COBRA-GCN: Contrastive Learning to Optimize Binary Representation Analysis with Graph Convolutional Networks

Anonymous Author(s)

No Institute Given

Abstract. The ability to quickly identify whether two binaries are similar is critical for many security applications, with use cases ranging from triaging millions of novel malware samples, to identifying whether a binary contains a known exploitable bug. There have been many program analysis approaches to solving this problem, however, most machine learning approaches in the last 5 years have focused on function similarity, and there have been no techniques released that are able to perform robust many to many comparisons of full programs. In this paper, we present the first machine learning approach capable of learning a robust representation of programs based on their similarity, using a combination of supervised natural language processing and graph learning. We name our prototype COBRA: Contrastive Learning to Optimize Binary Representation Analysis. We evaluate our model on several different metrics for program similarity, such as compiler optimizations, code obfuscations, and different pieces of semantically similar source code. Our approach outperforms current techniques for full binary diffing, achieving an F1 score and AUC .6 and .12, respectively, higher than BinDiff while also having the ability to perform many-to-many comparisons.

Keywords: Graph Learning · Binary Code · Similarity

1 Introduction

Detecting similar files is often used for tasks such as malware triage [30, 34], patch analysis [23, 26], and bug search [12, 18, 17, 16, 19, 24]. However, identifying binary code similarity is very challenging – much of the program semantics can be lost during the compilation process. Many compiler optimizations such as

function inlining will drastically change the syntax and structure of a binary even with the same source code. Figure 1 shows the *addr2line* program compiled with two different compilers and two different optimizations. As you can see, different compiler settings can cause identical source code to appear dramatically different at the binary level.

There has been a recent explosion in machine learning approaches to solve this issue. According to a recent survey of binary similarity techniques [28], 7/12 of the binary similarity papers since 2018 have been machine learning based, as opposed to 4/28 of the papers in the 3 years prior. However, recent solutions focus on one-to-one comparisons, and do not have the ability to perform efficient one-to-many or many-to-many comparisons of full programs.

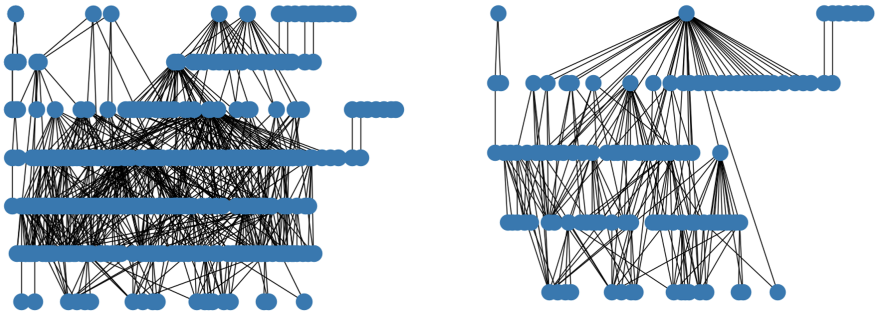

Fig. 1: Addr2line call graph compiled with two different optimizations and compilers. GCC and O0 on the left, Clang and Os on the right.

There are various levels of comparison granularity – some approaches only work at the function level, others only work with full programs. Additionally, there are various levels of comparisons – one-to-one, one-to-many, and many-to-many. One-to-one comparisons are often used for *binary code diffing* – they diff two different programs in order to determine the level of similarity. One-to-many comparisons compare one *query* piece of code to many target pieces of code in order to perform tasks such as bug searches. Many-to-many comparisons do not distinguish between source and target pieces. All inputs are considered equal, and many-to-many approaches often output *clusters* of similar binaries. In order to perform several tasks, such as malware clustering and triage, efficient many-to-many comparisons are needed. While one-to-one approaches could be used pairwise across an entire dataset to perform a one-to-many or many-to-many comparison, most one-to-many and many-to-many approaches avoid this due to inefficiency. None of the recent machine learning techniques are able to perform robust many to many matching at the full program level. We solve this problem by learning an *embedding*, i.e., a vector of numbers, for full programs. Similarity

between two programs can be measured efficiently by taking the distance between their embeddings, allowing for fast one-to-many and many-to-many comparisons.

Approach. We propose a supervised siamese graph convolutional network to learn an embedding for a full program's call graph. Our approach has several stages. First, we learn embeddings for individual assembly instructions. Next, we aggregate these instruction embeddings to generate an embedding for a full function. Finally, we combine these function embeddings with the full program's call graph to generate an embedding for a full program. These embeddings contain information from both the function semantics as well as the full call graph structure. The primary benefit in utilizing function and instruction embeddings is the modularity - we can easily swap out the function and instruction similarity components for newer techniques as the field progresses. A full overview of our approach in action can be seen in Figure 2.

We implement a prototype, COBRA, and conduct an evaluation on a program similarity dataset consisting of both compiler optimizations as well as obfuscations containing more than 8000 binaries. We also evaluate on more than 40,000 binaries from Google Code Jam, an annual programming competition. Binaries from Google Code Jam are different at both the source and binary level, making similarity detection much more difficult. Our evaluation shows that we outperform state of the art binary diffing tool BinDiff as well as locality sensitive hashing algorithms TLSH, SSDeep, SDHash.

Contributions. The contributions of this paper are as follows:

- We propose a supervised algorithm to learn robust representations for full programs. We leverage both natural language processing and graph learning techniques in order to generate high quality embeddings for full call graphs. To the best of our knowledge, it is the first supervised model able to perform many to many comparisons on full programs based on program similarity.
- We implement a prototype, COBRA, which takes as input a full binary and generates an embedding. It starts by learning embeddings for assembly instructions using fastText [9]. Next, it learns embeddings for functions using a bidirectional Recurrent Neural Network. Finally, we train a graph convolutional network [32] on the program's call graph which has been enriched with the previously learned function embeddings to learn an embedding for the full program.
- We show that our approach is able to outperform state-of-the-art binary diffing tools for cross-optimization, obfuscation, and different source code. In addition, we show that our model is capable of performing higher level tasks efficiently, such as clustering.
- We provide a brief comparison of Word2vec [39] and fastText [9]. Word2Vec has been a very popular method of representing instructions in the literature, but we believe fastText provides more robust representations for instructions.

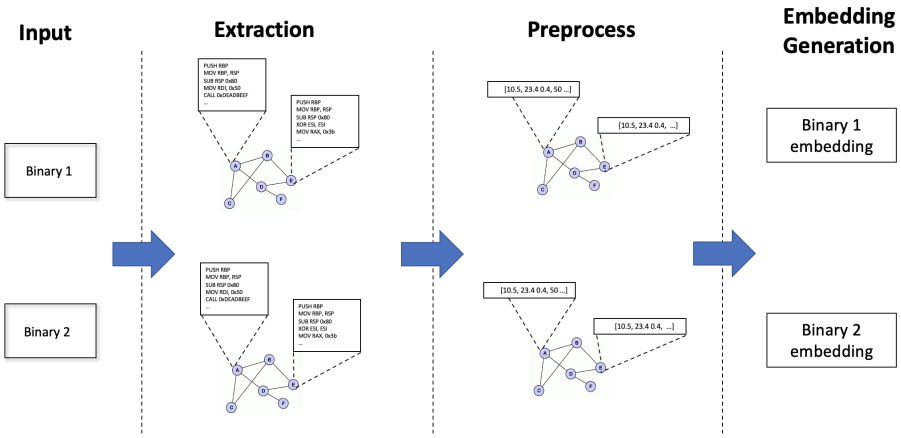

Fig. 2: An overview of COBRA.

2 Related Work

Non-learning based Binary Similarity. One common technique to identify similar files is a similarity digest or "hash" [42, 46, 15, 33, 50]. This technique is commonly known as locality sensitive hashing or fuzzy hashing. These hashing functions convert a large byte string into a smaller, unique string similar to cryptographic hashes such as MD5 or SHA-1. However, locality sensitive hashing algorithms are designed to generate similar hashes for similar inputs. Thus, to quantify the similarity between two files, we can compare the similarity between their respective hashes rather than needing to compare the files themselves. The ability to generate a hash is extremely useful for malware triage, as it is much less computationally expensive to compare hashes and allows analysts to identify similar pieces of malware. Antivirus tools and engines such as VirusTotal [6] can generate the locality sensitive hash for a piece of malware once, and then use that hash in perpetuity to compare that piece of malware to other samples. Despite the widespread adoption of locality sensitive hashing algorithms, their effectiveness has been called into question by Pagani *et al.* [43]. Their results show that fuzzy hashes work well primarily due to matching the *.data* section, as the *.data* section often remains unchanged through compiler optimizations and obfuscations.

Other research approaches perform binary diffing using various data structures. BinDiff [1] performs graph isomorphism detection and matches basic blocks as well as functions. BinGold [7] extracts a novel data structure called a *semantic flow graph* using the data-flow graph as well as the control flow graph and uses several metrics such as graph edit distance for function similarity. Several approaches generate function signatures[54, 51] for cross platform bug searches and patch detection. Another technique uses light emulation to generate traces, and performs a simple jaccard index on the traces of two func-

tions in order to determine their similarity [12, 55, 29]. Additionally, Bingo [12] and Bingo-E [55] perform selective inlining of callee functions. In other words, when their target function calls another function, they inline the assembly of the called function to ensure that it's semantics are captured. Another common technique is to decompose a binary into *strands* - small sequences of instructions after a binary has been lifted into an intermediate representation [18, 17, 16], then find matching strands between two binaries. COP2017 [36] has the ability to perform comparisons of full programs, but is only able to perform one-to-one matching by taking the longest common subsequence of basic blocks.

Learning based Binary Similarity. There have also been many learning-based approaches towards the problem of code diffing and similarity detection. However, many of these over the past five years focus on function similarity. According to Haq *et al.*, only 1/11 learning-based approaches are effective on full programs. DeepBinDiff [22], uses Text Associated DeepWalk [56] to learn embeddings for individual basic blocks. SAFE [38] uses a self-attentive recurrent neural network to generate function embeddings. ASM2Vec [21] learns a representation for functions using PV-DM [40] by converting each function into multiple sequences based on its control flow graph and using selective callee expansion. InnerEye [59] uses neural machine translation to generate embeddings for basic blocks. αdiff [35] combines two function's raw bytes, call graph, and imports to generate a similarity score between the two functions. Gemini [53] converts basic blocks into a vector of handcrafted features and uses Structure2vec [14] on a function's control flow graph to generate an embedding. Trex [45] first learns assembly instruction semantics by pre-training on sequences of assembly instructions similar to BERT [20] before finetuning on function pairs.

Malware Clustering A common use of full program similarity is malware clustering and categorization. Clustering requires the ability to perform many to many comparisons. AMCS [57] uses instruction frequency and function based instruction sequences for categorization and clustering. Rieck *et al.* [49] clusters malware based on the reports generated from executing malware in CWSandbox. EC2 [11] uses ensemble clustering and classification on both static and dynamic features to generate malware clusters. Zhuang *et al.* [58] clusters websites together with malware binaries by using term frequency for websites and instruction frequency for binaries with an ensemble clustering scheme. Firma [47] clusters based on network traffic after executing malware in a sandbox. Bayer *et al.* [8] uses taint tracing to generate an execution trace for a piece of malware after executing in a sandbox, and then clusters with Jaccard Distance and Cosine Similarity.

3 Approach

3.1 Approach Overview

The full system can be broken down into the following stages. Each stage requires separate preprocessing before training:

1. *Assembly Instruction Embeddings*: We leverage fastText to learn embeddings to encapsulate similarities between single instructions.
2. *Function Embeddings*: We feed in a function represented as a sequence of instruction embeddings to a bidirectional recurrent neural network in order to generate embeddings for functions. Our approach is based largely off of SAFE[38].
3. *Call Graph Embeddings*: We first annotate each vertex of the program's call graph with the embedding for that function. We then feed this annotated call graph into a siamese graph convolutional network in order to generate embeddings for full programs.

3.2 Assembly Instruction Embeddings

In the first stage of our system, we learn an embedding $\overrightarrow{i}$ for each instruction i such that $\overrightarrow{i} \in \mathbb{R}^n$.

Preprocessing. Before training, we preprocess all instructions to reduce the vocabulary size. We process instructions very similarly to SAFE[38]. We replace all base memory addresses with the special symbol *MEM* and all immediate values whose absolute value is above 500 with the special symbol *IMM*. We then concatenate the mnemonic and operands into a single string after performing this filtering. Several examples are detailed below.

$$\textit{MOV EAX, 600} \rightarrow \textit{MOVEAXIMM}$$

$$\textit{MOV EAX, [0xdeadbeef]} \rightarrow \textit{MOVEAXMEM}$$

$$\textit{MOV EAX, [EBP+4]} \rightarrow \textit{MOVEAX[EBP+4]}.$$

We implement this step using Ghidra to extract instructions before preprocessing.

fastText. Almost all machine learning models which generate instruction level embeddings use Word2vec [22, 38, 21, 59, 48]. However, Duan *et al.* showed in DeepBinDiff [22] that a Word2vec model trained on CoreUtils v8.29 compiled with gcc could only generate embeddings for 78.37% of instructions on CoreUtils v8.29 compiled with clang. In other words, 21.63% of instructions cannot be modeled on the same source code when a different compiler is used. DeepBindiff and some other algorithms solve this by modeling opcodes and operands separately. However, we feel that this limits the vocabulary size too much. In order to remedy these problems, we use fastText. We find that fastText offers both improved performance as well as the ability to generate embeddings for words that are not in the vocabulary. To the best of our knowledge, this is the first paper to use fastText to generate assembly instruction embeddings.

The primary difference between fastText and Word2vec is the use of character n-grams. Word2vec treats each word as an atomic unit, and can only learn embeddings for complete words. *MOV EAX 0x1* is treated as a completely separate word to *MOV EAX 0x2* despite being very similar, both using the same

opcode and operand. However, fastText captures this relationship by learning embeddings for the n-grams that are within a word. For example, the word *MOVEAX1* and $n = 3$ can be broken down into the following ngrams:

$$<MO,\ MOV,\ OVE,\ VEA,\ EAX,\ AX1,\ X1>$$

as well as the special sequence

$$<MOVEAX1>.$$

The < and > signs are to denote the beginning and end of the sequence. For fastText, the distance between two words is an aggregate metric between their n-grams, as opposed to treating the entire word as an atomic unit. We find this beneficial for two reasons:

1. Semantically similar instructions will often have the same mnemonics and registers. In the example above, *MOV EAX 0x2* has many of the same n-grams as *MOV EAX 0x1* that are captured, whereas Word2vec treats them both as completely separate units. While there are cases in which Word2vec performs better, such as *MOV RAX 0* and *XOR RAX, RAX*, we have anecdotally observed these to be less common.
2. fastText is able to generate embeddings for out-of-vocabulary words by taking the seen n-grams of the word. Because Word2vec treats words as an atomic unit, it cannot generate embeddings for words that is has not seen during training. Given that there are a very large number of potential assembly instructions and the generated assembly instructions are often dependent on compilers, we find this to be an important characteristic.

3.3 Function Embeddings

In the second stage of our system, we learn an embedding $\overrightarrow{f}$ for each function f such that $\overrightarrow{f} \in \mathbb{R}^n$. We represent each function f as a sequence of instruction embeddings $\overrightarrow{i}$, and feed that into a bidirectional recurrent neural network with a siamese architecture. Using this, we are able to generate similarity-preserving embeddings which capture the semantic behavior of functions.

Siamese Architecture. The embedding model parameters are learned using a pairwise approach and a siamese architecture. At training time, the model is fed two functions $f1$, $f2$ separately, and outputs an embedding for each function. The functions can then be determined to be similar or different by calculating their cosine similarity with the formula below:

$$\cos(\overrightarrow{f1}, \overrightarrow{f2}) = \frac{\overrightarrow{f1}\overrightarrow{f2}}{\|\overrightarrow{f1}\|\|\overrightarrow{f2}\|} = \frac{\sum_{i=1}^{n} \overrightarrow{f1}_i \overrightarrow{f2}_i}{\sqrt{\sum_{i=1}^{n} (\overrightarrow{f1}_i)^2}\sqrt{\sum_{i=1}^{n} (\overrightarrow{f2}_i)^2}} \tag{1}$$

The network is trained using function pairs $<\overrightarrow{f1},\ \overrightarrow{f2}>$, and labelled with ground truth $y_i \in \{+1, -1\}$. To train the network, we use the contrastive loss function proposed by Hadsell *et al.* [27] below:

$$L(Y, \overrightarrow{f1}, \overrightarrow{f2}) = (1 - Y)\frac{1}{2}(D_w)^2 + (Y)\frac{1}{2}\{max(0, m - D_w\})^2 \tag{2}$$

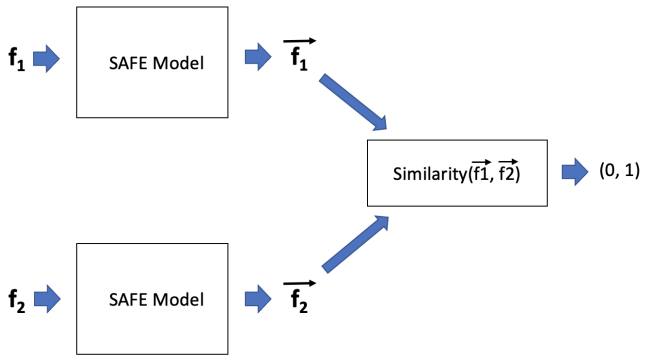

Fig. 3: The SAFE embedding network.

where D_w is defined as the cosine similarity between $\overrightarrow{f1}$ and $\overrightarrow{f2}$. m is the margin, meaning that dissimilar pairs only contribute to the loss if their euclidean distance is within this radius. An overview of SAFE can be seen in Figure 3.

3.4 Program Embeddings

Finally, we learn an embedding $\overrightarrow{p}$ for each program p such that $\overrightarrow{p} \in \mathbb{R}^n$. In order to represent the entire program, we use the call graph. We represent each program p as two matrices using it's call graph. The first is an $N \times D$ matrix, where N is the number of functions, and D is the length of features associated with each node. The feature for each node is the function embedding for that node, of length 100 in our case. The second matrix is an adjacency matrix of the call graph.

Intuition. Two similar programs will often have similar call graph structures. There have been multiple efforts to using call graphs in order to determine program similarity [52, 35]. However, using only the call graph structure does not encapsulate the contents of the functions themselves. Likewise, taking a purely function-based approach for binary similarity neglects information provided by the structure of the program as a whole. We aim to combine these two approaches by using a graph convolutional network. By annotating each node in the call graph with the embedding for that function, each node contains information about the semantics of that particular function. By passing in the full call graph structure, our model is able to learn from the holistic functionality of the program in order to generate full program embeddings. We then train a graph convolutional network, a model which learns from both node features and graph structure, in order to learn embeddings for full programs.

Graph Convolutional Networks. Traditional convolutional neural networks operate under the assumption that the input is grid-structured as opposed to arbitrarily structured graphs. However, Bruna *et al.* generalized convolutional

networks to graphs, applying filters on a graph's frequency modes computed by graph Fourier transform [10]. Since then, graph convolutional networks have evolved and been used for many graph-based data problems, such as protein function prediction [25] and fake news detection [41]. Broadly speaking, graph convolutional networks learn a set of features from a graph taking as input:

- A feature vector x_i for every node i in the form of an $N \times D$ matrix, where N is the number of nodes and D is the number of features.
- The graph structure in matrix form, typically the adjacency matrix.

and outputs an $N \times F$ feature matrix, where F is the number of output features per node. We can then use normal pooling and fully connected layers across the full graph to train a siamese-style network. The high-level siamese architecture of this network is very similar to that of the function embedding network.

4 Datasets

We use four different datasets to train and evaluate our models. We refer to them as the *Android NDK* [3], *X86-SOK* [44], *VCPKG* [5], and *Google Code Jam* [4]. Each dataset contains binaries and functions that appear diverse but perform the same functionality. Our datasets are diversified primarily using compiler optimizations and obfuscations. We remove all duplicate functions as well as functions that do not have names.

Android NDK. The Android NDK is a toolset that allows developers to implement parts of their apps using native-code languages such as C and C++ in order to interact more directly with the kernel or the hardware. It includes a number of example programs for demonstration purposes that we use as a dataset for evaluation. We can compile these while introducing obfuscations from Obfuscator-LLVM [31] to generate software diversity. The obfuscations used are bogus control flow (BCF), instruction substitution (SUB), and control flow flattening (FLA) for a total of 2632 binaries and 27441 functions.:

- **BCF** obfuscates the program by modifying the control flow graph. It adds a large number of irrelevant control flow and basic blocks as well as merging and reordering existing blocks.
- **SUB** replaces standard binary operators with functionally equivalent sequences of instructions. For example, addition can be rewritten as $a = b - (-c)$, and subtraction can be written as $r = rand(); a = b + r; a = a - c; a = a - r$
- **FLA** completely flattens the control flow graph of a function by replacing conditional statements with switches, as well as modifying the instructions for entering and exiting basic blocks.

We can combine these in order to generate heavily obfuscated programs as well as varying the architecture (X86, X86-64).

X86-SOK. We also use the Linux binaries from the dataset provided by Pang *et al.* [44]. It is a dataset compiled with different compilers (GCC-8.1.0, LLVM

6.0.0) and optimizations (O1, O2, O3, O4, Os, Ofast), resulting in 3342 distinct binaries and 971,305 functions. Below are summaries of the optimization levels. More details on the optimization levels can be found on their website [2].

- *O0*: Reduces compilation time and ensures debugging information produces the correct results.
- *O1*: The compiler tries to reduce code size and execution time while minimizing compilation time.
- *O2*: The compiler performs all optimizations that do not involve a space-speed tradeoff. As opposed to *O1*, *O2* increases compilation time.
- *O3*: A superset of *O2* with even more aggressive optimizations enabled.
- *Os*: The compiler enables all optimizations except those that increase the code size.
- *Ofast*: The compiler enables all optimizations from *O3*, as well as additional options to increase speed.

VCPKG. VCPKG is a tool developed by Microsoft that acts as a package manager for various open-source libraries written in C and C++. We downloaded a dataset of 446 programs and compiled with MSVC with various optimizations(O1, O2, Od, Ox, Os) for a total of 2230 binaries and 20970 functions. All programs are 64 bit. Below are summaries of the optimization levels. More details about the optimizations can be found on their website [13].

- *O1*: The compiler creates the smallest code size possible.
- *O2*: The compiler optimizes for maximum speed.
- *Od*: The compiler disables all optimization.
- *Os*: The compiler favors optimizations that reduce size. This is a subset of *O1*.
- *Ox*: The compiler favors optimizations that increase speed. This is a subset of *O2*.

Google Code Jam. Google Code Jam is a programming competition run by Google each year. They are an interesting test case for program similarity - programs that solve the same problem have similar semantics and functionality, but are implemented differently by different authors. Unlike the previous datasets, these samples are different at both the source and binary level. However, because these binaries are much smaller and are meant to only be run from the command line, we consider these samples separate from our other three datasets. There are a total of 41573 unique solutions to 222 problems. Several examples of problems are listed below.

- Given a string of digits S, insert a minimum number of opening and closing parentheses into it such that the resulting string is balanced and each digit d is inside exactly d pairs of matching parentheses.
- Given a number $\mathbf{N}$ where $\mathbf{N}$ contains at least one digit that is a *4*, find two numbers A and B such that neither A nor B contains any digit that is a 4, and $A + B = \mathbf{N}$.

5 Evaluation and Results

5.1 Assembly Embeddings

The assembly instructions are difficult to evaluate objectively, as there does not exist a centralized dataset for comparing assembly instruction similarity. To show how fastText outperforms Word2vec, we generate a small dataset of 5 classes with 10 instructions each, with hand-written labels according to their semantic similarity. We then generate the corresponding embeddings for each, and plot it using *t-SNE* [37], a useful tool for dimensionality reduction. The five classes are *Subtraction instructions, Addition Instructions, XOR instructions, MOV instructions, Stack Operations.*

To evaluate, we generated all pairwise instruction comparisons between our dataset. As seen in a ROC curve in Figure 4, fastText has a substantially higher AUC of *.93* compared to Word2vec's *.59*, as well as higher precision and recall as shown in Table 1. However, It is important to note that there are certain instructions that Word2Vec performs better on, such as *JZ MEMORY* and *JNZ MEMORY* which are both JUMP instructions. This is unsurprising, as they do not share ngrams. We leave a more robust evaluation of fastText against Word2vec with a larger dataset for future work.

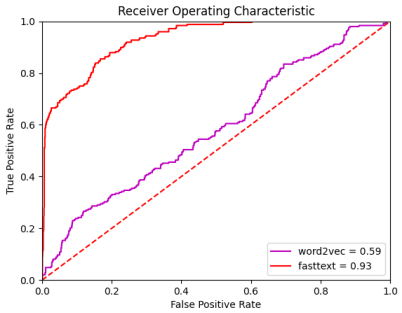

Fig. 4: ROC Curve of fastText compared to Word2vec.

Model	Precision	Recall	F1
fastText	.50	.84	.63
Word2vec	.19	.54	.28

Table 1: Precision and Recall of W2V vs. FastText

5.2 Function Embeddings

Android NDK, X86-SOK, VCPKG. We train our network on pairs of functions. For every function in our dataset, we generate a similar pair labeled as +1 and a dissimilar pair labeled as -1. A similar pair is the same function either compiled with a different optimization or a different obfuscation. A dissimilar pair is a different function entirely. This results in a total number of pairs twice that of our total number of functions. We test our performance on all three datasets

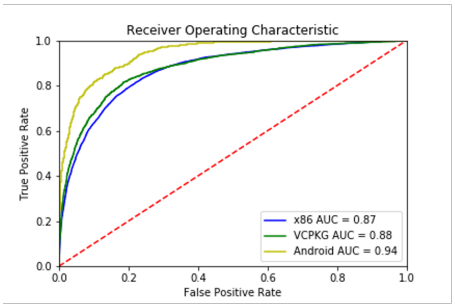

Fig. 5: ROC Curves on each of our function datasets.

Dataset	Precision	Recall	F1
Android	.852	.867	.859
X86-SOK	.801	.778	.793
VCPKG	.815	.814	.815

Table 2: Precision and Recall on each of our function datasets.

separately to ensure our model is able to learn all three, using a 90-10 train-test split to train and evaluate our models. It is important to note that we split our training and test set based on classes before generating the pairs. For example, our dataset might consist of $f1, f2, f3, f4, f5$, where each function has several different versions. We train on pairs generated from functions $f1, f2, f3$ and test on pairs generated from $f4, f5$. We do not evaluate our function embedding model against other state-of-the-art approaches, as our goal is not to outperform all function similarity models, but to outperform full program similarity models. However, we are able to see in Table 2 as well as in Figure 5 that our models are able to generate embeddings which capture function similarity, achieving F1 scores of *.859, .793, and .815* on the Android, X86-SOK, and VCPKG datasets respectively.

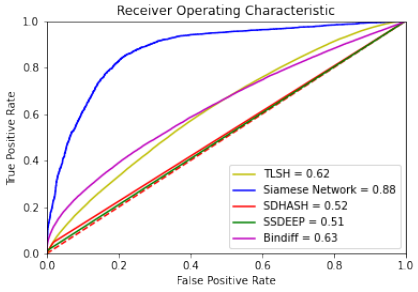

Fig. 6: ROC Curves from Google Code Jam functions.

Google Code Jam. We also train and evaluate our function similarity model on the functions extracted from Google Code Jam and compare to some existing approaches. We find that our model is able to outperform some common tools used for program similarity, achieving an AUC of .88 as compared to the next

highest of .63 by BinDiff. This suggests that machine learning models are capable of learning a higher level of semantic similarity than existing tools. The functions may not be noticeably similar at the assembly level, making it hard for tools such as *TLSH* which rely on heuristics such as shared instruction ngrams. A ROC curve can be seen in Figure 6.

5.3 Program Similarity

For the full program similarity embeddings, we use a similar approach to our function embeddings. For each program, we extract the call graph, and annotate each node with the function embedding for that node. Then, for each call graph in our dataset, we generate similar pairs labelled as +1 and dissimilar pairs labelled as -1. Similar to learning our function embeddings, we split our training and test set based on classes before generating the pairs. We perform multiple experiments to test the robustness of our final embeddings:

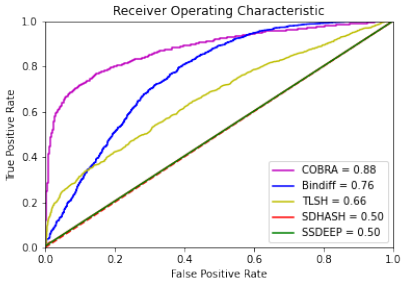

Fig. 7: ROC Curve for COBRA compared to Bindiff, TLSH, SDHASH, and SSDEEP for samples from Coreutils.

Tool	Precision	Recall	F1
ssdeep	.33	1	.49
sdhash	.33	.999	.49
tlsh	.46	.54	.60
Bindiff	.81	.81	.81
COBRA	.88	.86	.87

Table 3: Precision and Recall for Coreutils.

- We perform 10-fold Cross Validation using the X86-SOK dataset. We report results from the entire dataset, as well as isolating the test samples from GNU Coreutils. We select Coreutils as it is difficult for binary similarity as the programs share large amounts of library functions.
- We train on 90% of the X86-SOK dataset, validate on 10% of the X86-SOK dataset, and test on VCPKG and OLLVM obfuscations
- We perform a 90-10 train-test split on Google Code Jam.
- COBRA is the first tool able to generate many-many comparisons of full programs based on similarity, allowing for additional applications such as clustering. We showcase this using a dimensionality reduction algorithm t-SNE.

Our final model had two convolutional layers with sizes 64 and 32 with reLU activation functions, followed by two dense layers of size 256 and 128 with reLU

activation functions. The final layer outputs the final embedding. The model is trained with contrastive loss and optimized with RMSProp on a Volta V100 GPU with 32GB memory. We evaluate against Bindiff, TLSH, SDHash, and SSDeep. We would have liked to evaluate against COP2017 [36] but the authors did not respond to our email requesting to obtain the tool.

Tool	Metric	O0 O1	O0 O2	O0 O3	O0 Os	O0 Of	O1 O2	O1 O3	O1 Os	O1 Of	O2 O3	O2 Os	O2 Of	O3 Os	O3 Of	Os Of	Avg
ssdeep	*Precision*	.54	.55	.54	.49	.46	.52	.53	.53	.50	.54	.50	.52	.48	.49	.51	**.51**
	Recall	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	**1.0**
	F1	.70	.71	.70	.66	.63	.69	.69	.69	.67	.70	.66	.69	.65	.66	.67	**.68**
sdhash	*Precision*	.54	.55	.54	.49	.47	.52	.53	.53	.50	.54	.50	.52	.48	.49	.51	**.51**
	Recall	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	**1.0**
	F1	.70	.71	.70	.66	.63	.69	.69	.69	.67	.70	.66	.69	.65	.66	.67	**.68**
TLSH	*Precision*	.87	.93	.85	.91	.84	.67	.76	.84	.85	.86	.77	.89	.89	.88	.69	**.83**
	Recall	.49	.59	.57	.50	.59	.77	.81	.68	.70	.60	.53	.60	.51	.74	.86	**.64**
	F1	.63	.72	.68	.65	.69	.72	.78	.75	.77	.71	.63	.71	.65	.80	.77	**.71**
BinDiff	*Precision*	.83	.85	.85	.80	.84	.80	.82	.79	.81	.79	.77	.87	.83	.77	.76	**.81**
	Recall	.80	.73	.71	.88	.76	.87	.84	.89	.85	.89	.92	.61	.80	.93	.94	**.83**
	F1	.82	.78	.77	.84	.80	.83	.83	.84	.83	.84	.84	.72	.81	.84	.84	**.81**
COBRA	*Precision*	.93	.86	.84	.85	.95	.92	.87	.81	.87	.84	.89	.84	.88	.91	.93	**.88**
	Recall	.90	.97	.93	.93	.78	.94	.95	.93	.96	.90	.93	.93	.91	.91	0.91	**.92**
	F1	.91	.91	.88	.89	.86	.93	.91	.87	.91	.87	.91	.88	.89	.91	.92	**.90**

Table 4: Detailed precision and recall across varying levels of optimization.

X86-SOK. For the X86-SOK dataset, we use 10-fold Cross Validation, and generated one positive pair and one negative pair during training. During testing, we generated two positive pairs and two negative pairs for each sample to ensure that we had sufficient results. We compared our approach to four existing techniques for binary similarity. We have two scenarios for testing.

– In the first scenario, we aggregate our results over each of our testing splits during our 10-fold cross validation. The ROC curve is shown in Figure 9. We get .94 AUC, outperforming the next highest of .90 with Bindiff. The precision and recall of each comparison is shown in Table 4. One interesting result is TLSH's relatively high precision rate. This is likely because it

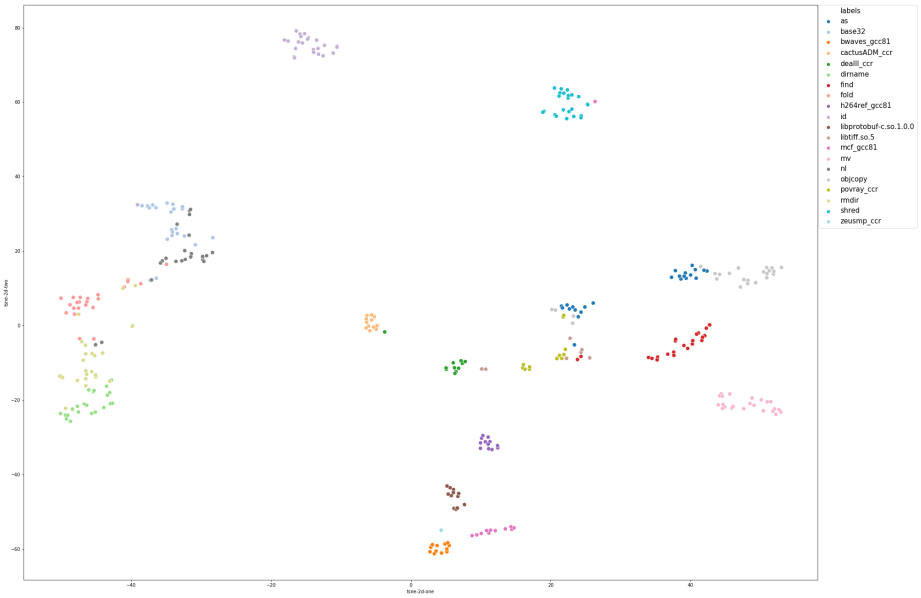

Fig. 8: Final Embeddings of X86-SOK dataset with t-SNE.

is very uncommon for two different programs to share a significant number of instruction n-grams. However, the recall rate is relatively low given that compiler optimizations can cause significant perturbations at times. We also showcase the many-to-many capabilities of our tool by generating a visualization with t-SNE, shown in Figure 8. We can see that many of the classes are in visibly distinct clusters.

– In the second scenario, we only test on Coreutils. The ROC curve is shown in Figure 7. We get .88 AUC, outperforming the next highest of .76 again with BinDiff. The precision and recall is shown in Table 3. COBRA is still able to detect similarity despite Coreutils being notably harder than the rest of the dataset due to the amount of shared code.

Evaluation on Android and VCPKG. A concern of ours was that COBRA was simply memorizing the transformations in our datasets and would be unable to predict any new transformations or obfuscations in the future. In order to ensure that COBRA is capable of generalizing to new, unseen transformations, we train our model on 90% of the X86-SOK dataset, then test on the OLLVM obfuscations as well as Windows binaries. Details of these datasets can be seen in Section 4. This ensures that at testing time, the model has to predict whether two binaries are the same even when unseen transformations are applied. Precision, recall, and F1 can be seen in Tables 5 and 6. ROC curves can be seen in Figures 10 and 11. Even evaluated against unseen transformations, COBRA is able outperform all one-to-many and many-to-many approaches, and is only

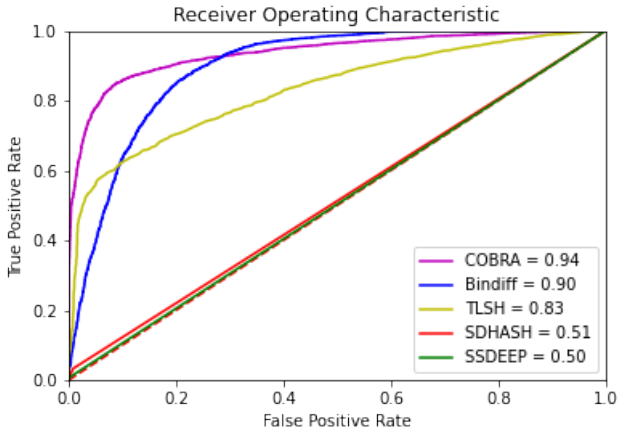

Fig. 9: ROC Curve for COBRA compared to Bindiff, TLSH, SDHASH, and SSDEEP across the entire X86-SOK dataset.

slightly worse than BinDiff. It is important to note that BinDiff has a significant advantage over COBRA in that it takes in two programs and directly computes a similarity score, whereas COBRA must learn a generic representation for each program, which is a significantly harder task.

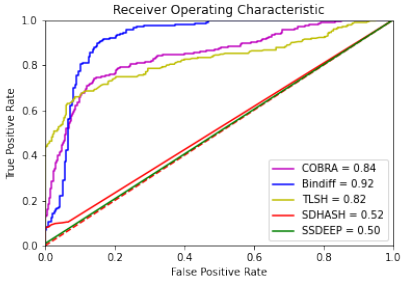

Fig. 10: ROC curve for COBRA compared to BinDiff, TLSH, SDHASH, and SSDEEP across the Android NDK compiled with OLLVM Obfuscations.

Tool	Precision	Recall	F1
ssdeep	.46	1	.63
sdhash	.46	.999	.63
tlsh	.87	.66	.75
COBRA	.77	.86	.81
BinDiff	.86	.91	.89

Table 5: Detailed COBRA Precision and Recall after being trained on X86-SOK and evaluated on Android.

Google Code Jam. Finally, we train and evaluate our model on a selection of 41573 solution files to 222 problems from Google Code Jam. We train it in a similar fashion, except similar samples are two solutions to the same problem, and dissimilar samples are two solutions to different problems. No special inlining

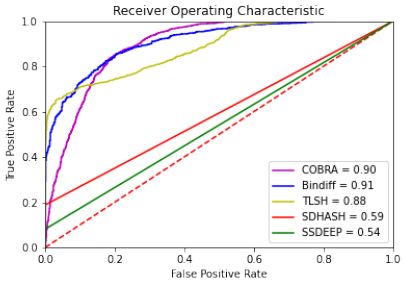

Fig. 11: ROC curve for COBRA compared to BinDiff, TLSH, SDHASH, and SSDEEP on VCPKG compiled with MSVC.

Tool	Precision	Recall	F1
ssdeep	.39	1	.64
sdhash	.48	.999	.64
tlsh	.93	.65	.77
BinDiff	.88	.85	.86
COBRA	.83	.82	.82

Table 6: Detailed COBRA Precision and Recall after being trained on X86-SOK and evaluated on vcpkg.

or preprocessing was performed with this dataset. A ROC curve can be seen in Figure 12, where we achieve .77 AUC, outperforming the next highest of BinDiff at .66. Precision and recall can be seen in Table 7, where we achieve an F1 score of .73, outperforming BinDiff's score of .61. It is worth noting that SSDeep and SDHash have the next highest F1 scores by outputting a positive prediction every time.

Predicting whether two binaries are solutions to the same problem is much harder than our previous experiments using compiler optimizations and obfuscations, as the code is no longer identical at the source level. The performance of all tools is notably lower. This is likely both due to the inherent difficulty of the problem as well as the size of the binaries. The solution is implemented in a couple of functions at most, and so binaries are primarily comprised of compiler intrinsics and library functions, making it much harder to differentiate between classes at the binary level. However, we still believe these are promising results that our model can distinguish between high levels of semantic similarity at the binary level. Future work will address how we can improve our technique and training to better handle cases of semantic similarity, as seen in the Google Code Jam dataset.

6 Limitations and Future Work

There are some limitations to our current approach. First, we are only able to handle the x86 and x86-64 architectures. Second, it has very limited interpretability. Given an embedding for a program, it is very difficult to know why the model generated that embedding. It will be interesting to incorporate some interpretability work into our model. Finally, our performance on Google Code Jam has room for improvement.

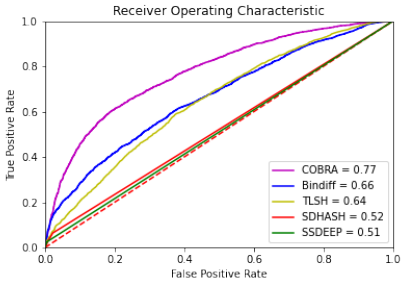

Fig. 12: Final ROC curve for full Google Code Jam programs.

Tool	Precision	Recall	F1
ssdeep	.51	.999	.67
sdhash	.51	.999	.67
tlsh	.61	.59	.60
BinDiff	.62	.60	.61
COBRA	.68	.80	.73

Table 7: Google Code Jam precision and recall

7 Conclusion,

In this paper, we propose a novel graph learning algorithm to learn representations for full programs that are robust to syntactic changes. We start by learning function embeddings using natural languge processing techniques, and then train a graph convolutional network over the program's full call graph while incorporating the function embeddings. We have found that our model outperforms current approaches on detecting the same code with various optimizations and transformations applied, and on detecting semantically similar source code written by different programmers.

References

1. https://www.zynamics.com/bindiff.html
2. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
3. Android ndk. https://github.com/android/ndk-samples, accessed: 2010-09-30
4. Google code jam. https://codingcompetitions.withgoogle.com/codejam, accessed: 2010-09-30
5. Vcpkg. https://github.com/microsoft/vcpkg, accessed: 2010-09-30
6. Virustotal. https://virustotal.com, accessed: 2010-09-30
7. Alrabaee, S., Wang, L., Debbabi, M.: Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs)
8. Bayer, U., Comparetti, P.M., Hlauschek, C., Krügel, C., Kirda, E.: Scalable, behavior-based malware clustering
9. Bojanowski, P., Grave, E., Joulin, A., Mikolov, T.: Enriching word vectors with subword information
10. Bruna, J., Zaremba, W., Szlam, A., Lecun, Y.: Spectral networks and locally connected networks on graphs
11. Chakraborty, T., Pierazzi, F., Subrahmanian, V.S.: Ec2: Ensemble clustering and classification for predicting android malware families. https://doi.org/10.1109/TDSC.2017.2739145

12. Chandramohan, M., Xue, Y., Xu, Z., Liu, Y., Cho, C.Y., Tan, H.B.K.: Bingo: Cross-architecture cross-os binary search. https://doi.org/10.1145/2950290.2950350
13. Corob-Msft: /o options (optimize code), https://docs.microsoft.com/en-us/cpp/build/reference/o-options-optimize-code?view=msvc-160
14. Dai, H., Dai, B., Song, L.: Discriminative embeddings of latent variable models for structured data, http://arxiv.org/abs/1603.05629
15. Damiani, E., di Vimercati, S.D.C., Paraboschi, S., Samarati, P.: An open digest-based technique for spam detection.
16. David, Y., Partush, N., Yahav, E.: Firmup: Precise static detection of common vulnerabilities in firmware. https://doi.org/10.1145/3173162.3177157
17. David, Y., Partush, N., Yahav, E.: Similarity of binaries through re-optimization. https://doi.org/10.1145/3062341.3062387
18. David, Y., Partush, N., Yahav, E.: Statistical similarity of binaries. https://doi.org/10.1145/2980983.2908126
19. David, Y., Yahav, E.: Tracelet-based code search in executables. https://doi.org/10.1145/2666356.2594343
20. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding, http://arxiv.org/abs/1810.04805
21. Ding, S.H.H., Fung, B.C.M., Charland, P.: Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. https://doi.org/10.1109/SP.2019.00003
22. Duan, Y., Li, X., Wang, J., Yin, H.: Deepbindiff: Learning program-wide code representations for binary diffing. https://doi.org/10.14722/ndss.2020.24311
23. Dullien, T.: Graph-based comparison of executable objects
24. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E.: discovre: Efficient cross-architecture identification of bugs in binary code
25. Fout, A., Byrd, J., Shariat, B., Ben-Hur, A.: Protein interface prediction using graph convolutional networks
26. Gao, D., Reiter, M., Song, D.: Binhunt: Automatically finding semantic differences in binary programs
27. Hadsell, R., Chopra, S., LeCun, Y.: Dimensionality reduction by learning an invariant mapping. https://doi.org/10.1109/CVPR.2006.100
28. Haq, I.U., Caballero, J.: A survey of binary code similarity. https://doi.org/10.1145/3446371
29. Hu, Y., Zhang, Y., Li, J., Gu, D.: Binary code clone detection across architectures and compiling configurations. https://doi.org/10.1109/ICPC.2017.22
30. Jang, J., Brumley, D., Venkataraman, S.: Bitshred: feature hashing malware for scalable triage and semantic analysis
31. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-llvm – software protection for the masses. https://doi.org/10.1109/SPRO.2015.10
32. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks
33. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing
34. Lakhotia, A., Walenstein, A., Miles, C., Singh, A.: Vilo: a rapid learning nearest-neighbor classifier for malware triage
35. Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A., Zou, W.: α diff: Cross-version binary code similarity detection with dnn. https://doi.org/10.1145/3238147.3238199

36. Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S.: Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. https://doi.org/10.1109/TSE.2017.2655046
37. van der Maaten, L., Hinton, G.: Viualizing data using t-sne
38. Massarelli, L., Luna, G.A.D., Petroni, F., Querzoni, L., Baldoni, R.: Safe: Self-attentive function embeddings for binary similarity
39. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space
40. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J.: Distributed representations of words and phrases and their compositionality
41. Monti, F., Frasca, F., Eynard, D., Mannion, D., Bronstein, M.M.: Fake news detection on social media using geometric deep learning, http://arxiv.org/abs/1902.06673
42. Oliver, J., Cheng, C., Chen, Y.: Tlsh–a locality sensitive hash
43. Pagani, F., Dell'Amico, M., Balzarotti, D.: Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis
44. Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., Xu, J.: Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask
45. Pei, K., Xuan, Z., Yang, J., Jana, S., Ray, B.: Trex: Learning execution semantics from micro-traces for binary similarity, https://arxiv.org/abs/2012.08680
46. Raff, E., Nicholas, C.: Lempel-ziv jaccard distance, an effective alternative to ssdeep and sdhash
47. Rafique, M.Z., Caballero, J.: Firma: Malware clustering and network signature generation with mixed network behaviors
48. Redmond, K., Luo, L., Zeng, Q.: A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis
49. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning
50. Roussev, V.: Data fingerprinting with similarity digests
51. Shirani, P., Wang, L., Debbabi, M.: Binshape: Scalable and robust binary library function identification using function shape
52. T. Kim, Y. R. Lee, B.K., Im, E.G.: Binary executable file similarity calculation using function matching
53. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection, http://arxiv.org/abs/1708.06525
54. Xu, Y., Xu, Z., Chen, B., Song, F., Liu, Y., Liu, T.: Patch based vulnerability matching for binary programs
55. Xue, Y., Xu, Z., Chandramohan, M., Liu, Y.: Accurate and scalable cross-architecture cross-os binary code search with emulation. https://doi.org/10.1109/TSE.2018.2827379
56. Yang, C., Liu, Z., Zhao, D., Sun, M., Chang, E.Y.: Network representation learning with rich text information
57. Ye, Y., Li, T., Chen, Y., Jiang, Q.: Automatic malware categorization using cluster ensemble. https://doi.org/10.1145/1835804.1835820
58. Zhuang, W., Ye, Y., Chen, Y., Li, T.: Ensemble clustering for internet security applications. https://doi.org/10.1109/TSMCC.2012.2222025
59. Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., Zhang, Z.: Neural machine translation inspired binary code similarity comparison beyond function pairs. https://doi.org/10.14722/ndss.2019.23492