage information from the call graph. The goal is to intervene in cases that previously viewed internal function calls as unknown `call` instructions, and to incorporate their function embeddings as meaningful semantic input for the embedding of the caller function. In our evaluation, we compare how the performance of both GNNs and a simple averaging baseline develops across varying degrees of inter-procedural context (call depths). Beyond the BCSD task, we extend the application of the context-enhanced embeddings to a semantic function labeling task as well as a syntactic compiler provenance task. Moreover, we conduct an explanatory analysis to uncover in which cases the embedding refinements prove to be most beneficial. For this, we split the dataset into various dimensions and compare how well the models perform for the different function groups. Thereby, we make the following contributions:

(1) We show that adding a pretraining task on the call graph can improve embeddings for BCSD, but worsen their generalizability to other semantic downstream tasks like function labeling.
(2) We show that training and better performance on semantic similarity tasks correlates with a decreased performance on syntactic tasks.
(3) We show that the contextual additions for BCSD make the embeddings more robust to larger context sizes, diversity within function pairs, and technical implementations.
(4) We identify that the enhanced embeddings improve more for namespace-related functions than for other functions.

Our code and data is available online[1].

2 Methodology

Figure 1 provides a general overview of our workflow, which consists of four main steps:

(1) We disassemble an input binary and extract all functions as well as the call graph, a directed graph where each node corresponds to an internal function and each edge denotes a caller-callee relationship.
(2) We generate an embedding for each function using a pretrained model which we refer to as the backbone.
(3) After initializing its nodes with the corresponding functions' pretrained embeddings, the call graph is processed by a model which updates each node's embedding by aggregating its own features with those of its callees within a specified call depth.
(4) The resulting context-enhanced embeddings are then used for various function-level machine learning tasks as outlined in the following sections.

We use IDA Pro [1] for disassembly and generate the initial function embeddings with CLAP [35] and jTrans [36] as backbones. Both are state-of-the-art Transformer-based models that also base their preprocessing pipeline on IDA Pro. Besides implementation-specific differences that provide CLAP with more information obtained by IDA Pro, the key difference lies in their training method. Initially, both models are trained via masked language modeling, a common pretraining task known from the NLP domain. However, after

[1] https://github.com/lmu-plai/binary-callgraph-pretraining

that, CLAP is trained using natural language supervision to align the binary code embeddings with explanations of the source code, whereas jTrans attempts to predict the targets of jump instructions.

For creating the context-enhanced function embeddings, we compare a simple averaging baseline and two prominent GNN variants: The Graph Convolutional Network (GCN) [19] and the Graph Attention Network (GAT) [34] are two GNNs that employ layer-specific weights, effectively distinguishing between different call depths. The model architectures differ in the way they aggregate information from neighbors. GCNs apply predefined weights at inference time, whereas GATs leverage an attention mechanism and thus can learn to evaluate the importance of neighbors—or in our case, callee functions. For each of these three model types, we consider a call depth from 1 to 5, resulting in 15 models per backbone. This means that, for a call depth of n, we extract a subgraph containing n steps of node successors to the function in question, and train a GNN with n layers.

All of our experiments are conducted on a dataset derived from BinaryCorp-26M [36], a large set of binaries compiled with different optimization settings which both CLAP and jTrans were trained on. The complete dataset of binaries, including all programs and functions referenced later in this paper, is available online[2]. After stripping each binary to remove all debugging symbols, we disassemble and extract all functions. Next, we apply two filtering steps. First, we only consider functions with at least four instructions to reduce the noise caused by trivial functions such as empty functions, basic wrappers, or compiler-generated PLT stubs. To facilitate the training process described below, we only keep one function pair per source function if compilation with different optimization configurations results in binary code with differing opcode hashes. This way, we avoid function pairs that only differ in register allocations or the binary's address layouts. The resulting dataset consists of approximately 1.4M function pairs from 34.3k binaries compiled from 9.4k source programs. We retain the train-test split of BinaryCorp-26M and finetune hyperparameters on a secluded part of the train set.

3 Tasks

We train the GNNs on the BCSD task, ensuring a comparable approach to related work on cross-configuration binary code detection by Guo et al. [12]. After training, we evaluate the generalizability of the refined function embeddings via two downstream tasks, namely eXtreme Function Labeling (XFL) [28] and Compiler Optimization Detection (COD).

3.1 Binary Code Similarity Detection

In BCSD, the goal is to measure the similarity between two instances of binary code. All GNNs are trained on BCSD using the contrastive InfoNCE loss [27], specifically the NT-Xent variant [5] as implemented by Silva [32], which rewards the similarity of functions compiled from the same source function and penalizes the similarity between other function pairs. Per batch of positive function pairs, the remaining functions in the batch are used as negative samples for each function. Explored hyperparameters include the batch size, the dropout rate, the learning rate and weight decay for the Adam optimizer, as well as the temperature for the InfoNCE loss. Starting with common values found in literature and conducting empirical evaluations to decide on fixed values for our experiments with both GNNs, each model is then trained for 50 epochs using the largest possible batch size of 8,192 function pairs on our system with NVIDIA H100 GPUs.

To sanity-check the implementation of our approach, we conduct an additional experiment where each function is assigned a random function embedding. In this scenario, we ensure that the GNN's performance does not improve significantly after training, as that would reveal a flaw in the methodology.

3.2 Function Labeling

The first downstream task we consider is XFL, a recent approach to predict function names which leverages the output of pretrained embedding models in a straightforward manner. We select this task for our evaluation because profound semantic understanding of the binary function is required to make accurate predictions. In essence, XFL creates a label space for a dataset and decomposes function names into a set of normalized labels. These are then predicted in a multilabel classification task by passing the precomputed embeddings through a tree-based classifier. This methodological split between the function embedding and the function label prediction stage makes it difficult to finetune deep learning-based embedding models on this task. As a result, it makes sense to consider alternative methods to refine the embeddings, for instance on the call graph.

Adapting it to our dataset and embeddings, we reuse the original XFL implementation as far as possible and leave functionality such as the handling of common statically identifiable function names untouched. As our main goal is to compare the generalization capabilities of precomputed embeddings rather than achieving a new state of the art, we abstain from any XFL-specific hyperparameter tuning and use the default values set by the authors. For this task, we consider only C functions and exclude C++ functions due to their mangled function names, resulting in a subset of approximately 351.3k functions.

3.3 Compiler Optimization Detection

To examine the retention of technical details in the embeddings, we additionally perform a subtask of compiler provenance recovery. Leveraging the metadata in the BinaryCorp dataset, we introduce COD as a second downstream task. Predicting the underlying compiler optimization allows us to evaluate how integrating information from a function's callees affects the resulting embeddings' sensitivity to low-level structural variations induced by the compiler. Moreover, by training the GNN models on BCSD, we assess how the focus on higher-level semantic similarity tasks affects the downstream performance on syntactic tasks.

As COD is a multiclass classification task that labels the functions with a compiler optimization from {`O0`, `O1`, `O2`, `O3`, `Os`}, a linear layer is trained on the precomputed embeddings to predict the correct optimization level. Similar to the setup used for XFL, our approach is equivalent to freezing all parameters of both CLAP and jTrans as well as the GNNs for the downstream task. Training of the linear head is performed by minimizing the cross-entropy loss.

[2] https://github.com/vul337/jTrans

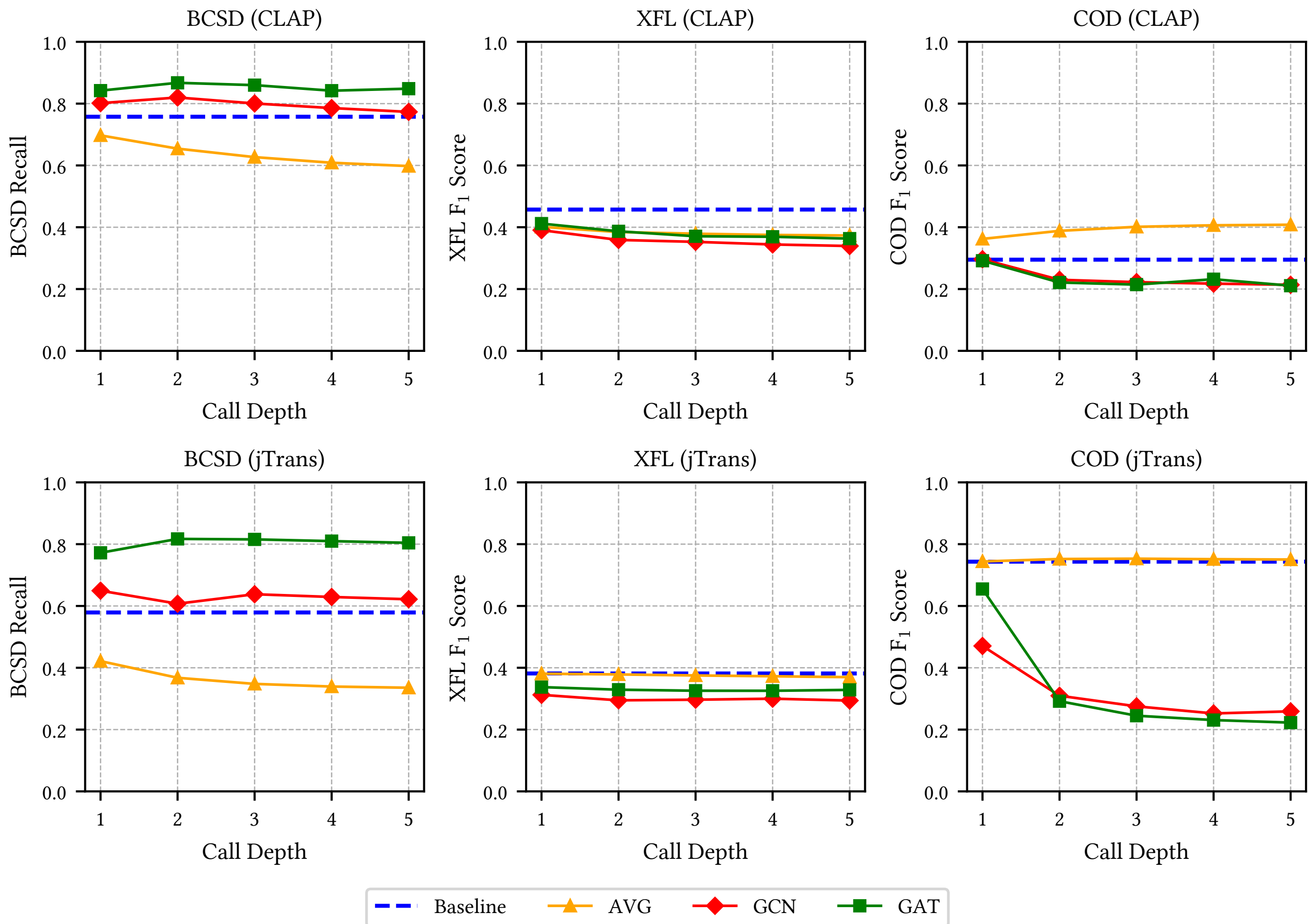

Figure 2: Task-specific results of CLAP and jTrans (Baseline) as well as the GNNs (GCN and GAT) and averaging approach (AVG) given different depths of the call graph. The diagrams plot the recall@1 scores for binary code similarity detection (BCSD), micro F_1 scores for the function labeling task (XFL), and macro F_1 scores for the compiler optimization detection task (COD).

Initially, we also experimented with multilayer perceptrons (MLPs) containing one hidden layer, though the added complexity did not yield significantly different results. By selecting one function per function pair, our dataset consists of approximately 1.4 million functions and the model is trained for 50 epochs with a batch size of 32.

4 Results

In this section, we evaluate all pretrained baseline models and graph-based models on the three tasks outlined in Section 2. Figure 2 plots the test set performance of each model per task. The top row depicts the diagrams using CLAP as a backbone, whereas the bottom row concerns the jTrans backbone. The following sections provide an overview of the evaluation method and an analysis of the results. As we regard the call graph-based models with a call depth from 1 to 5, we refer to them as AVG(n), GCN(n), and GAT(n) for a call depth of n, respectively.

4.1 Embedding Refinement on BCSD

To evaluate the BCSD task, we use a pool size of 1,000 function pairs (2,000 functions in total). For each function, this pool consists of the single positive counterpart from the same source and 1,998 negative function samples drawn from the other 999 pairs. The first column in Figure 2 shows the recall@1 scores of each model after training. This metric represents the share of functions for which the correct positive counterpart is the top-ranked result, based on cosine similarity, among all 1,999 candidate functions. Following the approach of CLAP [35] and jTrans [36], the mean reciprocal rank (MRR) was also taken into consideration as a performance metric. This metric proportionally accounts for the rank of the correct item, with higher scores for items ranked closer to the top. However, the observable patterns do not differ considerably between both metrics. Due to its more direct interpretability, we choose to only focus on the recall@1 metric.

Multiple insights can be obtained from these results. Firstly, we are able to reproduce previous findings that CLAP performs

significantly better on the BCSD task than jTrans [7, 35]. Moreover, it becomes evident that GNN-based models are able to extract meaningful information solely from the call graph in order to improve a backbone's performance on detecting same-source functions, confirming the intuition applied in previous work [12, 37]. With exception of the generally worst-performing model—the GCN on jTrans—the call depth of 2 appears to strike the best balance between contextual yet relevant information on our dataset. This appears to be slightly more shallow than the typical call depth of 4 reported in related work [12, 16, 37, 40], highlighting that the optimal call depth is sensitive to the general experiment setup. In our experiments, the more sophisticated GNN variant clearly outperforms all other models. The usage of GAT(2) leads to a performance improvement of almost +0.11 from 0.758 to 0.867 for CLAP and an improvement of +0.24 from 0.579 to 0.817 for jTrans, almost achieving comparable results to the CLAP-based GATs.

While similar trends can be observed for both models, the variance between context-enhanced embeddings is much larger with the jTrans backbone than CLAP. We hypothesize that particularly the GATs learn to recognize similar call graph structures and callee embeddings and investigate this further in Section 5. Taking into account the fact that the preprocessing pipeline CLAP retains more contextual information—though mostly about called library functions rather than internal calls—we believe the context-enhanced embeddings are able to address some of the shortcomings of jTrans compared to CLAP. Having said that, the results of the AVG models suggest that the new contextual information must be combined in a more involved manner, as none of the models outperforms the baseline set by the backbone, and the performance deteriorates further for larger call depths.

4.2 Semantic Downstream Performance on XFL

The middle column of Figure 2 presents the F_1 scores of each model on the XFL task. With a label space of approximately 1,024 distinct classes, the same class imbalance occurs on our dataset as discussed in the original paper [28]. Therefore, we follow the same approach and only investigate the micro-averaged scores, i.e., where all true positives, false positives, and false negatives are aggregated across classes rather than weighting each class equally.

The most striking insight is that both CLAP and jTrans outperform every model that attempts to enhance the embeddings with contextual information from the call graph. It appears that the semantic aggregation of functions learned on BCSD is not directly applicable to XFL. The comparison between the graph-based models underscores this further. While the simple averaging of embeddings in the call graph is significantly outperformed by the GNNs on the BCSD task, the AVG models perform comparably, if not even better, than the GNNs on the XFL task.

These results suggest that same-source binary code detection may not be a suitable task to ensure a semantic understanding that can be generalized to other tasks. At the same time, it is not possible to conclude that function embeddings cannot be refined on the call graph to improve for function labeling tasks. In Section 5.4, we investigate more closely in what way the investigated models may have benefited from the training regardless.

4.3 Impact of Semantic Similarity Pretraining on Syntactic Downstream Tasks

The right column of Figure 2 shows the models' F_1 scores on COD. As there is a minor class imbalance caused by the dataset which should be evened out, we use the macro-averaged F_1 score as opposed to micro-averaging as done for XFL. The patterns visible in the results of this syntactic task contrast strongly with the more semantic BCSD task. Firstly, neither GNN model type benefits the performance after being trained on refining the embeddings for BCSD. Instead, the performance keeps deteriorating particularly starting at a call depth of 2. The AVG models, on the other hand, outperform their backbone and continue improving with access to larger call depths when using CLAP as a backbone. These observations suggest that, to a certain extent, the simple approach of averaging all embeddings aggregates low-level technical details and patterns much better than high-level semantic information.

That being said, it is important to note that neither CLAP nor jTrans reach evaluation scores similar to those reported in literature that investigates task-specific models which are specifically developed for compiler provenance recovery [15, 31]. While it is intuitive that models trained to abstract away low-level features will perform worse on a syntactic task like COD, it is remarkable how much the performance differs between both backbones. CLAP clearly outperforms jTrans on both other tasks, whereas it is significantly surpassed on COD with an F_1 score of 0.295 as opposed to 0.743. In this case, the reduced access to information reconstructed by IDA in the preprocessed inputs may actually benefit jTrans on the COD task. This establishes a consistent pattern across both the pretrained backbones and the refined call graph-based models, highlighting the tension between abstractive power and technical precision and thus an inverse correlation where stronger semantic similarity capabilities come at the cost of lower performance on syntactic downstream tasks such as COD.

5 Explanatory Analysis

In this section, we dismantle the semantic task results from Section 4 by grouping them across various dimensions, including function and call graph size, compiler optimization, and source language. Using the normalized labels from XFL, we also inspect the results per label, categorizing them into namespace-related and other labels as described in Section 5.4. Specifically targeting BCSD, we additionally group by function pair properties such as the similarity between both functions, the highest similarity between any of their callees, and whether their call graphs are likely isomorphic or not. Our analyses focus on answering the following four research questions:

- **RQ1:** How beneficial is the embedding refinement on the call graph for semantic tasks if large amounts of contextual information is available?
- **RQ2:** How robust are the embedding refinements toward diversity within function pairs?
- **RQ3:** How robust are the embedding refinements toward different technical implementations?
- **RQ4:** Are the call graph-based models more advantageous on certain types of function labels?

In our deep-dive, we focus on the best-scoring instance of each model family for the respective task. As such, we neglect the GCN models entirely due to their inferior performance compared to the GATs. Additionally, we only take into account grouped results with a sample size of at least 200. While we take both CLAP- and jTrans-based results into consideration, the data points we present focus mostly on CLAP as a backbone due to its superior performance. Because CLAP embeddings provide a more meaningful starting point of the node embeddings, we expect the insights to be more informative. However, we do point out cases if the patterns differ for jTrans.

5.1 Benefit of More Context

Intuitively, the contextual embedding refinements should be particularly useful when more context is available. Take, for instance, the `CB_InputChanged` function from `photivo-git-photivo` on the BCSD task. Both function instances compiled with the optimizations `O1` and `O3` call nearly 140 distinct internal functions. Their CLAP embeddings do not show any significant similarity, with both functions having a higher similarity to more than a quarter of the remaining batch. However, using both the GAT(2) model and even AVG(1), the function pair is correctly matched among all 1,000 function pairs.

To quantitatively verify our intuition regarding the impact of context size on the performance of call graph-enhanced function embeddings, we consider both the number of nodes in the call graph as well as the number of instructions in the regarded function itself. Specifically, we group the functions logarithmically in buckets labeled $\lfloor \log_2(\text{Nodes}) \rfloor + 1$ and $\lfloor \log_2(\text{Instructions}) \rfloor + 1$, representing the number of bits needed to store the number of nodes or instructions, respectively. For the sake of comparability between models, we only present the number of nodes for a depth of 1. This equates the number of distinct internal callees plus one, as the key function in question is also included in the call graph.

Starting with the XFL task, all models demonstrate better performance for the shortest and longest functions with the fewest and the most callees. On the one hand, this means that all models profit from large amounts of available context. The reasoning for the performance increase given short functions is addressed further in Section 5.4. Regardless, the trend is comparable across all models including the backbones. That is, the performance change on XFL after the training on BCSD with the call graph does not depend noticeably on the function size.

In contrast, the embedding refinement does show a noticeable improvement on the BCSD task particularly for large functions and call graphs. Exemplarily, Figure 3 plots the CLAP-backed model performance based on the number of nodes in the call graph and displays the number of functions per bucket. While the call graph-enhanced embeddings demonstrate a steady increase in performance with more unique callees, CLAP performance rapidly declines starting at a number of $2^{(5+1)} = 64$ nodes in the call graph.

Intuitively, both grouping dimensions—number of call graph nodes and number of instructions comprising the function—are correlated with each other, as larger functions will have more call instructions with more distinct callees on average. Therefore, it comes as no surprise that largely equivalent patterns emerge when

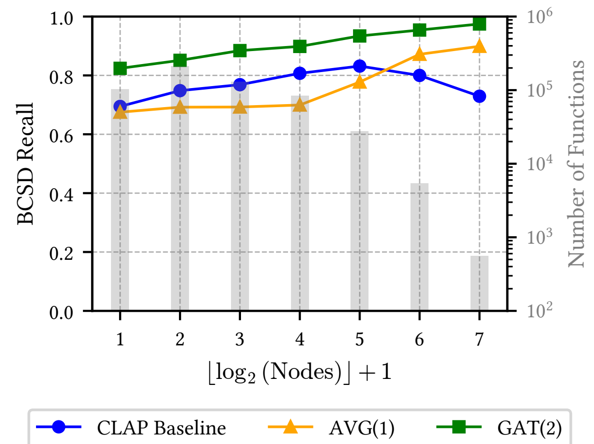

Figure 3: Grouped BCSD results of CLAP-backed models by exponential number of nodes (i.e., number of bits needed to represent the number) in the call graph (call depth 1).

focusing on the instruction buckets. It is important to note that grouping by both dimensions jointly yields comparable results as well—the only exception being jTrans, which performs best on longer functions with smaller call graphs, likely due to the fact that its preprocessing pipeline obscures potentially helpful contextual information. Particularly for the best-performing GAT, on the other hand, the performance generally increases both with more callees and with more instructions.

The key explanation for the performance drop of CLAP and jTrans lies in their limited context size of 1,024 tokens and 512 tokens, respectively. Taking into account that instructions typically consist of several tokens, this limit coincides well with the cutoff point before the performance drop. As the graph-based models examined in this paper do not aggregate semantic information on a token- or instruction-level, but rather on a function-level, they are able to effectively work against the fixed-window-size restriction of Transformer-based function embedding models. Furthermore, in the case of jTrans, the GAT(2) model is able to beneficially aggregate the callee embeddings to counteract the backbone's performance decrease given larger call graphs.

Summary: *Contextual models learn to harness larger amounts of context for BCSD, whereas the performance of CLAP and jTrans decreases eventually due to the limited size of their context window.*

5.2 Semantic Robustness on BCSD

In this section, we analyze how similarities and divergences between paired functions impact the models' performance on BCSD. The function `window_editor_map_draw_panels` from the program `augustus-game-git-augustus-game` serves as a motivating sample. The function pair using the optimizations `O0` and `O2` yields CLAP embeddings with a cosine similarity close to 0, and there are no remotely identical function embeddings with a similarity above 0.75 in their differently sized call graphs with a depth of 2. Nonetheless, both GAT(2) and AVG(1) successfully aggregate the embeddings such that the function pair is correctly matched in

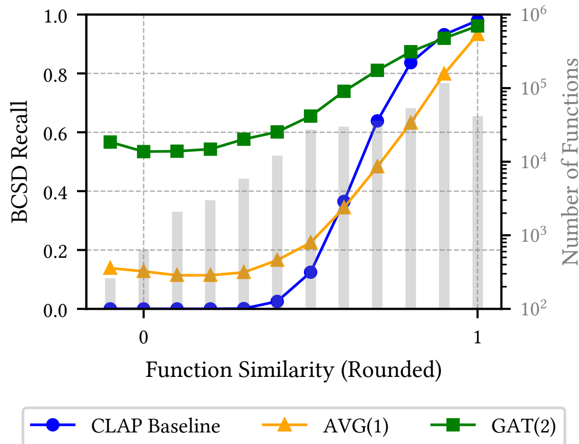

Figure 4: Grouped BCSD results of CLAP-backed models by function similarity.

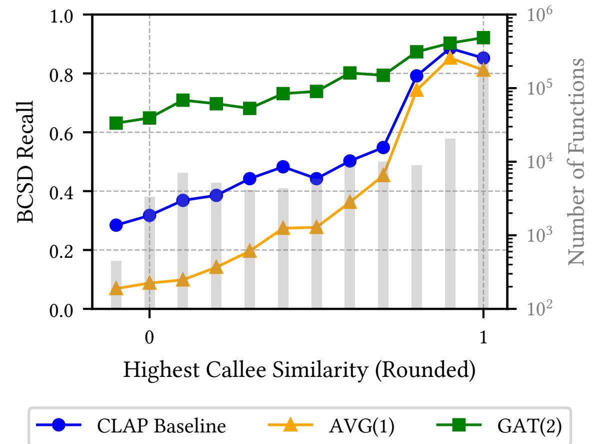

Figure 5: Grouped BCSD results of CLAP-backed models by highest similarity between callees (call depth 1).

its evaluation pool, raising the question how these models deal with cases in which the function embeddings or call graphs do not display significant similarities.

To attain a quantitative conclusion, we examine the grouped results for each related dimension. As the patterns align for both backbones, we only present results for CLAP. Figure 4 plots the recall@1 score grouped by the rounded cosine similarity between both functions' CLAP embeddings. Note that cosine similarities can assume values within $[-1, 1]$, which is why the data points also include negative similarities. It becomes visible that for pairs with a similarity near 1, it may not be beneficial to add more context from the call graph. However, as anticipated, particularly the CLAP baseline's model decreases rapidly toward 0 for lower similarities. In contrast, the GAT(2) model always demonstrates a performance above 0.53 for all buckets, showcasing that the GNN is capable of effectively integrating the knowledge from the call graph in more than half of the cases where the function bodies themselves are entirely dissimilar.

One natural reason behind this performance is that some function pairs call other functions which do not differ across compiler optimizations and therefore support the graph-based plug-in models to bring the function embeddings closer together. However, the data shows that GAT(2) also yields superior results in more intricate scenarios in which none of the callees are similar between both functions in question. This is shown in Figure 5, which plots the recall@1 score grouped by the highest similarity between any two callees in the function pair's call graphs. While this diagram only refers to the functions' direct neighbors in the call graph, it should be noted that the trends stay the same with a call depth of 2. Drawing from these insights, we find that the GNN is capable of meaningfully aggregating all embeddings in the call graph despite the individual embeddings yielding low similarity scores to their counterparts in the function pair.

In addition to the node embeddings, similar insights can be made with regard to the graph structure. Table 1 lists the models' performance for cases where both call graphs of depth 2 are non-isomorphic or likely isomorphic, as determined using the NetworkX library [13]. On this dimension, particularly the jTrans-based results reinforce our previous findings on the semantic robustness of GAT(2) on BCSD, displaying a difference in recall@1 scores of only 0.06 compared to the baseline's 0.20.

Table 1: Grouped BCSD recall@1 results by graph isomorphism of function pair's call graphs (call depth 2).

	non-isomorphic	**likely isomorphic**
Functions	385,218	173,506
CLAP Baseline	0.702	0.878
CLAP AVG(1)	0.613	0.881
CLAP GAT(2)	0.836	0.933
jTrans Baseline	0.517	0.714
jTrans AVG(1)	0.277	0.741
jTrans GAT(2)	0.796	0.860

Summary: *The GAT(2) model demonstrates more robust BCSD performance in cases where the function embeddings or the call graphs are less similar.*

5.3 Technical Robustness on BCSD

Turning to the technical robustness of the regarded models, an expressive sample is the C++ function `decompress_etc_eac` from the program `nvidia-texture-tools-git-nvcompress` compiled with the optimizations `O0` and `Os`—the pair on which CLAP demonstrates the worst performance. In this case, the CLAP embeddings are more similar to more than a third of the remaining batch than to each other, while being matched correctly by GAT(2) and AVG(1), suggesting that embedding refinements become more robust to technical details.

To evaluate this robustness more systematically, we first consider the source language, i.e., if the binary code originates from

Table 2: Grouped BCSD recall@1 results by source language.

	C	C++
Functions	108,616	450,108
CLAP Baseline	0.889	0.725
CLAP AVG(1)	0.890	0.650
CLAP GAT(2)	0.930	0.851
jTrans Baseline	0.780	0.529
jTrans AVG(1)	0.692	0.356
jTrans GAT(2)	0.884	0.799

Table 3: Statistics of BCSD recall@1 results grouped by compiler optimization pairs.

Model	min	max	std
CLAP Baseline	0.650	0.884	0.100
CLAP AVG(1)	0.561	0.878	0.131
CLAP GAT(2)	0.802	0.933	0.048
jTrans Baseline	0.341	0.864	0.172
jTrans AVG(1)	0.206	0.776	0.185
jTrans GAT(2)	0.732	0.912	0.064

Table 4: Grouped BCSD recall@1 and XFL micro F_1 results of CLAP-backed models by function label class. The best-scoring instance per model family is chosen for each task (AVG(1) and GAT(2) for BCSD, AVG(1) and GAT(1) for XFL).

	BCSD (recall@1)		XFL (micro F_1)	
	namespace	other	namespace	other
Functions	55,134	278,614	27,567	139,307
CLAP	0.802	0.867	0.625	0.411
CLAP AVG	0.799	0.868	0.607	0.342
CLAP GAT	0.894	0.919	0.604	0.358

a C or a C++ function. Table 2 shows the recall@1 values for all models, revealing the same patterns for CLAP- and jTrans-backed models. Without falling behind on C functions, the GAT(2) model achieves much more similar performance on C++ functions, with a performance difference of less than 0.08 compared to the twice as large drop by the CLAP baseline. Investigating possible correlations between the source language and other examined dimensions, we find that there is no integral correlation between C++ functions and a larger graph size. However, the stronger level of abstraction found in C++ compared to C can be unraveled by splitting the grouped results into non-isomorphic and likely isomorphic graphs. Here it becomes apparent that many more graphs are isomorphic for C functions than for C++ functions. Nonetheless, filtering only for non-isomorphic graph pairs and thus subtracting out correlations between source languages and the share of isomorphic graph pairs, the GAT(2) model continues to demonstrate a much smaller performance drop from 0.910 to 0.825 from C to C++ functions, whereas the scores for CLAP drop from 0.875 to 0.676.

That being said, an indicator for the technical robustness of the GAT-based model that has less potential for correlations with other factors is the differentiation of results between compiler optimizations. For this, we group the BCSD results by pair of compiler optimizations such as `O0,O1` or `O1,Os`, and regard the absolute range as well as the standard deviation between the different pairs. As can be seen in Table 3, the GAT(2) model demonstrates a significantly lower performance fluctuation across compiler optimization pairs compared to both CLAP and jTrans. For instance, while recall@1 scores differ by up to 0.23 for CLAP and 0.52 for jTrans, this difference is reduced to 0.13 and 0.18 using the GNN, respectively. While it is arguable that in the case of CLAP, both models triple their error rate on the worst-performing optimization pair compared to the best-performing one, the GAT(2) model demonstrates superior relative robustness using the jTrans backbone. Here, the error rate of jTrans increases almost fivefold, whereas GAT(2) remains at a threefold increase. However, in the absolute sense as argued above, we consider the GNN-based model more robust compared to CLAP as well. On BCSD, the call graph-based GNNs thus prove that their focus on aggregating higher-level information makes them less sensitive toward lower-level changes induced by different compiler options. Due to the limited transferability of these learnings to XFL, it is important to note that no such differences in robustness toward compiler optimizations can be observed for XFL. At the same time, with CLAP's XFL F_1 scores ranging from 0.406 to 0.457, for instance, the baseline models themselves do not demonstrate nearly as large susceptibility to such technical factors.

Summary: *The GAT(2) model demonstrates more robust BCSD performance toward different source languages and compiler optimizations.*

5.4 Impact of Label Classes

Lastly, we aim to determine at a higher level which types of functions may benefit more from the application of the call graph-based models. For this, we analyze the performance change concerning both BCSD and XFL for each function label generated by XFL. As the number of labels is too large to systematically draw insights from the results, we classify them into namespace-related labels versus other types of labels, effectively distinguishing between labels outlining the function context and function-specific descriptors. As it is not feasible for us to perform a per-function determination of which label instances act as a namespace and which act as the actual function description, we assign all instances of each label to either the namespace class or the other class. For this, we prompted three different LLMs, namely Gemini 2.5 Pro [6], Claude Sonnet 4 [2], and DeepSeek R1 0528 [9]. We adopted unanimous classifications after a brief manual verification, while resolving all disagreements through domain knowledge and online searches to identify library namespaces.

Table 4 presents the resulting BCSD and XFL scores grouped by the function label class for CLAP-backed models. On average for BCSD, functions with a stronger focus on the namespace perform worse. However, the margin is much smaller for the GAT(2) model, where the label class-specific scores are 0.025 apart from each other, compared to the baseline model with a margin almost three times

the size. For XFL, in turn, all models perform significantly better on predicting the contextual labels compared to function-specific labels. While the GAT(1) model performs worse on the function labeling task than its baseline, as discussed in Section 4, the data shows that the performance drop is much less significant for namespace-related labels. While the F_1 score decreases by approximately 0.02 in this case, the score specific to other function labels drops by more than 0.05. This reaffirms our previous findings that the call graph-based models are more beneficially used in context-dependent scenarios such as predicting namespaces compared to cases that mostly concern a function individually.

Take, for instance, the following two exemplary labels providing an illustrative view on the effect of using the GNN model: `glfw`, the namespace for an OpenGL library and the 53rd most common label in our dataset of C functions, has a comparatively high F_1 score of 0.836 using CLAP, which is nevertheless slightly outperformed by the GAT(1) model which achieves an F_1 score of 0.844. In contrast, the most common label, `get`, though achieving a comparatively high score of 0.640 using CLAP, is not predicted as well by the GAT(1) model with an F_1 score of 0.533.

An additional noteworthy insight can be made by grouping jointly by the label class as well as the number of callees. Interestingly, the worst-performing group on BCSD and by far the best-performing group on XFL correspond to namespace-related function names without any internal calls. While a possible explanation on CLAP's side may be that these functions likely call external library functions particularly often—for instance, functions in the `java` namespace calling external functions with `java` in their name—this reasoning cannot be applied to jTrans. Here, such function calls are preprocessed as `call callfunc_xxx`, shifting the explanation toward structural similarities between namespace labels. However, as this trend stays consistent throughout both the backbones as well as the embedding refinement models, we deem a thorough explanatory analysis at this point beyond the scope of this paper and leave it open for future work.

Summary: *Performance changes of contextual models on semantic tasks are favorable for namespace-related functions compared to those focused on individual logic.*

6 Threats to Validity

Internal validity. The threats to internal validity arise primarily from design choices made in this study. We focused on examining trends in model performance changes as opposed to achieving the highest possible metric scores. As such, we opted for widely known and comparatively simple GNNs instead of the latest state of the art, and did not perform XFL-specific hyperparameter tuning. Nonetheless, we acknowledge that our findings could partly be influenced by our experiment setup, as seen by the deviation of the optimal call depth in our experiments from previous literature referenced in Section 4.1. Similarly, our contextual call graph-based models were trained only on BCSD. While specifically targeting downstream tasks that may not allow for simple finetuning of the embeddings, it must be noted that the selected training task may have restricted which information the models learned to capture. Finally, our two-step approach using a plug-in architecture potentially affected the interaction between intra- and inter-function information. Subject to computational constraints, architectures integrating contextual information from the call graph more directly into the encoder's training process could yield different patterns and would be an interesting avenue for future work.

External validity. A main threat to external validity is the selection of our downstream tasks. In particular XFL's decoupled approach to predicting function names may limit the extent to which our insights generalize to related function labeling tasks such as BLens [4], which optimizes the encoder's weights throughout the entire training process. Another external validity concern is the selected dataset. As all experiments are grounded on BinaryCorp, the observed behaviors might be influenced by the dataset's code distribution, compilers, or labeling schemes, and may not hold for other binary corpora. Moreover, the selected corpus is limited to benign binaries. Other trends may be observed when analyzing the context-enhanced embeddings on malware, particularly obfuscated code with complex function relationships, which may limit the extent to which our findings transfer to the software security domain.

Construct validity. Our explainability analyses pose potential threats to construct validity. Although we explored a diverse and as comprehensive set of dimensions as possible, additional grouping factors may exist that were not captured in this study. Particularly higher-level semantic dimensions may yield more findings in addition to our analysis where we split by namespace versus function-specific labels. Similarly, we relied on a specific type of explainability approach centered on dataset slicing. Other established approaches, such as attribution techniques that identify the most salient parts of the input, have started being applied in the domain of binary code [7] and could provide complementary insights into how models leverage the call graph if applied systematically. Lastly, because our explainability analyses were performed on GNNs trained on the BCSD task, the patterns observed may be influenced by task-specific factors, and could differ if the models were trained on other tasks.

7 Related Work

A lot of recent literature is dedicated to creating generalizable embeddings of binary code semantics. BCSD is commonly the central task evaluated in this context. Well-known work in this realm includes models such as Asm2Vec [10] and SAFE [23], two approaches based on static embedding models like word2vec [25] that aggregate fixed token vectors to a function embedding. More recent models leverage the success of Transformers [33] in the NLP domain and apply them to binary functions as well, thus tuning the embeddings more profoundly on the context of instructions. Prominent examples include PalmTree [20] and Trex [29], with the latter applying the Transformers on so-called micro-traces of the execution. However, these models are significantly outperformed by jTrans [36] and CLAP [35], the Transformer-based backbone models investigated in this work.

Due to the structural nature of software code that can be represented as graphs, the use of GNNs has been explored in various forms as well. The control flow graph (CFG) between single instructions or linear basic blocks is leveraged in various architectures

including Gemini [39], Order Matters [41], and VulHawk [22], as well as in experiments by Massarelli et al. [24] on how to extract features from the CFG. Some models incorporate graph features in addition to the CFG. For instance, VulSeeker [11] integrates the data flow graph (DFG) and XBA [18] also models information such as external function calls and references to string literals.

However, the key graph component investigated in this work is the call graph, in which nodes are represented by functions rather than lower-level sections of the code. While early work by Liu et al. on αDiff [21] leverages information from the call graph, it boils down to the functions' in- and out-degrees only. In more recent years, models such as BMM [12] and CFG2VEC [40] were presented, which process call graphs alongside CFGs and—in the case of BMM—DFGs. Lastly, BinEnhance [37] acts as a framework that improves embeddings generated by a pretrained backbone. It aggregates the call graph into a custom graph with additional edges representing data-co-use, address-adjacency, and string-use, and passes this graph through a GNN. It is important to note that our goal is not to outperform any of these approaches. For instance, the authors of BinEnhance argue that the call graph is insufficient to fully leverage inter-function semantics [37]. Instead, this work specifically aims to carve out and comprehend the role of the call graph, hence understanding in which cases the usage of call graph-dependent GNNs is particularly beneficial.

The task of predicting function names is closely related to BCSD, though less commonly used as a downstream task to evaluate the generalizability of function embeddings. Of the models just discussed, only CFG2VEC [40] is evaluated using a retrieval-based methodology for function name prediction. While earlier landmark papers in this domain, such as Debin [14], use a probabilistic model on a dependency graph, more recent work relies on creating meaningful semantic embeddings for better performance. Their embedding models are conceptualized as encoders for this task in much the same way as for BCSD, and are combined with some sort of decoder model that performs the prediction which is commonly set up as a multilabel classification task. We use XFL's [28] tree-based classifier in our work. The authors present DEXTER as its function embedding model which focuses on encoding features obtained via static analysis, including information from the call graph such as the number of the function's callers and callees. Other noteworthy approaches to function name prediction opt for deep learning-based decoders. For instance, SymLM [17] encodes functions using the pretrained Trex [29] model. It then decodes the function name with an MLP by concatenating embeddings from the target function as well as its most frequent callers and callees, thus also leveraging information from the call graph. NERO [8] and BLens [4], on the other hand, employ Transformer-based decoders. NERO augments the CFG with the calling context using static analysis for its encoder, whereas BLens attempts to consolidate previous advancements in the areas of BCSD and function name prediction by applying an ensemble encoder that leverages both PalmTree [20] and CLAP [35] as well as DEXTER [28].

Compiler provenance recovery, in turn, focuses on technical details of the code rather than its semantic meaning, e.g., the classification of compiler optimization levels. While this task is used to evaluate BMM [12], showing that task-specific training can indeed leverage the call graph as well, the most prominent work in this area is mostly decoupled from semantic tasks due to their contradictory goals as we found in Section 4.3. For instance, an early landmark paper by Rosenblum et al. [31] uses support vector machines (SVMs) and conditional random fields (CRFs) on features extracted from the CFG to predict the compiler family and version as well as the optimization, surpassing an accuracy of 97% on the latter classification given their dataset. More recent work leverages deep learning models including Transformer-based encoders in the case of BinProv [15], which employs a lightweight classification head to predict the compiler type and optimization level. GNNs have been employed in this area as well. Apart from Massarelli et al. [24], who also examine compiler family prediction in their experiments on extracting features from the CFG, Vestige [16] is a recent approach that applies a GAT on an attributed call graph to reconstruct the compilation provenance, demonstrating similarities to the models we analyze in this paper.

To our knowledge, only a limited amount of literature exists in the area of binary analysis that focuses on the understanding of binary function embedding models beyond conventional ablation studies. Most notably, previous work [7, 38] has leveraged explainability methods to investigate the importance of different instruction types on the function embedding. This is done by systematically applying occlusion-based saliency methods [3], an approach that—in the context of binary functions—assesses instruction importance by measuring how much the embedding changes after concealing it in the input. In our case, we focus on investigating the model performance in different cases rather than the importance of specific parts of the input. Therefore, we turn to an approach found in various application areas, for example to detect racial biases in algorithms [26], in which the dataset is sliced along domain-specific dimensions in order to better understand the behavior of machine learning models.

8 Conclusion

In this work, we investigate the impact of enhancing binary function embeddings with context from the call graph. Training several graph-based models on binary code similarity detection, we find that even the best-performing models do not necessarily generalize to a function labeling task. Moreover, our results suggest that optimizing for such semantic understanding will typically deteriorate model performance on syntactic tasks such as compiler optimization detection. By examining the performance of models along domain-specific dimensions, we show that GNN-generated embeddings are able to effectively aggregate larger amounts of contextual information found in the call graph, hence making them more robust in a number of scenarios in which the initial embedding models demonstrated more significant performance drops. We hope that our research motivates model developers to further explore how inter-function information can best be leveraged in binary analysis and which pretraining tasks allow for the best generalizability toward rather semantic or syntactic tasks, particularly given the inherent trade-off between semantic similarity and syntactic nuance identified in this paper. Finally, we advocate for the wider adoption of similar explanatory analyses to identify and address both shortcomings and strengths of emerging models.

Acknowledgments

We would like to thank Moritz Dannehl for developing the initial preprocessing pipeline that formed the foundation of our experimental setup. Additionally, we extend our gratitude to the anonymous reviewers for their helpful comments that improved the clarity of this paper.

References

[1] Hex-Rays 2021. *IDA Pro.* Hex-Rays. https://www.hex-rays.com/ida-pro

[2] Anthropic. 2025. Claude Sonnet 4. https://www.anthropic.com/news/claude-4. Accessed: 2025-08-18.

[3] Jasmijn Bastings and Katja Filippova. 2020. The elephant in the interpretability room: Why use attention as explanation when we have saliency methods?. In *Proceedings of the Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP.* Association for Computational Linguistics, Online, 149–155. doi:10.18653/v1/2020.blackboxnlp-1.14

[4] Tristan Benoit, Yunru Wang, Moritz Dannehl, and Johannes Kinder. 2025. BLens: Contrastive captioning of binary functions using ensemble embedding. In *Proceedings of the 34th USENIX Conference on Security Symposium* (Seattle, WA, USA) *(SEC '25).* USENIX Association, USA, Article 353, 20 pages.

[5] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119).* PMLR, 1597–1607. https://proceedings.mlr.press/v119/chen20j.html

[6] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and Chris Hahn. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).

[7] Moritz Dannehl, Samuel Valenzuela, and Johannes Kinder. 2025. Which instructions matter the most: A saliency analysis of binary function embedding models. In *2025 IEEE Security and Privacy Workshops (SPW).* 145–151. doi:10.1109/SPW67851.2025.00019

[8] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 225 (Nov. 2020), 28 pages. doi:10.1145/3428293

[9] DeepSeek AI. 2025. DeepSeek-R1-0528. https://huggingface.co/deepseek-ai/DeepSeek-R1-0528. Accessed: 2025-09-20.

[10] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP).* 472–489. doi:10.1109/SP.2019.00003

[11] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE).* 896–899. doi:10.1145/3238147.3240480

[12] Yixin Guo, Pengcheng Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2022. Exploring GNN based program embedding technologies for binary related tasks. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22).* Association for Computing Machinery, New York, NY, USA, 366–377. doi:10.1145/3524610.3527900

[13] Aric A Hagberg, Daniel A Schult, and Pieter J Swart. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference.* SciPy, Pasadena, CA, 11–15.

[14] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18).* Association for Computing Machinery, New York, NY, USA, 1667–1680. doi:10.1145/3243734.3243866

[15] Xu He, Shu Wang, Yunlong Xing, Pengbin Feng, Haining Wang, Qi Li, Songqing Chen, and Kun Sun. 2022. BinProv: Binary code provenance identification without disassembly. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses* (Limassol, Cyprus) *(RAID '22).* Association for Computing Machinery, New York, NY, USA, 350–363. doi:10.1145/3545948.3545956

[16] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. Vestige: Identifying binary code provenance for vulnerability detection. In *Applied Cryptography and Network Security: 19th International Conference, ACNS 2021, Kamakura, Japan, June 21–24, 2021, Proceedings, Part II* (Kamakura, Japan). Springer-Verlag, Berlin, Heidelberg, 287–310. doi:10.1007/978-3-030-78375-4_12

[17] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. SymLM: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22).* Association for Computing Machinery, New York, NY, USA, 1631–1645. doi:10.1145/3548606.3560612

[18] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. 2022. Improving cross-platform binary analysis using representation learning via graph alignment. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA '22).* Association for Computing Machinery, New York, NY, USA, 151–163. doi:10.1145/3533767.3534383

[19] Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *Proceedings of the 5th International Conference on Learning Representations* (Palais des Congrès Neptune, Toulon, France) *(ICLR '17).*

[20] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. PalmTree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21).* Association for Computing Machinery, New York, NY, USA, 3236–3251. doi:10.1145/3460120.3484587

[21] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-version binary code similarity detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18).* Association for Computing Machinery, New York, NY, USA, 667–678. doi:10.1145/3238147.3238199

[22] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture vulnerability detection with entropy-based binary code search. In *30th Annual Network and Distributed System Security Symposium* (San Diego, CA, USA) *(NDSS '23).* doi:10.14722/ndss.2023.24415

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

[24] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, Roberto Baldoni, et al. 2019. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In *Proceedings BAR 2019 Workshop on Binary Analysis Research.* 1–11. doi:10.14722/bar.2019.23020

[25] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[26] Ziad Obermeyer, Brian Powers, Christine Vogeli, and Sendhil Mullainathan. 2019. Dissecting racial bias in an algorithm used to manage the health of populations. *Science* 366, 6464 (2019), 447–453. doi:10.1126/science.aax2342

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

[28] James Patrick-Evans, Moritz Dannehl, and Johannes Kinder. 2023. XFL: Naming functions in binaries with extreme multi-label learning. In *2023 IEEE Symposium on Security and Privacy (S&P).* IEEE, 2375–2390. doi:10.1109/SP46215.2023.10179439

[29] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2023. Learning approximate execution semantics from traces for binary function similarity. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2776–2790. doi:10.1109/TSE.2022.3231621

[30] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. 2018. Malware detection by eating a whole EXE. In *The Workshops of the Thirty-Second AAAI Conference on Artificial Intelligence* (New Orleans, LA, USA) *(AAAI Technical Report, Vol. WS-18).* AAAI Press, 268–276.

[31] Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2011. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11).* Association for Computing Machinery, New York, NY, USA, 100–110. doi:10.1145/2001420.2001433

[32] Thalles Santos Silva. 2020. Exploring SimCLR: A simple framework for contrastive learning of visual representations. *https://sthalles.github.io* (2020). https://sthalles.github.io/simple-self-supervised-learning/

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, CA, USA) *(NIPS'17).* Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[34] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *Proceedings of the 6th International Conference on Learning Representations* (Vancouver, BC, Canada) *(ICLR '18).*

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

the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (Vienna, Austria) *(ISSTA '24)*. Association for Computing Machinery, New York, NY, USA, 503–515. doi:10.1145/3650212.3652145

[36] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA '22)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3533767.3534367

[37] Yongpan Wang, Hong Li, Xiaojie Zhu, Siyuan Li, Chaopeng Dong, Shouguo Yang, and Kangyuan Qin. 2025. BinEnhance: An enhancement framework based on external environment semantics for binary code search. In *32nd Annual Network and Distributed System Security Symposium* (San Diego, CA, USA) *(NDSS '25)*.

[38] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving binary code similarity transformer models by semantics-driven instruction deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '23)*. Association for Computing Machinery, New York, NY, USA, 1106–1118. doi:10.1145/3597926.3598121

[39] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, TX, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 363–376. doi:10.1145/3133956.3134018

[40] Shih-Yuan Yu, Yonatan Gizachew Achamyeleh, Chonghan Wang, Anton Kocheturov, Patrick Eisen, and Mohammad Abdullah Al Faruque. 2023. CFG2VEC: Hierarchical graph neural network for cross-architectural software reverse engineering. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice* (Melbourne, Australia) *(ICSE-SEIP '23)*. IEEE Press, 281–291. doi:10.1109/ICSE-SEIP58684.2023.00031

[41] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-aware neural networks for binary code similarity detection. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 01 (Apr. 2020), 1145–1152. doi:10.1609/aaai.v34i01.5466