LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-865575

BobGAT: Towards Inferring Software Bill of Behavior with Pre-Trained Graph Attention Networks

J. Allen, G. Sanders

June 13, 2024

IEEE International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications
Washington, DC, United States
October 28, 2024 through October 28, 2024

Disclaimer

BobGAT: Towards Inferring Software Bill of Behavior with Pre-Trained Graph Attention Networks

Justin Allen
Lawrence Livermore National Lab
Livermore, U.S.
allen119@llnl.gov

Geoff Sanders
Lawrence Livermore National Lab
Livermore, U.S.
sanders29@llnl.gov

***Abstract*—In recent years, a variety of Graph Neural Network (GNN) and Natural Language Processing (NLP) techniques have been proposed for binary analysis tasks including assembly embedding generation and binary similarity. However, a majority of these techniques were developed and tested on rather restrictive datasets or tasks, generally limited to one compiler, a limited number of compiler flags, and a small or artificially augmented dataset ill-suited for the semantic understanding of binaries required to generate a software bill of behavior. To that end, we have scraped a new programming competition dataset an order of magnitude larger than previous largest datasets, as well as designed a novel GNN architecture tailor-made for Control Flow Graph (CFG) analysis. We achieve over 92% accuracy on a complex 8000+ class programming problem classification task and perform extensive ablation studies showing the increased efficacy our model architectures provide and the necessity of proper dataset construction and deduplication for generalizability. We pre-train our models on an embedding task designed to improve our models' semantic understanding of binaries and showcase their ability to group similar binaries together by behavior. Finally, we train our model as-is on a popular malware benchmark dataset and achieve near state-of-the-art performance with little effort: 99.23% accuracy on the Microsoft Malware Classification Challenge (Big 2015) dataset. Our data preparation tools are open-sourced, well-documented and tested, and pip-installable.**

***Index Terms*—Malware, Graph Neural Networks, Natural Language Processing, Malware Classification, Machine Learning, Deep Learning, Control Flow Graphs**

I. Introduction

Practical software analysis on modern computing systems often only involves compiled binaries without access to the associated source code (e.g. commercial software, firmware, device drivers, malware, etc.). As such, the field of binary analysis has become pivotal to ensuring safe and effective usage of such software in user environments. Binary analysis techniques are utilized for tasks such as bug hunting [1], [2], malware analysis [3], and software theft detection [4]. Recent machine learning techniques have opened the door to automatic generation of a Software Bill of Behaviors (SBOB) for binaries - a list of behaviors that a given binary exhibits. Accurate SBOBs would be an invaluable tool for cybersecurity professionals as it would automate much of the laborious and convoluted task of reverse engineering a binary.

Motivated by advances in graph and natural language processing techniques, there has been much focus on modeling techniques that represent the binaries as directed graphs, such as function call graphs (FCGs) or control-flow graphs (CFGs), with complex "natural language" metadata living on graph vertices (representing the functions or basic code blocks). These relational and natural language-like data formats, along with improvements in deep learning, provide a firm basis on which to ground behavior identification techniques.

Analyzing binaries presents a fundamentally different and often more difficult challenge than that of analyzing source code. Much information is lost during the compilation process including exact object types, function descriptors, variable names, comments, and clear delineation between functions. It is possible to reconstruct some of this information, however compilation is a fundamentally lossy operation in general. Modern compilers utilize various optimizations that can change control flow of the resulting binaries. For example, entire sections of code may be deleted if found to be unnecessary during compilation. Function inlining can drastically change code structure and ruin the clear abstractions and organization of well-written code [5]. Loops can be entirely replaced with SIMD instructions. Constant values can be precomputed during compile time. Instructions can be rearranged to make better use of pipelining and cache memory.

Traditional approaches used deterministic algorithms to solve binary analysis tasks including graph isomorphism, symbolic execution, and theorem proving techniques [6], opcode frequencies [7], and locality-sensitive hashing techniques [8]. Recent methods tend to make use of machine learning (ML), especially neural networks, and show improved performance. Some approaches model binaries using more traditional machine learning algorithms such as Support Vector Machines (SVM) [1], random forests [9], or clustering [2]. Others use more complex neural network models including Recurrent Neural Networks (RNN) [10], [11], Convolutional Neural

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LDRD 22-SI-005. IM#: [INSERT NUMBER]

Networks (CNN) [12], and Graph Neural Network (GNN) models [13], [14], [15]. Instruction embedding generation is often a focus with methods including *word2vec*-like [16] models [17], [18] as well as the more recent *transformer*-based methods [19] and BERT-like [20] methods [21] [22] [23]. Some methods [24], [25] combine both transformers and GNN models to make use of both sequential token information and graph structure. IU Haq and J Caballero provide a great overview and survey of techniques in the field of binary similarity in [26].

However, many of these techniques are trained on datasets containing either few datapoints relative to model size, or a significant amount of likely similar or duplicated data. Datasets are often constructed from a small number of original binaries compiled only with various optimization levels. This increases the likelihood of neural network 'memorization' - when the models overfit on non-generalizable artifacts specific to the training data. All binary data will require some level of preprocessing and normalization to combat *out-of-vocabulary* (OOV) problems, however there has been little work directly comparing assembly tokenization and normalization techniques. Assembly embedding generation is common and a powerful tool for helping models learn token semantics, but few compare assembly embedding model architectures for use in binary analysis models. Trained models are often task-specific requiring full retraining to apply to new tasks. Few, if any, methods in the literature have the data and labels to train unsupervised models that can learn general binary semantics and behaviors.

To this end, we propose and implement a novel binary analysis ML pipeline capable of processing millions to billions of binary artifacts and able to employ combinations of several NLP and GNN advancements that have shown promise in the recent literature. We perform extensive testing and ablation studies to give insight into the more efficacious alterations to binary analysis models, and train our models on a complex binary similarity task designed to learn binary semantics and behaviors as opposed to just syntactics. First, we scraped an extremely large dataset from the popular programming competition website *Codeforces* [27], which is an order of magnitude larger than the next largest programming competition dataset we could find [28]. Second, we developed open-sourced tools designed to handle binary analysis data and remove much of the burden of compiling, analyzing, preprocessing, and normalizing binaries, followed by packaging them in convenient data structures for ML processing. Third, we build a framework for training GNN and NLP models with fidelity at the assembly line, basic block, function, and full binary levels simultaneously. We pre-train these models in a task-agnostic way, followed by fine-tuning, and perform ablation studies on hyperparameters and model architectures. Lastly, we train our models as-is on the Microsoft Malware Classification Challenge (Big2015) dataset [29] as a means of comparison.

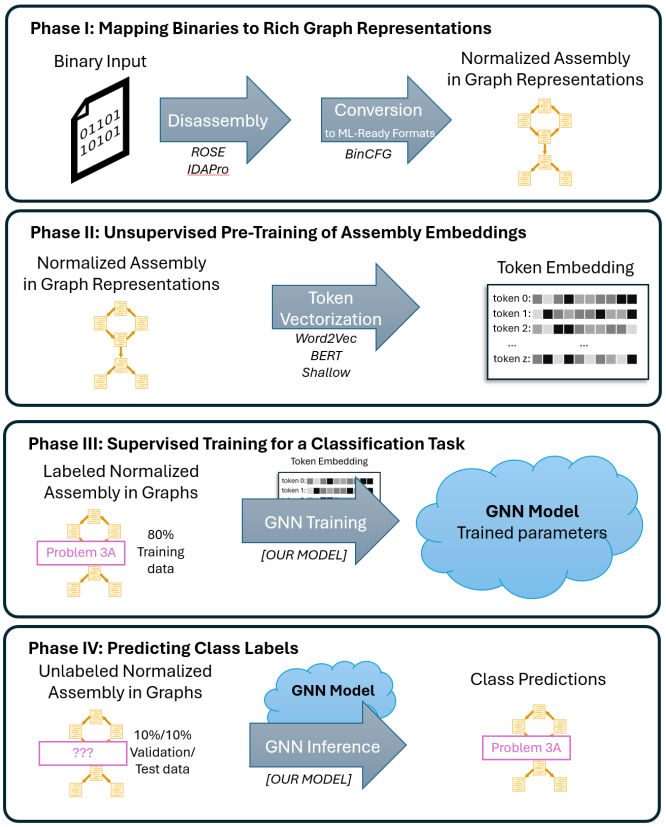

Fig. 1. Proposed work flow for binary modeling.

A. Contributions

We make the following contributions:

(a) We develop and implement scalable dataset curation and postprocessing tools for binary analysis artifacts and share them with the research community via open sourcing.
(b) We use these tools to cultivate and train models on a large, labeled binary analysis dataset comprised of programming competition submissions.
(c) We apply a unique CFG modeling approach capable of producing expressive pre-trained and fine-tuned models.
(d) We perform extensive evaluation and ablation studies to measure the efficacy of the models and to better understand which model components provide greater quality improvements.
(e) We train our final model as-is on the Big 2015 malware dataset as a means of comparison and achieve near state-of-the-art performance out of the gate.

B. Background

Function Call Graph. A Function Call Graph (FCG) is a directed graph $G_f = (V_f, E_f)$ with vertices V_f (functions) and edges $E_f \subseteq V_f \times V_f$ where an edge $(f_1, f_2) \in E_f$ denotes a directed function call edge from function f_1 to function f_2. These provide a high-level overview of the structure of a

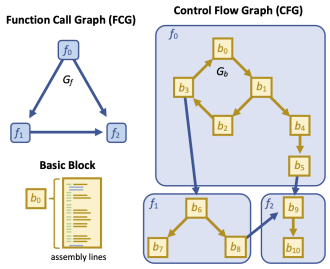

Fig. 2. Simple example of hierarchical graph representations of a binary.

binary, but are missing much of the fine-grained detail useful in training machine learning models on binary analysis artifacts.

Control Flow Graph. A Control Flow Graph (CFG) is text-attributed directed graph $G_b = (V_b, E_b)$ showing control flow through a binary. It has a set of vertices V_b (also called 'basic blocks') and a set of edges $E_b \subseteq V_b \times V_b$ where an edge $(b_1, b_2, \eta) \in E_b$ denotes a directed edge from block b_1 to block b_2 with an edge type $\eta \in \mathbb{Z}^+$. Vertices are uniquely determined by their memory address and are attributed with textual information in the form of a list of string assembly instructions. After preprocessing, a vertex b will contain an n_b-length sequence of integer tokens $T_b = (t_1^{(b)}, t_2^{(b)}, ..., t_{n_b}^{(b)})$. Edges can come in two types: 'normal', and 'function_call'.

Word2Vec. Given a large corpus of sentences (token sequences), words (tokens) can be embedded into a semantic vector space using word co-occurrence within context windows (local token subsequences of fixed length). In [16], authors develop two novel architectures: continuous bag of words (CBOW), which trains word prediction based on the few words before and afterwards, and skip-gram, which trains prediction of few words before and after based on each word. Previous authors utilize this approach to embed assembly [18], opcode [17], and basic blocks [**?**].

Each token is given an initial random shallow embedding vector. For some token at index t in a sentence, the surrounding $w-1$ tokens (w being the odd window length) have their embeddings summed and sent through a single linear layer to predict which token is actually present at index t using a softmax activation function. Errors are propagated backwards to the initial embedding vectors to learn useful embeddings.

BERT. By pre-training transformer encoder models on semi-supervised tasks, the authors in [20] produce pre-trained models able to be fine-tuned in later sequence learning tasks, and achieve state-of-the-art performance on common NLP benchmarks. They introduce two pre-training objectives: 1) the Masked Language Model (MLM) and 2) Next Sentence Prediction (NSP). For the MLM task, tokens in the sequence are randomly replaced with a '[MASK]' token, and the model is trained to predict which token was originally present in that location. For the NSP task, the model is presented two sentences (along with information about which tokens correspond to which sentences) and is trained to predict whether the second sentence immediately follows the first sentence somewhere in the corpus. These models are trained on vast amounts of data with intensive compute resources. This allows the models to learn general aspects of token semantics and sentence structure which can then be fine-tuned later with far fewer resources and time required than training from scratch.

Graph Neural Networks. GNNs have been gaining in popularity recently, with one of the more recent and effective architectures being the Graph Attention Network (GAT) [30]. Our models are based on a GAT approach. At each time step (layer), each node gathers the current embeddings from all of its neighbor nodes. The node then applies a multi-headed attention mechanism to those embeddings, and combines that output with its own embedding from the previous time step. A nonlinearity is applied to produce that node's new embedding for the current time step. This is repeated once for each 'layer' in the GAT. A GAT with k layers utilizes k-hop neighborhood information.

II. Data

The collection of binary analysis data for machine learning generally follows a similar workflow: gather source code, compile various ways, analyze with static analysis methods, and convert analysis artifacts into ML-ready formats. To this end, we have developed an open-sourced Python tool to help automate this process for binary analysis researchers. We then performed a large scraping of the Codeforces [27] programming competition website and sent that data through our compilation and analysis pipeline to generate training data for our models.

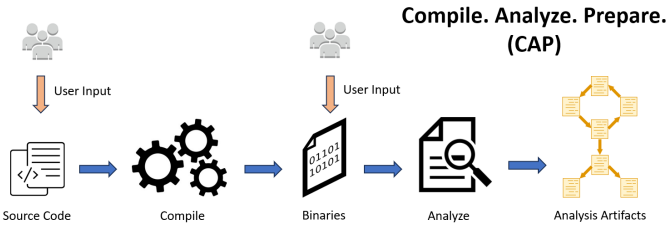

Fig. 3. Our data curation pipeline.

A. BinCFG (ML-Ready CFGs)

BinCFG [31] is a Python tool designed to parse analyzer outputs, tokenize and normalize the assembly lines, and convert that data into ML-ready formats. It can automatically parse the outputs from ROSE [32], and has an interface for users to build their own CFGs to make use of outputs from other binary analysis tools. This will produce an easily traversable Python object containing information on the functions, basic blocks, assembly lines, edges, and metadata. One can then tokenize and normalize assembly lines using one of the built-in normalization methods based on literature, or easily create new normalizer classes to user specifications. Finally, these can be converted into memory and space efficient representations suitable for scalable graph learning implementations.

B. Data Curation

Real-world collections of binaries are challenging to train statistical models for many reasons, including (i) a significant percentage of the artifacts may not compile, (ii) artifacts may have an alarming number of near-duplicates, and (iii) the lack of discrete labels covering complex compiled binary relationships

To address these problems, we scraped 125 million submissions to over 9000 problems from the Codeforces programming competition website [27], along with associated metadata. Of those, around 100 million are C/C++ and compilable.

This metadata provides some important mitigation for (iii), especially with regards to binary similarity as all submissions can be labeled based on the problem to which they were submitted. One can assume that, after some deduplication of problems, submissions to the same problem should be considered 'similar' and those to different problems 'dissimilar'. We perform a deduplication of problems initially by comparing histograms of character frequencies in problem descriptions, removing duplicates above a threshold, and manually inspecting those at or near the boundary to ensure accurate deduplication.

With respect to (i) and (ii), we compile all 100 million C/C++ compilable submissions on HPC using our compilation pipeline CAP (Compile. Analyze. Prepare.) [33], and aim to remove duplicate source code artifacts. All source codes were compiled with GCC's g++ compiler, version 12.1.0, to x86_64 with '-O3' optimization using the "g++-12-multilib-i686-linux-gnu" multilib cross compiler within a singularity container running Ubuntu version 22.04. The code is then analyzed using ROSE's 'bat-ana' and 'bat-cfg' tools to produce the CFGs running ROSE version 0.11.145.0.

If the source code input is truly identical, then this is an easy situation to resolve by hashing the source code or binaries. However, in practice two essentially identical CFGs often result from two codes with different comments, variable names, orders of passages of code removed by compiler optimizations, and other compiler transformations that remove source code discrepancies. One simple approach for deduplication is building an opcode histogram for each binary. A more involved class of techniques involves building a CFG for each binary artifact and computing relatively inexpensive statistics involving the CFG topology.

For this purpose, the BinCFG tool will build a vector containing: graph-level stats (number of nodes, functions, and assembly lines), histograms of node degrees and sizes (number of assembly lines at each node), histograms of function degrees (in the function call graph) and sizes (number of nodes in each function), and histograms of assembly tokens. We used a normalization method that replaces immediate values with '#immval#', replaces general registers with '#reg#X' where 'X' is the size of that register in bytes, and replaces call instruction operands with '#innerfunc#', '#externfunc#', and '#self#' for function calls within the binary, external calls to dynamically loaded functions, and recursive calls, respectively. These normalizations remove much of the extraneous information that is likely to not change binary functionality (memory addresses, generic registers) while keeping much of the important assembly line information. This results in a set of over 100 million 15,000-dimensional graph statistics vectors.

We then generate subsets of this data based on these graph statistics by selecting some number of submissions per problem based on which CFGs are most 'distant' from one another in a greedy fashion. We randomly select one starting datapoint and greedily add datapoints that maximize the minimum manhattan distance between any two CFG graph statistics vectors within the current group at each step until the specified number of datapoints is reached.

C. Size Restrictions

Due to the complex nature of programs and compilers, CFGs have almost no guarantees about the size of CFG components and statistics. Within our Codeforces dataset, we have found basic blocks containing over 150,000 instructions due to loop unrolling, functions with 27,000 nodes, and binaries with over 4500 functions. Using this data as-is would create troubles for GNN models including overfitting, over-smoothing, and high memory usage for backpropagation passes. To mitigate these problems, we enact some size restrictions on our input data.

Assembly instructions are limited to 256 per node. This limit applies to the number of tokens, not necessarily the number of instructions (depending on the normalization method used). For any nodes with more than 256 assembly tokens, we create a new token '#large_block#' to designate that block as one with too many tokens. We concatenate together: ['#large_block#', the first 127 tokens, '#large_block#', the last 127 tokens]. Our intuition is that this gives the model 1) the notice that this is a large block and assembly instructions are missing and 2) information from the beginning and end of the block so inter-block sequences can be partially reconstructed.

Aggregations are limited per aggregation such that:

- Nodes aggregated into function embeddings are limited to 256 (2% of CFGs affected)
- Functions aggregated into binary embeddings are limited to 256 (0.7% of CFGs affected)
- Edge aggregations are limited to 256 (no effect on any CFGs or FCGs)

For any aggregation that had too many sub-units, a random subset of the max size of sub-units were selected before each epoch. If, for example, a function had too many nodes, we would artificially remove the 'connection' between some nodes and that function such that it only has 256 nodes (the maximum allowed) during its aggregation. Those nodes would still exist in the graph during message passing; it is only the function's aggregation which would no longer see those removed nodes. These selections were made on a per-epoch basis so future epochs working with the same CFG may select a different subset of sub-units during training for their aggregations.

D. Big 2015

In order to compare our models with other state-of-the-art techniques, we also include tests on the Microsoft Malware Classification Challenge (Big 2015) hosted on Kaggle [29]. This dataset is a collection of 9 different malware classes with both their raw bytes (minus PE file header) and IDAPro disassembly provided. We parse only the assembly lines from the IDAPro output and build them into CFGs to showcase BinCFG's usefulness on other (non-ROSE) data sources.

We increase the aggregation limits on these binaries to 512 (for all node, edge, and function aggregations) to use more of the binary as these binaries were on average noticeably larger than Codeforces binaries.

E. Datasets

We build subsets of data which are used to train our models:

- *rand100* - select 100 examples per problem in Codeforces uniformly randomly
- *dist100* - select 100 examples per problem in Codeforces using the algorithm in II-B to select the 100 most 'distant' examples
- *Big2015* - the Microsoft Malware Classification Challenge (Big 2015) training dataset consisting of 10868 datapoints

Both the *rand100* and *dist100* comprise of approximately 850k unique submissions, each of which is then compiled to x86_64 once with GCC's g++ compiler version 12.1.0 using '-O3' optimization, and once with a random compiler (either gcc or g++, where applicable depending on the programming language used in the submission), random compiler version (7.5.0, 8.4.0, 9.5.0, 10.3.0, 11.3.0, 12.1.0), and random -O['0', '1', '2', '3', 's', 'g', 'fast'] optimization. These were compiled with the 'g++-X-multilib-i686-linux-gnu' apt package (where 'X' is the major version number) within a singularity container running Ubuntu version 22.10. This results in around 1.7 million total datapoints per dataset.

III. Methodology

This paper sets out to answer the following research questions:

- **RQ.1 Basic Block Embedding.** What techniques are best to embed assembly instructions within basic blocks?
- **RQ.2 GNN Architecture.** Do more complicated GNN architectures improve performance over base models?
- **RQ.3 Dataset Deduplication.** How important is dataset deduplication before training?
- **RQ.4 Spatial Embedding and Pre-Training.** Can we train models to embed CFGs close in space when they are semantically similar, thus providing an initial attempt at a software bill of behavior?

A. Tasks

We test our models primarily on predicting which unique problem a Codeforces submission's CFG belongs to, dubbed the 'problem_uid prediction' task. Our models are tested on this task first in order to determine ideal hyperparameters and model architectures to use.

For spatial embedding, models attempt to minimize a distance metric between binary-level embedding vectors of Codeforces submission CFGs belonging to the same unique problem, and maximizing the distance between those of different problems. This allows the model to learn semantic similarity as authors find new and unique ways to solve the same problem, and those examples are compiled in various ways generating semantically similar but very syntactically different code samples. We also show this model could be used as an initial pre-trained model that can then finetune on Codeforces problem identification far faster than without pre-training.

B. Pre-Trained Embeddings

RQ.1 involves testing multiple assembly embeddings techniques on the problem_uid prediction task. To do this, we pre-train Word2Vec and BERT models on the assembly within CFGs. These NLP models are trained on random walks of assembly instructions following control flow within the CFG. The process slightly differs between Word2Vec and BERT.

Word2Vec Sentence Generation: To generate sentences for Word2Vec models, we perform random walks through the CFG starting at random blocks and following directed control flow edges. The assembly instructions at each block are concatenated together to form the final sentence. We enforce a maximum sentence length of 128 tokens.

BERT Sentence Generation: The process for generating BERT sentences follows the sentence generation method of the original BERT paper closely. We perform random walks through the CFG starting at random blocks and following control flow until we reach a 'target length' of contiguous tokens. The 'target length' defaults to the maximum length of a full BERT input (256 tokens), however there is a 10% chance we will instead use a uniform random target length in the range [2, 256]. Sequences with < 2 tokens are dropped. We then attempt to randomly partition these sequences on basic blocks into two sentences A and B such that each sentence contains at least one basic block, and at least 1 token. If this can't be done (such as when only one basic block was returned in the sequence), we partition the tokens randomly such that each sentence has at least 1 token. There is then a random chance (50%) that we replace sentence B with another randomly chosen sentence from a different CFG generated with the same random walk technique. This provides the two sentences for BERT's Next Sentence Prediction (NSP) task.

We then mask tokens within the sentences, with a 15% chance to mask any token up to a max of 20% of the tokens within any one sentence being masked. There is an 80% chance that a masked token is replaced with the "[MASK]" token, a 10% chance it is instead replaced with a random token in our lexicon, and a 10% chance it is left unaltered. These operations produce the data for the Masked Language Model (MLM) BERT task.

Finally, BERT inputs are generated by concatenating "[CLS]" token + sentence A + "[SEP]" token + sentence B + "[SEP]" token, and padded with a "[PAD]" token out to the full 256-token length.

NLP Model Hyperparameters: Word2Vec models use the *Continuous Bag of Words* (CBOW) architecture with a hidden dimension of 128 and window size of 11. BERT models have a maximum sentence length of 256, a hidden dimension of 128, 4 transformer layers, 8 attention heads, and use a dropout of 0.1. Both models use the Adam optimizer with the default initial learning rate of 0.001.

C. Model Architecture

Symbol	*Description*
L_t	GNN layer at index t
Agg^t_{type}	Aggregation function for $type$ at layer t
U^t_{type}	Update function for $type$ at layer t
h_d	Hidden dimension of our model
N_{type}	Number of elements of a given $type$
F^t_{type}	Feature matrix $\in \mathbb{R}^{N_f \times h_d}$ for $type$ at layer t
$\hat{F}_{type}$	The final embeddings matrix for $type$
C_{type}	'Connections' matrix $\in \{0,1\}^{N_o \times n_f}$
n	Regarding the $type$ of nodes
f	Regarding the $type$ of functions
b	Regarding the $type$ of binaries
a	Regarding the $type$ of assembly tokens
c	Regarding the $type$ of function call graph
l	Regarding the $type$ of GNN layers
$[\cdot, \cdot, \dots]$	Concatenation of matrices column-wise

Our base GNN model consists of one or more hierarchical GNN layers $L_0, L_1, ...$ similar to [15]. Each GNN layer performs a hierarchical set of message passings and aggregations for a layer t at the basic block, function, and binary level:

$$F^t_n = U^t_n([Agg^t_n(F^{t-1}_n,\ C_n),\ \ F^{t-1}_n])$$

$$F^t_f = U^t_f([Agg^t_f(F^t_f,\ C_f),\ \ F^{t-1}_f])$$

$$F^t_b = U^t_b([Agg^t_b(F^t_b,\ C_b),\ \ F^{t-1}_b])$$

The initial node embeddings are determined by aggregating token embeddings for each node:

$$F^0_n = U_a(Agg_a(F_a,\ C_a))$$

The final embeddings at the node, function, and binary level are simply the last hidden states:

$$\hat{F}_{type} = F^{N_l}_{type}$$

Where an element $c_{ij} = 1, c \in C_a$ if node i contains the token embedding at index j, $c_{ij} = 1, c \in C_n$ if there is a directed edge from node i to node j, $c_{ij} = 1, c \in C_f$ if the node at index j belongs to the function at index i, and $c_{ij} = 1, c \in C_b$ if the function at index j belongs to the binary at index i. The initial token embeddings F_a are computed in a variety of ways based on the test being performed: from word2vec-like embeddings, BERT embeddings, or a shallow learned embedding.

$Agg^t_{type}(F, C)$ is an aggregation function that, for each element in F, will aggregate all other elements in F that are 'connected to' that element, as determined by C. U^t_{type} is an update function meant to introduce a nonlinearity. For simplicity, we use the same Agg and U for each model, though they will use different learned parameters for each type/layer. Agg is a Simple Attention Mechanism (see below), and U is a single learned linear layer with a ReLU nonlinearity.

If we were only using one CFG per batch, then C_b would consist of all 1's. However, our current formulation allows for the ability to 'stack' together multiple binaries at a time to perform mini-batching during training. We concatenate together connections matrices diagonally such that elements from one binary will not interfere with those from another, and use C_b to designate which functions belong to which binary within a batch.

Simple Attention: This attention mechanism computes a simple multi-headed attention on a matrix of features $F \in \mathbb{R}^{N_{feat} \times d}$, and a 'connections' matrix $C \in \mathbb{R}^{N_{obj} \times N_{feat}}$ where $c_{ij} = 1$ if the object at index i 'connects to' the feature at index j within F. This is a general attention mechanism that reduces along a dimension and can be used to perform one or more different aggregations within models.

For each 'object' (row in F), we use a multi-headed simple attention attention mechanism to aggregate all feature vectors from F that 'connect to' that object, as designated by C. This process is repeated for all objects to produce new embedding vectors. In practice, we vectorize these operations using PyTorch sparse tensor operations.

Shallow Embeddings and Positional Encodings: For some experiments, we make use of a shallow learned embedding layer instead of pre-trained NLP embedding methods. Optionally, we insert positional encodings into the assembly embeddings within each node. We implement the same sinusoidal positional encoding scheme to those used in the original transformer paper [**?**]:

$$PE_{p,i} = \begin{cases} sin(p/10000^{i/d}), & i \text{ is even} \\ cos(p/10000^{(i-1)/d}), & i \text{ is odd} \end{cases}$$

$$T_{n,p} = E_{emb}(n, p) + PE_p$$

Where $T_{n,p}$ is the final pre-aggregation token embedding for the token at index p within node n, $E_{emb}(n, p)$ is the current learned shallow embedding for the token at index p within node n, and PE_p is the full positional encoding vector for the position p.

Skip Connections and Jumping Knowledge: For some experiments, we add skip connections to the initial node features and jumping knowledge into the models. We theorize that the assembly at each node plays a pivotal role in model efficacy, and that having direct connections to those unaltered initial assembly embeddings (rather than having to retain information through layers) will boost performance. When using jumping knowledge, the final node, function, and binary embeddings instead become the mean of all of the embeddings from previous layers:

$$\hat{F}_{type} = \frac{\sum_{t=1}^{N_l} F_{type}^t}{N_l}$$

Adding skip connections changes the updating of node hidden states:

$$F_n^t = U_n^t([Agg_n^t(F_n^{t-1},\ C_n),\ \ F_n^{t-1},\ \ F_n^0])$$

FCG Neighborhood Aggregation: For some experiments, we introduce an FCG neighborhood aggregation step to the function hidden state update within each layer.

$$h_f^t = Agg_f^t(F_n^t,\ C_f)$$
$$F_f^t = U_n^t([h_f^t,\ \ F_f^{t-1},\ \ Agg_c(h_f^t,\ C_c)])$$

Node Function Call Information: Node hidden state updates can optionally be supplemented with function call information. When a node calls another function, that function's hidden state from the previous layer is concatenated to the node's values immediately before the update linear layer that produces that node's embedding. If a node does not call a function, we instead concatenate a learned "No Call" embedding E_{nc}. If the node's call location could not be parsed by the binary analysis tool, then we assume the "No Call" embedding. If a node calls multiple locations (e.g. using a call table), then we pick the first function present as the embedding to use.

$$F_n^t = U_n^t([Agg_n^t(F_n^{t-1},\ C_n),\ \ F_n^{t-1},\ \ E_n^t])$$
$$E_n^t = \begin{cases} F_{f,i}^{t-1}, & \text{if the node calls the function at index } i \\ E_{nc}, & \text{if the node does not call any function} \end{cases}$$

Bidirectionality: One improvement we propose in this paper is the use of a bidirectional GNN. Our CFGs are directed, however they are quite sparse (most nodes having < 3 outgoing edges and < 2 incoming edges) leading to few edges for the network to learn. On the other hand, converting to undirected graphs may lose useful directional information for training. We propose a bidirectional GNN which uses two separate layers during each node or function neighborhood aggregation: one to learn (H_f, Agg_{type}^t, U_n^t) - the 'forward' direction following program execution, and one to learn (H_b, $\overline{Agg}_{type}^t, \bar{U}_n^t$) - the 'backward' direction working with the transposed connection information C_{type}^T. These are then averaged to produce the final node/function embedding in that layer.

$$H_f = U_{type}^t([Agg_{type}^t(F_{type}^{t-1},\ C_{type}),\ \ F_{type}^{t-1}])$$
$$H_b = \bar{U}_{type}^t([\overline{Agg}_{type}^t(F_{type}^{t-1},\ C_{type}^T),\ \ F_{type}^{t-1}])$$
$$F_{type}^t = \frac{H_f + H_b}{2}$$

D. Implementation

These models are implemented in Python 3.9 using PyTorch 2.1 and are trained on 2 NVIDIA A100 GPUs.

TABLE I
RQ.1 BASIC BLOCK EMBEDDING METHOD PERFORMANCE
BEST PERFORMANCE IN EACH COLUMN MARKED WITH **bold**.

	Embedding Method					
	Self		Word2vec		BERT	
Normalizer	Acc	F1	Acc	F1	Acc	F1
deepbindiff-inst	0.690	0.672	0.601	0.586	0.614	0.598
deepbindiff-op	0.467	0.464	0.540	0.529	0.558	0.543
deepsemantic-inst	**0.728**	**0.710**	0.633	0.616	0.641	0.626
deepsemantic-op	0.583	0.571	0.565	0.552	0.589	0.574
innereye-inst	0.682	0.668	0.496	0.499	0.540	0.535
innereye-op	0.684	0.667	0.600	0.586	0.610	0.594
safe-op	0.711	0.695	**0.638**	**0.623**	**0.669**	**0.654**

IV. EXPERIMENTS

All experiments (except for **RQ.3**) are performed on both *rand100* and *dist100* datasets combined (**RQ.3** trains on one *or* the other, then tests on both). Models are trained to predict which problem a given CFG is from. There are 8016 'unique' problems with > 1 valid submission in these datasets, creating an 8016-class classification task for our models to solve. The data is split into 80% training set, 10% validation set (on which all **RQ**'s are tested), and a final 10% test set which is used to show final performance. Models are trained for 1500 epochs using 16,000 CFGs per epoch and a batch size of 64.

A. RQ.1 Basic Block Embedding

We test 4 different assembly line normalization methods based on *deepbindiff* [34], *innereye* [35], *safe* [10], and *deepsemantic* [23]. We followed what was outlined in their respective papers as closely as possible, using code as guidance (when available), implementing our own version so they can be tested in a common framework and contain necessary specifications for uncommon assembly lines.

For each of those normalization methods we test 3 embedding methods: a self-learned shallow embedding, pre-trained word2vec-like embeddings, and pre-trained BERT embeddings. See Section III-B for embedding method descriptions.

Finally, for each configuration, we test tokenization both at the instruction level and at the opcode/operand level. When tokenizing at the instruction level, normalizations are applied to all opcodes/operands then all opcodes/operands within an instruction are concatenated together to form the final token. Otherwise, each opcode/operands constitutes its own token post-normalization. We do not test the *safe-inst* method as it produces far too many tokens to be feasible (and, generalizable), generating millions of tokens due to its lax normalization scheme.

For these 21 tests, we used a hidden dimension of 128, 3 GNN layers, and the simple attention mechanism with 8 heads as the aggregation method for all aggregations. All other specialized architecture additions are turned off.

Table I shows the results of these embedding models. Instruction-level tokenization performed better than op-level tokenization for the *deepbindiff* and *deepsemantic* normalization methods, while *innereye* performed better with the op-level tokenization. Across the board, self-learned embedding

TABLE II
RQ.1 NORMALIZER AND EMBEDDING METHOD DATA SIZE (GB)

Normalizer	# Unique Tokens	Embedding Method		
		Self	Word2vec	BERT
deepbindiff-inst	2105	30	833	833
deepbindiff-op	460	76	2900	2900
deepsemantic-inst	17606	30	833	833
deepsemantic-op	723	106	4000	4000
innereye-inst	615799	36	833	833
innereye-op	557	100	3800	3800
safe-op	10556	100	3800	3800

methods had higher or essentially equivalent accuracies and F1-scores than both the word2vec and BERT embedding methods, with less expensive training. The best performing normalization/tokenization technique was *deepsemantic-inst*.

This embedding/normalization method has added benefits as instruction-level tokenization reduces the number of tokens per basic block, and self-embedding requires no pre-training nor embedding of assembly instructions, drastically reducing the storage space (see Table II) and time required to generate data. The *deepsemantic-inst* normalization method maintains a good balance between reducing the vocabulary size while keeping enough useful information for the models to train on.

B. RQ.2 GNN Architecture

We perform ablation studies targeting 1) model architecture and hyperparameters, and 2) complexity of input data. We start with a model using the parameters: hidden dimension of 256, 3 GNN Layers, bidirectionality, simple 8-headed attention mechanism, function call graph embedding aggregations, positional encodings on assembly lines, initial node feature skip connections, jumping knowledge, and node function call information.

Figure 4 shows the ablation study results. Most ablations show a degradation in accuracy and F1-score implying the utility of our model architecture choices.

When looking at the complexity of the input data, we start with the same model as was used at the start of the initial ablation studies (however, using an 'undirected' model). We test three alterations to input data and model architecture: 1) removing graph topology, treating everything as a 'bag-of-nodes' or 'bag-of-functions', 2) removing function information with binary embeddings instead computed from the aggregation of all nodes within that binary, and 3) removing assembly line information forcing the models to rely solely on graph topology.

Table III shows the results of these tests under *Ablations - Dataset Complexity*. The models lose a small amount of performance without function information, a noticeable amount without graph topology information, and most of their performance without assembly line information. This indicates that, while all types of information are useful to the model, proper assembly line data is vital for model performance.

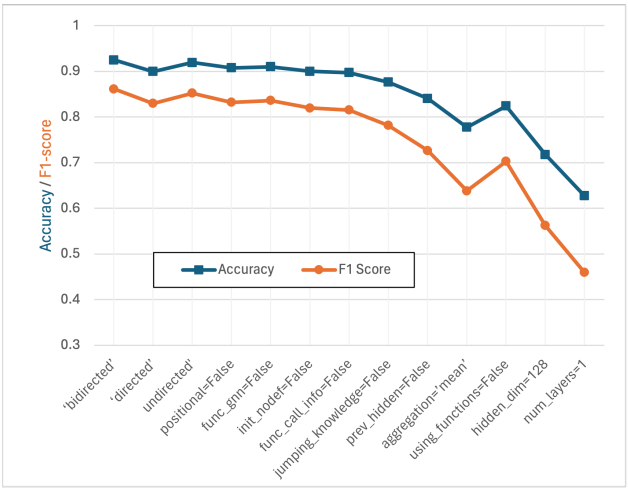

Fig. 4. Ablation studies described in §IV-B. From left to right: the full bidirected model, directed model, undirected model, removing positional encodings, removing function message passing layers, removing node skip connections to initial node embeddings, removing function call embeddings from node updates, turning off jumping knowledge, removing previous hidden states from message passing updates, using averaging as aggregation, removing function information entirely to instead aggregate node embeddings into final binary embeddings, using a hidden dimension of 128, and using a single GNN layer.

C. RQ.3 Dataset Deduplication

Models are trained either on *dist100*, or *rand100*, then tested on the combination of both validation sets for those datasets. These models use the same architecture as the starting model for the ablation studies in Section IV-B.

Table III shows the results of these tests under *Dist100 vs. Rand100*. They have similar accuracies and F1-scores, however models trained on the *dist100* dataset perform slightly better on both test sets joined together. These results show that, while deduplication can help train better models, simply having enough input data and compiling that data in various ways is still enough to produce decently generalizable models.

D. RQ.4 Spatial Embedding and Pre-Training

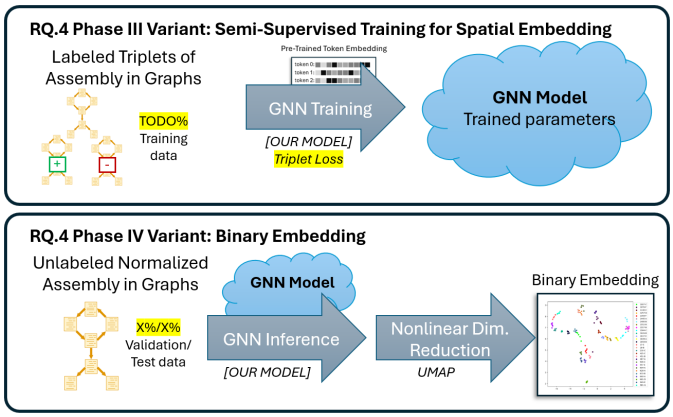

Fig. 5. Proposed work flow for binary embedding.

We start with the same model as was used at the start of the initial ablation studies (however, using an 'undirected' model). This model is pre-trained using a triplet loss function (with margin=1.0) where submissions from the same problem_uid are positive samples, and those from different problem_uid's are negative samples. The model trains for 1000 epochs on both the *dist100* and *rand100* datasets, and are split such that no two examples from the same problem appear in the same set.

On the test set, the model achieves a Silhouette score of 0.1198 indicating a decent amount of clustering with some overlapping clusters present (likely due to the $>$ 8000 clusters that would be present in the training set). The model has an accuracy (percentage of triplets with anchor closer to the positive) of 0.9901 on the test set. We also select some datapoints to plot using UMAP [36] shown in Figure 6. That plot shows excellent grouping of samples together, with the occasional overlap of clusters, as predicted by the Silhouette score.

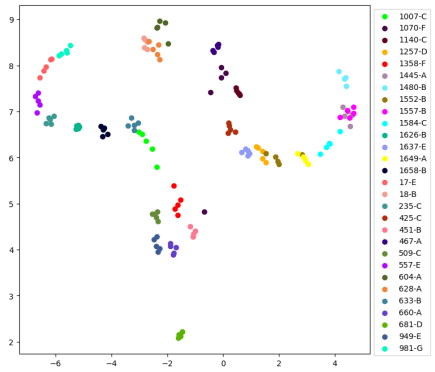

Fig. 6. UMAP of pre-trained embeddings from 30 random problem_uid's, 5 samples per problem

This model was also used as a starting point to fine-tune the problem identification task on. While they had similar performance (0.9198 accuracy, 0.8526 F1-score), the fine-tuned model trained far faster, starting with a jump in validation set accuracy The first epoch to achieve $>$ 80% validation set accuracy was epoch 166, compared to epoch 699 of base models.

E. Final Performance

We test our best performing model on our final test set and achieve and accuracy of 0.9923 and F1-score of 0.9824.

We take that same model architecture and train on the *Big2015* dataset with a 70/30 train/test split and train for 20 epochs. Our model achieves an accuracy of 0.9923 and F1-score of 0.9844. Note that we do include the 'Simda' malware in our tests; some papers remove that example due to it having very few examples in the dataset (42 / 10868). We also rely only on the raw disassembly from the IDAPro output, not including any information on variables, basic blocks, functions, data types, etc. produced from IDAPro. Basic blocks and functions were built manually by checking jump or call locations without any extra static or dynamic analysis. These results are near state-of-the-art with very little effort taken to parse/analyze the binaries, and using only basic CFG information.

TABLE III
MODEL RESULTS ON PROBLEMUID PREDICTION TASK
BEST PERFORMANCE IN APPLICABLE TABLES MARKED WITH BOLD.

Ablations - Dataset Complexity		
Removed Information	Accuracy	F1-Score
graph_topology	0.8536	0.7464
function_information	**0.9053**	**0.8283**
assembly_lines	0.1585	0.0877

Dist100 vs. Rand100		
Trained Dataset	Accuracy	F1-Score
dist100	**0.9281**	**0.8675**
rand100	0.9190	0.8519

Pre-Trained Model		
	Accuracy	F1-Score
pre-trained model	0.9198	0.8526

Final Performance		
	Accuracy	F1-Score
dist100/rand100	0.9256	0.8625

V. RELATED WORK

The highest accuracy model we could find at the time of this paper on the *Big2015* dataset was that of HYDRA [37]. The authors provide an extensive comparison of machine learning methods both in the literature and developed by them with their method having the highest combined accuracy (0.9975) and F1-score (0.9951). Their technique uses a multimodal network for extracting and combining data from various sources including API information, byte sequences, and opcode sequences. Our models are within one percentage point and would rank 4th on their list of 27 state-of-the-art modeling techniques.

VI. CONCLUSION

In this paper, we developed open-sourced tools to aid in the construction of a new, complex binary identification task performed on millions of binaries scraped from the Codeforces [27] programming competition website and compiled various ways. We implemented novel GNN architectures and trained those models on this dataset showing $>$ 92% accuracy on an 8016-class classification problem. Our ablation studies show the utility of our architecture choices including a bidirectional GNN structure, positional encodings applied to assembly lines within each node, hierarchical node $\rightarrow$ function $\rightarrow$ graph aggregations, and skip connections. Our tests on assembly normalization and embedding techniques indicate *deepsemantic* to be an effective normalization scheme, and show no improvement when using pre-trained assembly embedding models as opposed to shallow embeddings learned during GNN training. Ablations involving data input formats show that, while the graph topology and function information are helpful to the models, assembly tokens are pivotal to learning this task. Pre-Training these models to embed binaries by their problem_uid generated distinct clusters defined by binary

behavior, and increased training speed when fine-tuning on the problem_uid prediction task. Finally, we showed that with little effort, our modeling approach can be applied to another common binary analysis benchmark achieving near state-of-the-art performance on the Microsoft Malware Classification Challenge (Big 2015) dataset.

REFERENCES

[1] D. Zhao, H. Lin, L. Ran, M. Han, J. Tian, L. Lu, S. Xiong, and J. Xiang, "Cvsksa: cross-architecture vulnerability search in firmware based on knn-svm and attributed control flow graph," *Software Quality Journal*, vol. 27, no. 3, p. 1045–1068, sep 2019. [Online]. Available: https://doi.org/10.1007/s11219-018-9435-5

[2] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 480–491. [Online]. Available: https://doi.org/10.1145/2976749.2978370

[3] S. Mitra, S. A. Torri, and S. Mittal, "Survey of malware analysis through control flow graph using machine learning," 2023.

[4] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1157–1177, 2017.

[5] A. Jia, M. Fan, W. Jin, X. Xu, Z. Zhou, Q. Tang, S. Nie, S. Wu, and T. Liu, "1-to-1 or 1-to-n? investigating the effect of function inlining on binary similarity analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, may 2023. [Online]. Available: https://doi.org/10.1145/3561385

[6] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Information and Communications Security*, L. Chen, M. D. Ryan, and G. Wang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 238–255.

[7] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based malware detection," in *Engineering Secure Software and Systems*, F. Massacci, D. Wallach, and N. Zannone, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–43.

[8] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary function clustering using semantic hashes," in *2012 11th International Conference on Machine Learning and Applications*, vol. 1, 2012, pp. 386–391.

[9] C. D. Morales-Molina, D. Santamaria-Guerrero, G. Sanchez-Perez, H. Perez-Meana, and A. Hernandez-Suarez, "Methodology for malware classification using a random forest classifier," in *2018 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)*, 2018, pp. 1–6.

[10] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds. Cham: Springer International Publishing, 2019, pp. 309–329.

[11] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, p. 99–116.

[12] D. Tian, X. Jia, R. Ma, S. Liu, W. Liu, and C. Hu, "Bindeep: A deep learning approach to binary code similarity detection," *Expert Systems with Applications*, vol. 168, p. 114348, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417420310332

[13] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," *Proceedings of the AAAI Conference on Artificial Intelligence*, no. 01, pp. 1145–1152, Apr. 2020. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5466

[14] S. Arakelyan, S. Arasteh, C. Hauser, E. Kline, and A. Galstyan, "Bin2vec: learning representations of binary executable programs for security tasks," *Cybersecurity*, vol. 4, 2021. [Online]. Available: https://doi.org/10.1186/s42400-021-00088-4

[15] Y. Wang, P. Jia, C. Huang, J. Liu, P. He, and C.-H. Chen, "Hierarchical attention graph embedding networks for binary code similarity against compilation diversity," *Sec. and Commun. Netw.*, vol. 2021, jan 2021. [Online]. Available: https://doi.org/10.1155/2021/9954520

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[17] K. N. Khan, N. Ullah, S. Ali, M. S. Khan, M. Nauman, and A. Ghani, "Op2vec: An opcode embedding technique and dataset design for end-to-end detection of android malware," *Security and Communication Networks*, vol. 2022, 2022. [Online]. Available: https://doi.org/10.1155/2022/3710968

[18] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 472–489.

[19] M. Q. Li, B. C. Fung, P. Charland, and S. H. Ding, "I-mad: Interpretable malware detector using galaxy transformer," *Comput. Secur.*, vol. 108, no. C, sep 2021. [Online]. Available: https://doi.org/10.1016/j.cose.2021.102371

[20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[21] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T.-Y. Liu, "How could neural networks understand programs?" 2021.

[22] Y. Gui, Y. Wan, H. Zhang, H. Huang, Y. Sui, G. Xu, Z. Shao, and H. Jin, "Cross-language binary-source code matching with intermediate representations," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2022, pp. 601–612. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER53432.2022.00077

[23] H. Koo, S. Park, D. Choi, and T. Kim, "Semantic-aware binary code representation with bert," 2021.

[24] Y. Wang, C. Dong, S. Li, F. Luo, R. Su, Z. Song, and H. Li, "Flowembed: Binary function embedding model based on relational control flow graph and byte sequence," in *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, 2023, pp. 950–957.

[25] G. Liu, X. Zhou, J. Pang, F. Yue, W. Liu, and J. Wang, "Codeformer: A gnn-nested transformer model for binary code similarity detection," *Electronics*, vol. 12, no. 7, 2023. [Online]. Available: https://www.mdpi.com/2079-9292/12/7/1722

[26] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Comput. Surv.*, vol. 54, no. 3, apr 2021. [Online]. Available: https://doi.org/10.1145/3446371

[27] "Codeforces," http://codeforces.com, accessed: 2023-10-11.

[28] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, 2021.

[29] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," 2018.

[30] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[31] J. Allen, "Bincfg (software version 1.0.0)," 2024. [Online]. Available: https://github.com/LLNL/BinCFG

[32] D. J. Quinlan and C. Liao, "The rose source-to-source compiler infrastructure," 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:9663820

[33] J. Allen, "Cap (software version 1.0.0)," 2024. [Online]. Available: https://github.com/LLNL/CAP

[34] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," *Proceedings 2020 Network and Distributed System Security Symposium*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:195063875

[35] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *ArXiv*, vol. abs/1808.04706, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52004699

[36] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," 2020.

[37] D. Gibert, C. Mateu, and J. Planes, "Hydra: A multimodal deep learning framework for malware classification," *Computers & Security*, vol. 95, p. 101873, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404820301462