Cybersecurity

RESEARCH Open Access

Bin2vec: learning representations of binary executable programs for security tasks

Shushan Arakelyan*, Sima Arasteh, Christophe Hauser, Erik Kline and Aram Galstyan

Abstract

Tackling binary program analysis problems has traditionally implied manually defining rules and heuristics, a tedious and time consuming task for human analysts. In order to improve automation and scalability, we propose an alternative direction based on distributed representations of binary programs with applicability to a number of downstream tasks. We introduce Bin2vec, a new approach leveraging Graph Convolutional Networks (GCN) along with computational program graphs in order to learn a high dimensional representation of binary executable programs. We demonstrate the versatility of this approach by using our representations to solve two semantically different binary analysis tasks – functional algorithm classification and vulnerability discovery. We compare the proposed approach to our own strong baseline as well as published results, and demonstrate improvement over state-of-the-art methods for both tasks. We evaluated Bin2vec on 49191 binaries for the functional algorithm classification task, and on 30 different CWE-IDs including at least 100 CVE entries each for the vulnerability discovery task. We set a new state-of-the-art result by reducing the classification error by 40% compared to the source-code based inst2vec approach, while working on binary code. For almost every vulnerability class in our dataset, our prediction accuracy is over 80% (and over 90% in multiple classes).

Keywords: Binary program analysis, Computer security, Vulnerability discovery, Neural networks

Introduction

For many security problems, researchers are relying on binary code analysis, as they need to inspect binary executable program files without access to any source code. This is often needed when analyzing commercial code that is protected by intellectual property and its source code is not available, but can be also useful in other scenarios. Those include dealing with unsupported or legacy executables, where the information about the exact version of the source code is lost, or even the original source code itself may be lost. Additionally, it is frequently used for testing in order to improve the security of the system, like in black-box penetration testing when the goal is to check the binary for any weaknesses or vulnerabilities that can potentially be abused. And finally, it is an important part of investigating how hard recovering key parts of the algorithm is, e.g., for the sake of preventing intellectual property theft.

The translation process of going from source code to binary executable programs also called compilation, is a lossy process in the sense that only basic low-level instructions and data representations understood by the target CPU are preserved. Because of this, it is impossible in the general case to reconstruct the original source code from compiled binary code. The task is even more complicated with commercial binaries as they are often stripped. Stripping of the binary removes any debug information and its symbol tables, which contain semantics of variables in the program. When dealing with stripped binaries, even reconstructing function entry points can be challenging.

With a constantly growing number of computing devices in consumer, commercial, industrial and infrastructure applications, as well as with the growing complexity of software applications, the scope of binary code

*Correspondence: shushana@usc.edu
Information Sciences Institute, 4676 Admiralty Way, Marina Del Rey, CA, USA

analysis becomes increasingly large. Fast, automated analysis would allow preventing the spreading of bugs and vulnerabilities in all those complex software systems through shared and reused code.

Analyzing binary executable code is difficult because of two related challenges - the size of binary executable programs and the absence of high-level semantic structure in binary code. Indeed, when dealing with a compiled executable, a security engineer is often looking at a file containing up to megabytes of binary code. A precise analysis of such files with existing tools requires large amounts of computational power, and it is particularly difficult or even impossible to do manually. Instead, state-of-the-art tools often rely on a combination of formal models and heuristics to reason about binary programs. Replacing these heuristics with more advanced statistical learning and machine learning models has a high potential for improving performance while keeping the analysis fast.

In recent years we have seen a big surge in applications of machine learning (ML) to the field of security, where researchers routinely turn to ML algorithms for smarter automated solutions. For example, due to rapidly evolving modifications of malware, ML algorithms are frequently applied to malware detection problems. Similarly, ML algorithms allow detecting and reacting to network attacks faster.

Having ML algorithms operate on binary executable programs is a promising direction to bridge the large semantic gap between human abstractions and machine code representations, and to recover high-level semantics which was lost during compilation. Using ML requires obtaining a good, vectorized representation of the data. In the field of security, this problem is usually solved by hand-selecting useful features and feeding those into an ML algorithm for a prediction or a score. Approaches range from defining code complexity metrics and legacy metrics (Theisen et al. 2015), to using a sequence of system calls (Grieco et al. 2016) and many more. Besides being non-trivial and laborious, hand-selecting features raises other issues as well. First, for every task researchers come up with a new set of features. For example, what indicates memory safety violations is unlikely to also signal race conditions. Additionally, some features get outdated and will need to be replaced with future versions of the programming language, compiler, operating system or computer architecture.

The state-of-the-art in machine learning, however, no longer relies on hand-designed features. Instead, researchers use learned features, or what is called *distributed representations*. These are high-dimensional vectors, modeling some of the desired properties of the data. The famous word2vec model (Mikolov et al. 2013a; Mikolov et al. 2013b), for example, is representing words in a high-dimensional space, such that similar words are clustered together. This property of word2vec has made it a long-time go-to model for representing words in a number of natural language processing tasks. We can take another example from computer vision, where it was discovered that outputs of particular layers of VGG network (Simonyan and Zisserman 2015) are useful for a range of new tasks.

We see an important argument for trying to learn distributed representations - a good representation can be used for new tasks without significant modifications. Unfortunately, some types of data are more challenging to obtain such a representation for, than others. For instance, finding methods for representing longer sentences or paragraphs is still an ongoing effort in natural language processing (Zhang et al. 2017; Lin et al. 2017b). Representing graphs and incorporating structure and topology into distributed representations is not fully solved either. Binary executable programs are a "hard" case for representing as they have traits of both longer texts and structured, graph-like data, with important properties of binaries best represented as control or data flow graphs.

Distributed representations for compiled C/C++ binaries – the kind that engineers in the security field deal with the most – have not received much attention, and with this work, we hope to start filling that gap. In fact, current approaches leveraging deep learning models to reason about binary code focus on code clone detection, and therefore, their application to algorithm classification and vulnerability detection is limited to syntactically similar patterns. In contrast, our approach aims to generalize code semantics based on new insights by introducing a graph embedding model which encompasses notions of local control-flow and data-flow in a novel way. We propose a graph-based representation for binary programs, that when used with a Graph Convolutional Network (GCN) (Kipf and Welling 2017), captures semantic properties of the program.

Our main contributions are: (i) To the best of our knowledge we are the first to suggest a distributed representation learning model approach for binary executable programs that is demonstrated to work for different downstream tasks;(ii) To this end, we present a deep learning model for modelling binary executable programs' structure, computations, and learning their representations; (iii) To prove the concept that distributed representations for binary executable programs can be applied to downstream programs analysis tasks, we evaluate our approach on two distinct problems - functional algorithm classification (i.e., the task of recognizing functional aspects of algorithmic properties, as opposed to their syntactic aspects) and vulnerability discovery across multiple vulnerability classes, and show improvement over current state-of-the-art approaches on both.

Related work

Many tasks that rely on the analysis of binary executables are frequently approached by rule-based systems and manually defined heuristics (Aafer et al. 2013; Santos et al. 2009; Karbab et al. 2018; Yamaguchi et al. 2014; Rawat and Mounier 2012; Cha et al. 2012). Machine learning has a proven reputation for boosting performance compared to heuristics and there has been a lot of interest in applications of machine learning to security tasks. We briefly discuss previous work in binary program analysis that relies on machine learning. We structure the literature based on the types of features extracted and by the type of the embedding model applied.

Hand designed features

Designing and extracting features can be considered equivalent to manually crafting representations of binaries. We can classify such approaches based on which form of the compiled binary program was used to extract the features.

Code-based features The simplest approach to representing a binary is by extracting some numerical or textual features directly from the assembly code. This can be done by using n-grams of tokens, assembly instructions, lines of code, etc. N-grams are widely used in the literature for malware discovery and analysis (Li et al. 2019a; Lee et al. 2018; Kang et al. 2016), as well as vulnerability discovery (Pang et al. 2015; Murtaza et al. 2016). Additionally, there have been efforts focusing on extracting relevant API calls or using traces of system calls to detect malware (Wu et al. 2016; Kolosnjaji et al. 2016).

Graph-based features Many solutions rely on extracting some numerical features of Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs) and/or Data Flow Graphs (DFGs). We combine these under models with graph-based features. discovRE (Eschweiler et al. 2016), among other features, uses closeness of control flow graphs to compute similarity between functions. Genius (Feng et al. 2016) converts CFG into numeric feature vectors to perform cross-architecture bug search. Yet other works have used quantitative data flow graph metrics to discover malware(Wüchner et al. 2015).

Learned features

Besides manually crafting the representations it is also possible to employ neural models for that purpose. This allows expressing and capturing more complicated relations of characteristics of code. Here we can classify the approaches based on whether they use sequential neural networks or graph neural networks.

Sequence embeddings The body of work on the naturalness of software (Hindle et al. 2016; Ray et al. 2016; Allamanis et al. 2018) has inspired researchers to try applying NLP models for security applications in general, and binary analysis in particular. Researchers have suggested serializing ASTs into text and using them with LSTMs for vulnerability discovery (Lin et al. 2017a). Some of previous vulnerability discovery efforts also use RNNs on lines of source code (Li et al. 2018). More recently, INNEREYE proposed to use LSTMs in a Siamese architecture for binary code similarity detection (Zuo et al. 2019). The closest to our work is that of (Ding et al. 2019), which is starting by constructing a graph that is enriched with selective callee expansion. The authors then sample random walks from this graph to generate sequences of instructions and train a paragraph-to-vector model on these sequences. This approach is similar in spirit to earlier graph embedding approaches, such as DeepWalk (Perozzi et al. 2014), that were sampling random walks of nodes and using word embedding models on sequences of adjacent nodes for representation learning. However, today these approaches for graph embedding are no longer popular, as graph neural networks based on message passing and neighbourhood aggregation have been shown to perform much better.

Graph embeddings Graph embedding neural models are a popular choice for tackling binary code-related tasks because the construction of Control Flow or Data Flow Graphs is frequently an intuitive and well-understood first step in binary code analysis. For instance, graph embedding models have successfully been used on top of Control Flow Graphs for tackling the task of code clone detection in source code and binary programs (White et al. 2016; Xu et al. 2017; Li et al. 2019b; Zhou et al. 2019). From these, Gemini (Xu et al. 2017) uses a Siamese architecture on top of a graph embedding model for binary code clone detection task. The graphs they use - attributed control flow graphs, or ACFGs, - are CFG graphs that are enriched with a few manually defined features. In our work, instead of enhancing the basic blocks in CFG with a few attributes, we suggest enriching them by expanding the computations in each basic block into a computational tree, and rely on the fact that the graph embedding model will be able to capture attributes like the number of instructions if necessary. Graph Matching Networks (GMNs), (Li et al. 2019b) on the other hand, are based on the idea that instead of computing an embedding and then either using a distance function, or a Siamese network for the comparison, it might be beneficial to directly compare two graphs. So, as opposed to Gemini, where representations for known vulnerable or benign programs were pre-computed, GMN needs to compute similarity for every pair of programs individually starting from scratch.

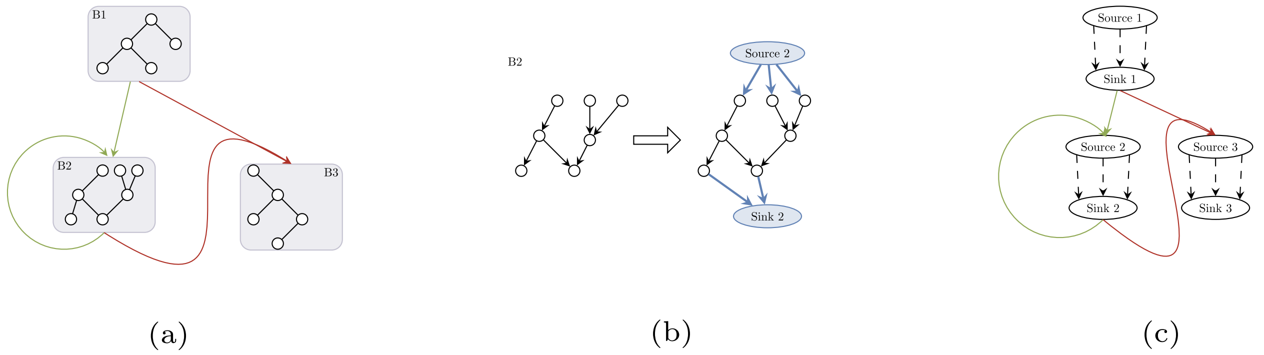


Fig. 1 This figure illustrates how the program graph is constructed from the CFG on a toy example with three basic blocks. Figure (**a**) demonstrates, that for each basic block independently we construct a computational tree. On fig (**b**) we can see, that each tree is completed with two nodes - a source and a sink, this way turning it into a directed acyclic graph (DAG). On fig (**c**) we demonstrate how all DAGs are connected following the same topology as CFG had initially

They demonstrate that this approach has better performance compared to Siamese architectures, but it is clearly slower, and more importantly for us - it does not produce program embeddings.

Other research using graph structure of binary programs include using Conditional Random Fields on an enhanced Control Flow Graph for attempting to recover the debug information of the binary program (He et al. 2018).

Model

We start by converting the binary executable to a program graph that is designed to allow mathematically capturing the semantics of the computations in the program. Next, we use a graph convolutional neural network to learn a distributed representation of the graph. Below we describe the process of constructing the program graph, followed by a brief introduction to how graph convolutional neural networks work. We also describe the baseline model that we use for evaluation and comparison, alongside previous existing approaches.

Program graphs

We start by disassembling the binary program and constructing a control flow graph (CFG). We use static interprocedural CFGs, which we construct using the angr library (Shoshitaishvili et al. 2016).

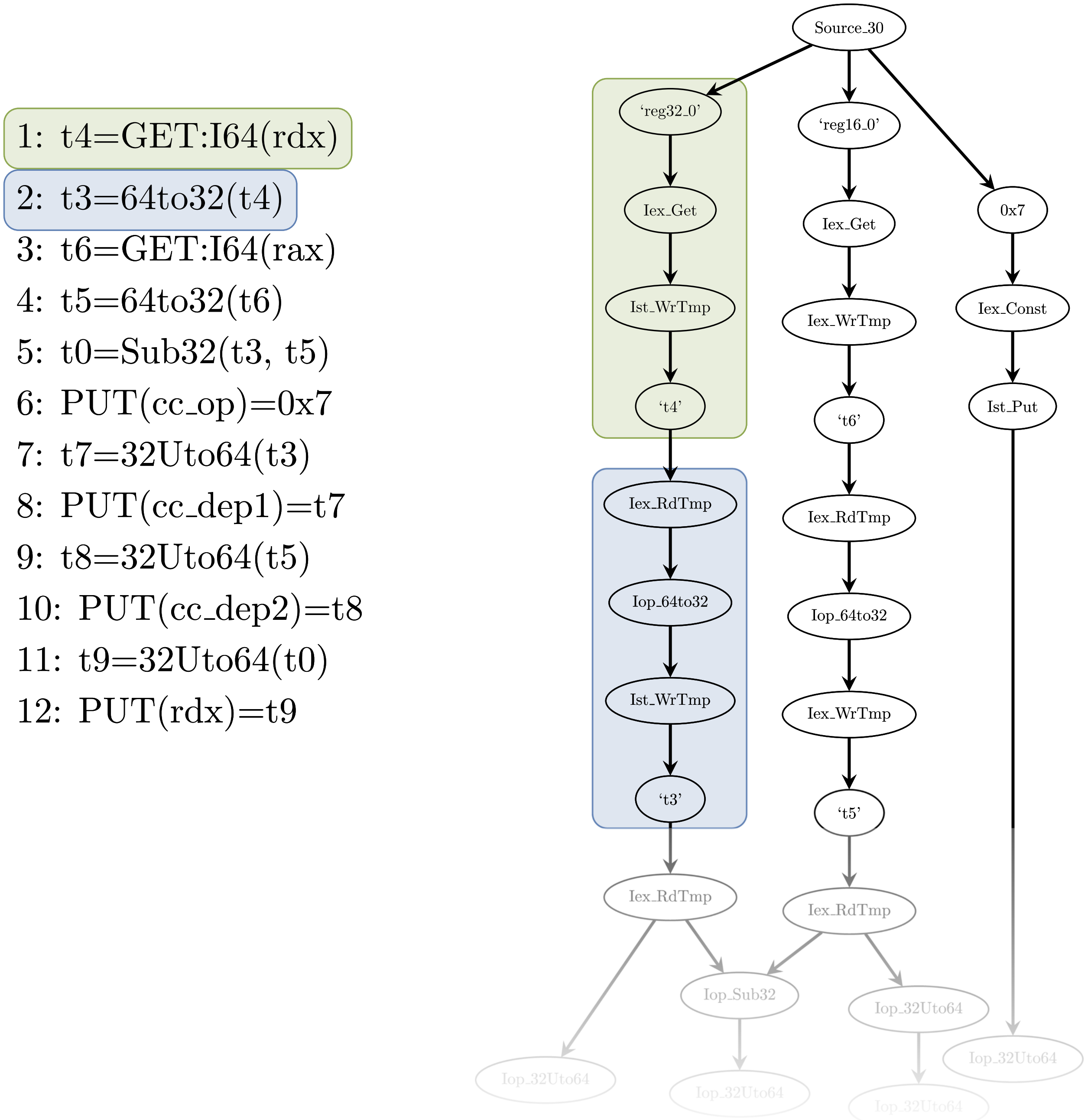
The fact that each basic block in CFG is executed linearly allows us to unfold the instructions within each basic block and represent them as a directed, computational tree, similar to an Abstract Syntax Tree (AST). The result of this process is schematically depicted in Fig. 1a.

Within each basic block, computations do not necessarily all depend on each other. There may be chunks of code that can be reordered inside the basic block without affecting the final result. In this case the approach described so far yields a forest of computations. To connect the trees in the forest we add *Source* and *Sink* nodes at the beginning and at the end of each basic block as a parent, or correspondingly a child, for all the trees generated from that basic block, which is demonstrated in Fig. 1b. The resulting graphs are then connected following the same topology that basic blocks originally had in the CFG, as shown in Fig. 1c.

We construct the above-mentioned computational trees from VEX intermediate representation (IR) of the binary. Figures 2 and 3 provide demonstration of the process.

Every node of the resulting tree is thus labelled with a constant, a register, a temporary or an operation. The edges of the tree are directed from the argument to the instruction. Within each basic block we reuse nodes that correspond to addresses, temporaries, constants and registers to tie together related computations. VEX IR provides Static Single Assignment form (SSA). This means that each assembly instruction in a basic block is lifted and "spilled" into multiple IR statements operating on temporary variables that are each used only once (the goal being to make all side effects of an instruction explicit). However, VEX does not track instances of different *definitions* and *uses* of the same register across instructions within the basic block, which we implemented to ensure we do not introduce fake data-dependence edges. In our implementation, if an instruction overrides or redefines the content of a register, its subscript is incremented. For example, for the `eax` register, we start from `eax_0` and increment it to `eax_1`. This is necessary so that we do not reuse the same node for `eax_0` and `eax_1`.

As a last step, we remove redundant edges and nodes, particularly, the `Iex_Const` node that follows



```
1:  t4=GET:I64(rdx)
2:  t3=64to32(t4)
3:  t6=GET:I64(rax)
4:  t5=64to32(t6)
5:  t0=Sub32(t3, t5)
6:  PUT(cc_op)=0x7
7:  t7=32Uto64(t3)
8:  PUT(cc_dep1)=t7
9:  t8=32Uto64(t5)
10:  PUT(cc_dep2)=t8
11:  t9=32Uto64(t0)
12:  PUT(rdx)=t9
```

Fig. 2 An example of the program graph. Parts of the graph are highlighted in the same color, as instructions on lines 1 and 2, to demonstrate where those instructions were mapped to in the graph

every constant, and chains of `Iex_WrtTmp` → 't%' → `Iex_RdTmp`[1]. This is demonstrated in Fig. 3.

After the graph construction is complete, we remove SSA indices for temporary variables and registers to reduce the number of distinct labels.

From the labels of the nodes we construct a "feature matrix" of the graph, which is a matrix of size $n \times d$, where n is the number of nodes in the graph, and d is the number of all distinct labels seen in the entire dataset. Thus, every node has one row in the feature matrix associated with it. We choose a random fixed ordering of all labels, and then for a given node, to convert its label into its feature row we assign all positions of the row to zero with the

[1] In VEX `Iex_Const` represents a constant value, `Iex_WrtTmp` a write operation (to a temporary variable), and `Iex_RdTmp` a read operation (from a temporary variable)

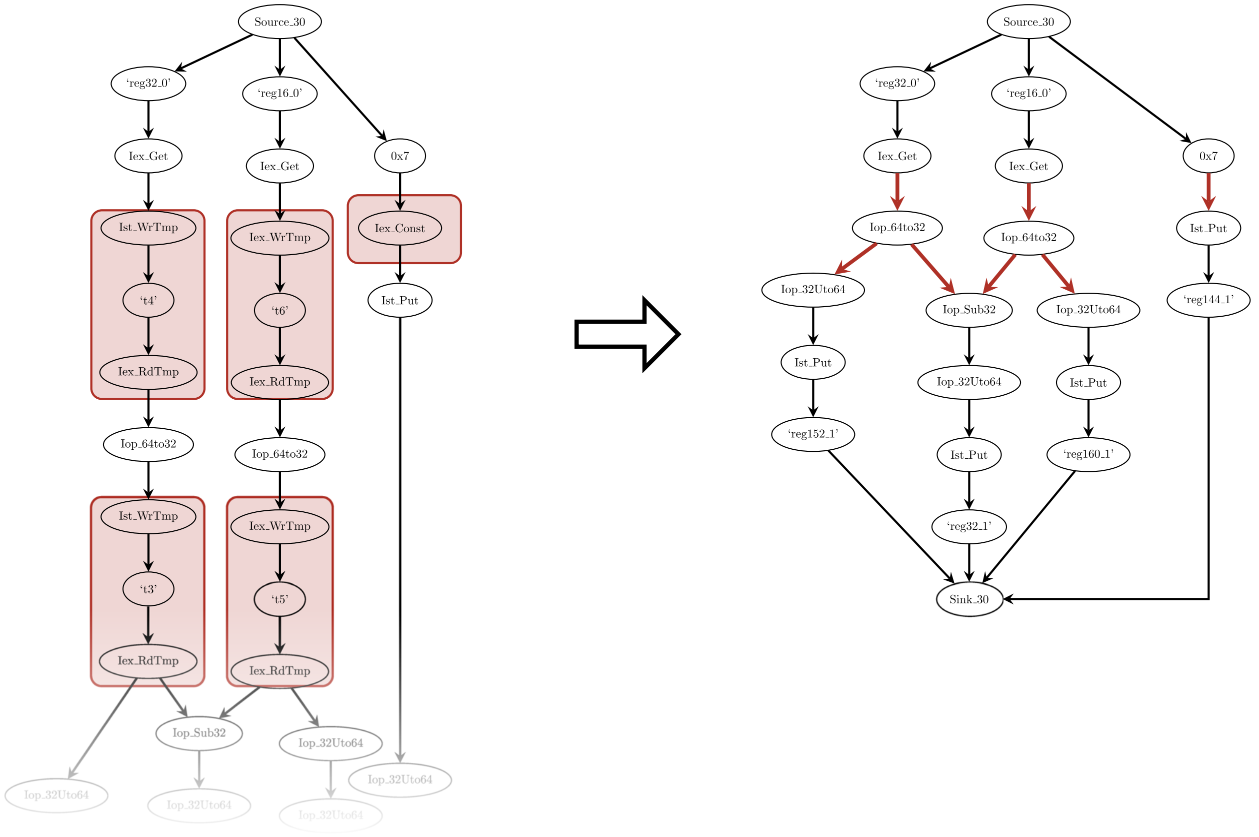


Fig. 3 Here we show how redundant instructions were removed to contract the graph. On the left the graph is shown before the contraction, and on the right it is demonstrated after the contraction

exception of the position that corresponds to the label of the node in our fixed ordering of labels. This representation is known as a one-hot representation. We will further refer to the feature matrix as X. Note that we use words "feature" or "features", "embeddings" and "representation" interchangeably.

Graph convolutional networks

The model we used for learning representations is a Graph Convolutional Neural Network (GCN) (Kipf and Welling 2017). Graph neural embeddings is a fast developing field, and some alternative graph representation learning models include GraphSAGE (Hamilton et al. 2017) or Gated Graph Neural Networks (Li et al. 2016) , as well as a number of others. In the literature GCNs consistently perform on par or better than more recent variants of graph neural networks (Monti et al. 2017; Liu et al. 2019; Velickovic et al. 2017; Chen et al. 2018), while being simpler and oftentimes, faster. We chose GCN because it provides a good trade-off between simplicity, performance, and speed. The latter is important due to the low-level nature of the binary code; it is reasonable to expect the program graphs to grow quite large, which forces us to favour a model with weight updates that can be efficiently computed in batches.

GCN consists of a few stacked graph convolutional layers. A graph convolutional layer is applied simultaneously to all nodes in the graph. For each node, it averages the features of that node with features of its neighbours. Features of different nodes are scaled differently in the process of averaging and these weights are learned, i.e. they are the parameters of the graph convolutional layer. After the averaging, each node is assigned the resulting vector as its new feature vector and we proceed to either apply a different graph convolutional layer, or compute the loss and perform backpropagation to update the parameters.

Formally, this process of computing new feature vectors, known as forward pass or propagation, for $(l+1)$-st graph convolutional layer can be described as follows:

$$H^{(l+1)} = \text{ReLU}\left(D^{\frac{1}{2}}\tilde{A}D^{\frac{1}{2}}H^{l}W^{l}\right) \qquad (1)$$

where $\tilde{A}$ is the adjacency matrix of the graph with added self-loops, D is its diagonal out-degree matrix, $\text{ReLU}(x) = max(0, x)$ is the non-linearity or activation function, $H^{(l)}$

is the result of propagation through previous layer, $H^{(0)}$ being X, and W^l is a layer-specific trainable weight matrix.

Since one graph convolutional layer averages representations of the immediate neighborhood of the node, after performing k graph convolutions we incorporate the information from k-th neighborhood of the node.

From our description, it follows that after the forward pass, the graph convolutional network outputs features for each node in the graph. We will refer to this new feature matrix as Z. Note that Z still has n rows - one row per node, but it can have a different number of columns.

To get the representation of the entire graph, we can aggregate the features of all nodes in the graph. Here it is possible to use any aggregation function - summation, averaging, or even a neural attention mechanism, but in our experiments we went for a simple sum aggregate. A schematic illustration of this entire process is available in Fig. 4.

The aggregated representation is used with a two-layer perceptron, and passed through a softmax which is defined like softmax$(x_i) = \frac{exp(x_i)}{\sum_i exp(x_i)}$, for the final classification.

We frame our tasks as classification and use cross-entropy error as the objective function for the optimization. We cover our procedure for selecting hyperparameters for GCN model in more detail in "Task 1. experimental setup" section.

Baselines

We wanted to compare our proposed representation to another task-independent representation, in particular, to one that used code-based features or embeddings. We experimented with Long Short Term Memory (LSTM) neural networks and Support Vector Machine (SVM) classifiers for that purpose. We interpreted instructions as words, and a sequence of instructions as a sentence, following a number of similar approaches in the field, e.g. (Zuo et al. 2019). We experimented using both SVM and LSTM with the assembly instructions directly, as well as with the code lifted to VEX IR. From our experiments, an SVM classifier with a Gaussian kernel and bag-of-words representation of VEX IR gave us the best performance, so that is the setup we chose as a baseline. Each line of IR is tokenized to be a single "word". Vocabulary for the bag-of-words was obtained from the training part of the dataset. We used frequency thresholding to remove infrequent entries and reduce data sparsity. Those frequencies were empirically found on the validation part of the dataset.

Task description

We evaluate the performance of our proposed representations on two independent tasks. In the first, we test the proposed representations for functional algorithm classification in binary executable programs through classifying coding challenges. In our second task, we want to demonstrate the performance of learned representations on a common security problem – discovery of vulnerable compiled C/C++ files. The two tasks are semantically different and we demonstrate in the later sections that both can be successfully tackled with representations constructed and learned in the same way.

Task 1: Functional algorithm classification

Algorithm classification is crucial for semantic analysis of code. We qualify it as "functional" by opposition to "syntactic", i.e., we aim to capture the semantics of functional properties of algorithms. It can be used for creating assisting tools for security researchers to understand and analyze binary programs, or discover inefficient or buggy algorithms, etc.

In this task, we are looking at real-world programs submitted by students to solve programming competition problems. We chose such a dataset because the programs in it, being written by different students, naturally encompass more implementation variability than it would be possible to get by using, for instance, standard library implementations. Our goal is to classify solutions by the problem prompts that the solution was written for.

We present a typical example of programming competition problem prompt in Table 1. Provided example is for illustrative purposes only, as it is taken from ACM Timus (http://acm.timus.ru) and is not part of our dataset[2].

From our definition and the dataset, it follows that we define the equivalence of two programs as them solving the exact same problem. Hence, in this task, we test the *ability of the model to capture the higher-level semantic* similarity, and to take into account program behaviour, functionality and complexity, while *ignoring syntactic differences* wherever possible.

Task 2: vulnerability discovery

Software contains bugs, which in the worst case can lead to weaknesses that leave vulnerable systems open to attacks. Such security bugs, or vulnerabilities, are classified in a formal list of software weaknesses - Common Weakness Enumeration (CWE). Vulnerability discovery is the process of finding parts of vulnerable code that may allow attackers to perform unauthorized actions. It is an important problem for computer security. The typical target of vulnerability discovery is programming mistakes accidentally introduced in *benign commodity programs* by their authors. Our work excludes software specifically crafted to behave in a malicious way, and focuses on benign programs. Due to the large variability among

[2] The dataset we used was collected as part of previous work for which the problem prompts are not released with the data

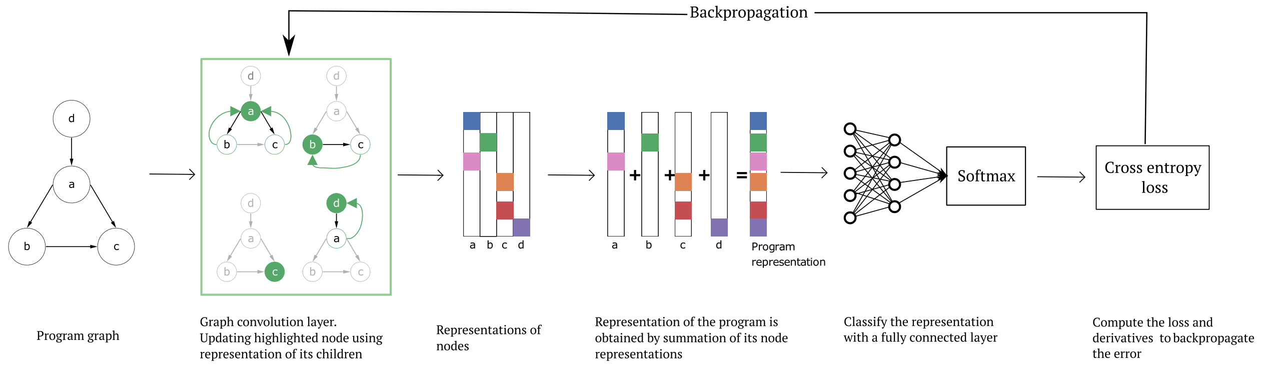

Backpropagation
d
a
b
c
a b c d
+
+
+
=
Program representation
Softmax
Cross entropy loss
Program graph
Graph convolution layer. Updating highlighted node using representation of its children
Representations of nodes
Representation of the program is obtained by summation of its node representations
Classify the representation with a fully connected layer
Compute the loss and derivatives to backpropagate the error

Fig. 4 Schematic depiction of obtaining a program representation with a single-layer GCN model

vulnerabilities, increasingly large sizes of software and increasing costs of testing it, the problem of vulnerability discovery is not solved.

Most vulnerability discovery techniques rely on dynamic analysis for program exploration, the most common one being fuzzing (Zalewski 2017). Such models offer a high level of precision, at the cost of shallow program coverage: only a subset of execution traces for a given program (along with a set of input test cases) can be observed in finite time, leaving large parts of the program unexplored. On the other hand, static analysis provides better program coverage at the cost of lower precision. In addition to these challenges come a range of fundamental problems in program analysis related to undecidability (e.g., the halting problem, i.e., “Does the program terminate on all inputs?”) and implementation. These issues emerge because vulnerabilities may span very small or very large chunks of code and involve a range of different programmatic constructs. This raises the question - at what level of granularity in the program should we inspect them for vulnerabilities or report to security researchers. In this work, we are concerned with the question of learning representations for the entire binary program that will help to discover vulnerabilities statically, while leaving the questions of handling large volumes of source code and working on variable levels of granularity for future work. Our work builds on standard binary-level techniques for control-flow recovery (i.e., the reconstruction of a CFG), which is a well-studied problem where state-of-the-arts models perform well with high accuracy and scalability (Andriesse et al. 2016).

Table 1 An example prompt for programming competition problems and their corresponding problem numbers and names. The example is taken from ACM Timus http://acm.timus.ru/

Prompt	Problem #
You have a number of stones with known weights $w_1, \ldots w_n$. Write a program that will rearrange the stones into two piles such that weight difference between the piles is minimal	1005. Stone Pile

Datasets and experimental setup

Our first dataset, introduced by Mou et al. (2016), consists of 104 online judge competition problems and 500 C or C++ solutions for each problem submitted by students. We only kept the files that could be successfully compiled on a Debian operating system, using gcc8.3, without any optimization flags. This left us with 49191 binary executable files, each belonging to one of 104 potential classes. Each class in this dataset corresponds to a different problem prompt and our goal is to classify the solutions according to their corresponding problem prompts.

The second dataset we used is the Juliet C/C++ test suite (Boland and Black 2012). This is a synthetically generated dataset, that was created to facilitate research of vulnerability scanners and enable benchmarking. The files in the dataset are grouped by their vulnerability type – CWE-ID. Each file consists of a minimal example to recreate the vulnerability and/or its fixed version. Juliet test suite has *OMITGOOD* and *OMITBAD* macros, surrounding vulnerable and non-vulnerable functions correspondingly. We compiled the dataset twice - once with each macro, to generate binary executable files that contain vulnerabilities and those that do not. The dataset contains 90 different CWE-IDs. However, some of them consist of Windows-only examples, that we omitted. Note that even though our approach is not platform-specific, in this work we limit our experimentation to Linux only.

Most CWE-IDs had too few examples to train a classifier and/or to report any meaningful statistics on[3]. Thus, we also omitted any CWE-ID that had less than 100 files in its testing set after 70:15:15 for training:validation:test

[3] In the future, we consider combining some CWEs into their umbrella categories, for example following the classification by Research Concepts: https://cwe.mitre.org/data/definitions/1000.html

split, because for those cases the reported evaluation metric would be too noisy. As a result, we experimented on vulnerabilities belonging to one of 30 different CWE-IDs, presented in Table 2. We trained a separate classifier for every individual CWE-ID, which was required because files associated with each CWE-ID may or may not contain other vulnerability types.

We trained the neural network model with early stopping, where the number of training epochs was found on the validation set.

Task 1. experimental setup

For experiments in the functional algorithm classification task, we randomly split all the binaries in the first dataset into train:test:validation sets with ratios 70:15:15. We use the training set for training and extracting some additional helper structures, such as vocabulary for the bag of words models and counting frequencies for thresholding in neural network models. We use the validation set for model selection and finding the best threshold values. After finding the best model, we evaluate its performance on the testing set. The experiments are cross-validated and averaged over 5 random runs.

For SVMs, in the model selection phase, we perform a grid-search over the penalty parameter C and pick a value for the vocabulary threshold to remove any entry that does not have a substantial presence in the training set to be useful for learning. After the trimming our vocabulary contains about 10-11K entries (the exact number changes from one random run to another).

For GCN-based representation, we follow similar logic and use the training set to find and remove infrequent node labels. Here too the exact threshold is decided via experimentation on the validation set. On average, we keep about 7-8K different node labels. Very infrequent terms are replaced with a placeholder *UNK*, or *CONST* if it is a hexadecimal.

We pick hyperparameters of the GCN model by their performance on the validation set. Figure 5 demonstrates the influence of the *depth* (number of graph convolution layers) and *width* (size of each graph convolution layer) on the performance of the model for Task 1. Figure 5**(A)** shows the peformance of models with depths from 1 to 8 layers, while the dimensionality of every layer is set to 64. As it can be seen, increasing the depth of the model up until 4 layers improves performance, however additional layers after that do not always improve performance. Figure 5**(C)** compares performances of four models where each model has the same number of layers (3), but different sizes of layers - 32, 64, 128 or 256. From here we see, that increasing the size of the layers from 64 to 128 provides a moderate improvement, but increasing the size further does not affect the performance. Figures 5**(B)** and **(D)** show the duration of training in seconds of each of the discussed above models on 100 examples. Based on these general findings we perform some additional experimentation and deploy a GCN with 3 layers, that has 128 dimensions in its first two layers, and 64 dimensions in its last layer.

Task 2. experimental setup

In the vulnerability discovery experiments, we train a separate classifier for each of 30 different CWE-IDs. Note, that for each CWE-ID classifier in its training and testing we only include the binaries that are specifically marked as good or bad with regard to that CWE-ID. For every CWE-ID, we split its corresponding binaries into train:validation:test with ratios 70:15:15, and report results averaged over 5 random runs. We use training sets for training the models and validation sets for grid search of the penalty parameter C in SVMs. We report the performance of the best model measured on testing sets. Here we reuse some statistics obtained on the first dataset, in particular, we reuse frequency thresholds and bag-of-words vocabularies. We need to train a separate classifier for each CWE-ID, 30 SVM classifiers and 30 NN classifiers in total, which would lead to a huge search space at the phase of the model selection.

We are not aware of related work on vulnerability discovery that performs their evaluation on Juliet Test Suite. Thus, to give the readers a better understanding of how our proposed model would fare compared to other existing approaches, we performed an additional experiment using Asm2Vec model (Ding et al. 2019) on the Juliet Test Suite. Asm2Vec is a clone search engine that relies on vector representations of assembly functions. In the original paper the authors suggested its usefulness as a vulnerability detection tool which allows finding duplicates of known vulnerable functions. We tried replicating that scenario as faithfully as possible, by training Asm2Vec[4] on Juliet Test Suite, and comparing resulting representations to differentiate between vulnerable and non-vulnerable instances. Since Asm2Vec poses the vulnerability detection as a retrieval problem, we follow their example in the paper and report Precision@15 metric. For each vulnerable function, we find 15 most similar functions to it according to cosine similarity and compute the percentage of vulnerable functions among them. It is worth noting that we are looking for similar functions among all vulnerable and non-vulnerable functions per CWE-ID.

We set most of the hyperparameters of Asm2Vec following the original paper, but finetune for dimensionality of the representation and learning rate. To find best values for those we use grid search in intervals {50,100,150,200} and {0.05, 0.025, 0.01} correspondingly. The final results

[4] We used implementation available here: https://github.com/Lancern/asm2vec

Table 2 In this table we provide total counts of binary executables for each of the CWE-IDs we studied in the Juliet Test Suite

CWE-ID	# examples	CWE-ID	# examples	CWE-ID	# examples
CWE121	9486	CWE197	2664	CWE476	888
CWE122	11946	CWE23	2960	CWE563	1116
CWE124	3612	CWE36	2960	CWE590	6954
CWE126	2639	CWE369	2736	CWE606	760
CWE127	3612	CWE400	2280	CWE617	918
CWE134	3800	CWE401	4176	CWE680	1776
CWE190	12093	CWE415	2588	CWE690	2368
CWE191	9048	CWE416	888	CWE758	1046
CWE194	3552	CWE427	740	CWE761	888
CWE195	3552	CWE457	2104	CWE762	6429

that we report for Asm2Vec are computed on the testing set, and are the average of 5 random runs.

Evaluation and results

For evaluating performance in our experiments we used accuracy following previous work that we proceed to compare our results to.

Task 1

Table 3 contains quantitative evaluation of our representation for Task 1. Our proposed representation outperforms our own SVM baseline, TBCNN model (Mou et al. 2016), and current state-of-the-art for this task - *inst2vec*(Ben-Nun et al. 2018). We manage to reduce the error by more than 40%, thus setting a new state-of-the-art result. It

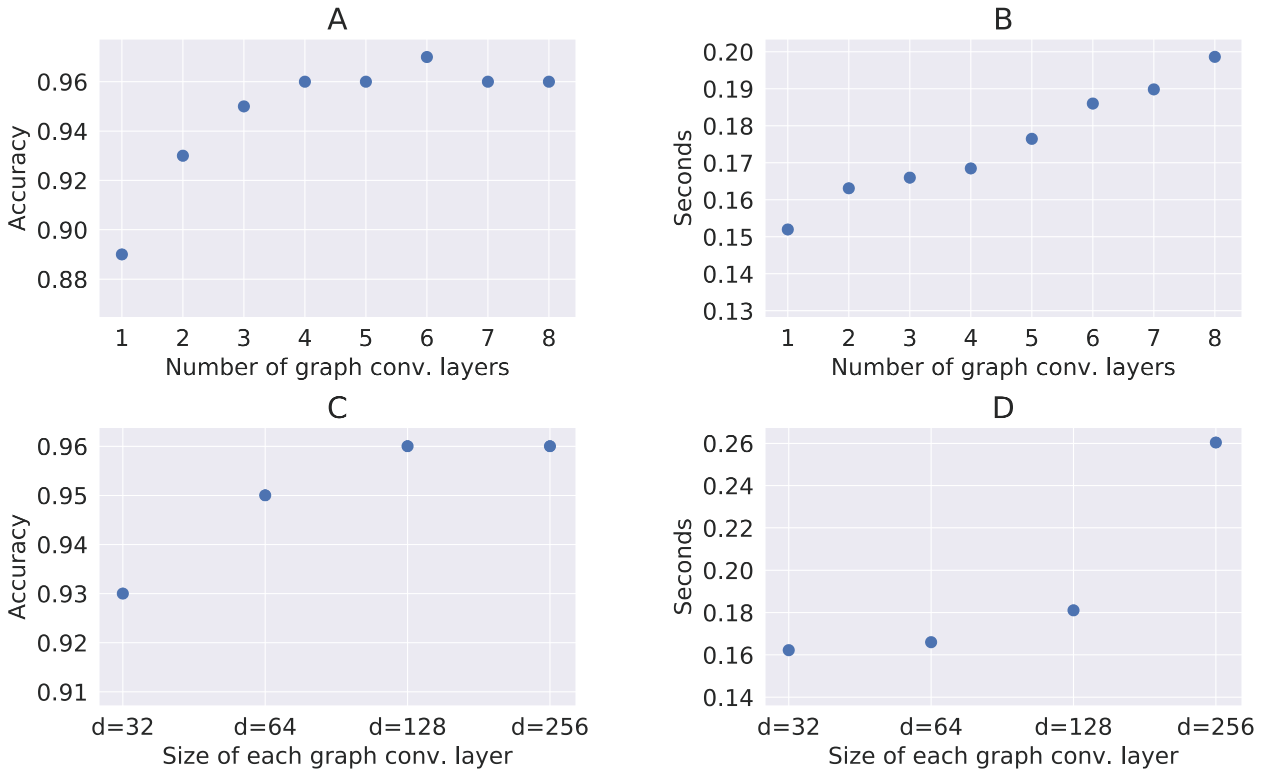


Fig. 5 Comparison of GCN models with different numbers of layers (A), and different sizes of each layer (C). (**A**) demonstrates the accuracy of GCN models with different numbers of layers - from 1 to 8 - on the validation set of Task 1; the size of each layer is 64. (**B**) shows the time in seconds that models from (A) take to train on 100 samples. (**C**) demonstrates the accuracy of GCN models with 3 layers, but different sizes of layers - from 32 to 256. (**D**) shows the time in seconds that models from (C) take to train on 100 samples

Table 3 Accuracy obtained for the first task on the online judge problem classification

Model	Accuracy
SVM on VEX IR	0.93
TBCNN (Mou et al. 2016)	0.94
inst2vec (Ben-Nun et al. 2018)	0.9483
Ours	0.97

should be additionally mentioned that both TBCNN and *inst2vec* start from the C source code of the programs to make predictions, whereas our baseline SVM and our proposed model are only using compiled executable versions.

Highlighting a few important differences between our approach and *inst2vec* helps better understanding some of the contributions of our approach. To construct the contextual flow graphs, the authors of *inst2vec* compile the source code to LLVM IR, which contains richer semantic information than VEX IR that we use in this work. Because it is more high-level, LLVM IR is a difficult target for lifting from binary executable files[5].

Another key difference is that instead of learning the representations of individual tokens and then combining the tokens into a program using a sequential model, we learn the representations of all the tokens in the program jointly, thus learning the representation of the entire program. The *inst2vec*, on the other side, ignores the structural properties of the program at that step. Our results show that we can achieve better performance, despite *inst2vec* starting from a semantically richer LLVM IR. We believe this indicates the importance of using the structural information at all stages of learning for obtaining good program embeddings.

Task 2

Figure 6 contains the evaluation of our representation for Task 2. Here, the classifier based on our proposed representation outperforms our SVM baseline in all cases except 2 – CWE-ID590, Free of Memory not on the Heap, and CWE-ID761, Free of Pointer not at Start of Buffer. In both cases we are seeing less than 5% difference in accuracy. On the other hand, our proposed representation demonstrates a significant gain in terms of performance. In the extreme case of CWE-617, Reachable Assertion, it outperforms the baseline by about 25%, in many other cases the gain is from 10% to 20% of prediction accuracy.

Table 4 reports the results we obtained from running Asm2Vec on Juliet Test Suite. It is important to keep in mind that these numbers are not directly comparable to our results, as they correspond to two different metrics. Rather, this experiment demonstrates the complexity of the dataset and the capacity of Asm2Vec to capture vulnerabilities on it. While Bin2Vec achieves more than 80% accuracy for all CWE-ID, Asm2Vec has Precision@15 equal to 0.5 or 0.6 in many cases, which means only about half of the retrieved similar functions were in fact vulnerable. Asm2Vec has highest Precision@15 of 0.77 for CWE-ID 416, Use After Free, which corresponds to about 1 in 4 retrieved functions being incorrectly labelled as vulnerable. For comparison, for the same vulnerability type Bin2Vec achieves near perfect performance.

Additionally, we can indirectly compare our results for the second task with those presented in two surveys that use Juliet Test Suite as a benchmark for evaluating commercial static analysis vulnerability discovery tools (Velicheti et al. 2014; Goseva-Popstojanova and Perhinschi 2015). It must be noted, that the commercial tools in those experiments probably did not use most of the programs for each CWE-ID as a training set. Additionally, the tools considered in those surveys are making their predictions based on source code and not binaries. Nevertheless, the comparison of the reported accuracies in those surveys with ours tells us that our proposed representation performs better for vulnerability discovery than static analysis commercial tools. For example, on CWE-IDs from 121 to 126, which are all memory buffer errors, (Velicheti et al. 2014) report less than 60% accuracy, whereas our model scores higher than 80% for each of those CWE-IDs. For tools studied in Goseva-Popstojanova and Perhinschi (2015), our model consistently outperforms three out of four static analysis tools, and for the last one it outperforms it by a considerable margin in all cases but two. Those two are CWE-ID122, Heap-based Buffer Overflow, where the commercial tool scores a few percents higher, and CWE-ID590, Free of Memory not on the Heap.

These results suggest that our representation has good prospects to be used in vulnerability discovery tools. For almost every vulnerability type our prediction accuracy performance is better than 80% and for many it is higher than 90%.

Discussion and future work

Software in production is usually complex and large, capable of performing many different functions in different use cases. On the contrary, programs in our evaluation datasets are single-purpose, solving a single task with a relatively small number of steps. Additionally, the entirety of each program in Juliet test suite is relevant to vulnerability discovery tasks, unlike real software where most of the code is not vulnerable and only a small part of it may have an issue. This can potentially be solved by introducing representations that can be computed on different levels of coarseness. This is a non-trivial task, but our findings hint that once completed we may be able to

[5] More discussion on this topic is provided at Angr's FAQ page: https://docs.angr.io/introductory-errata/faq

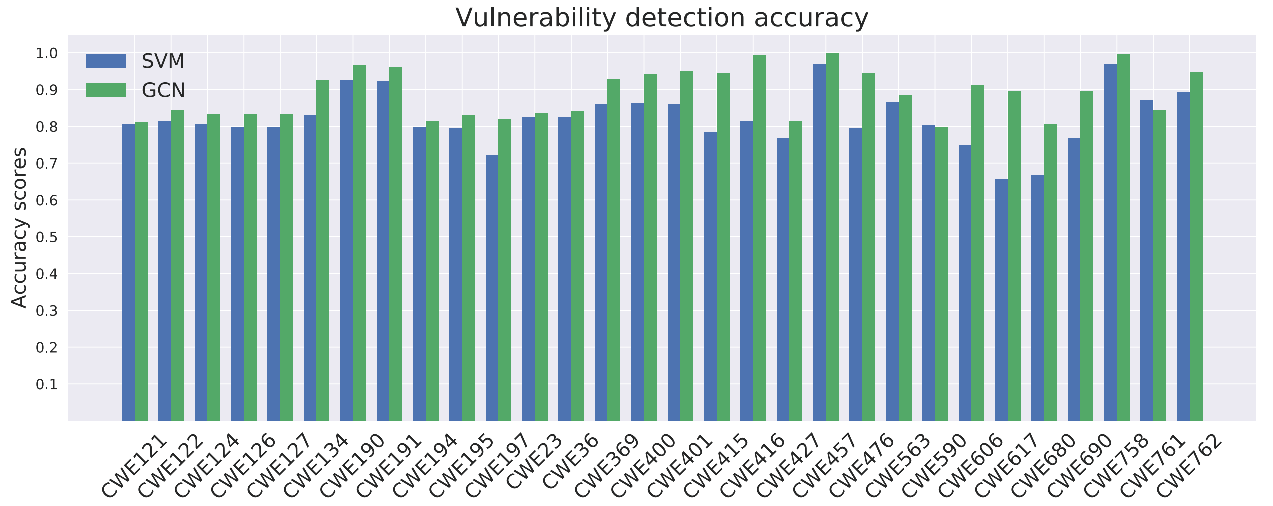


Fig. 6 Experimental results for vulnerability discovery on the Juliet test suite

achieve far better results for different problems on production software than is currently possible. Additionally, we need to get a better understanding of what properties are captured with such a representation and how is best to use those or how to add other desirable properties. Another challenge left for future work is extending this approach to cross-architecture and cross-compiler binaries.

There are several avenues for extending our work. First, it will be interesting to see whether using recent extensions of GCNs, such as the MixHop model (Abu-El-Haija et al. 2019) that propagates information through higher-order node neighbourhoods, will result in better performance. Additionally, to test the utility of Bin2Vec in real-world problems, we would like to apply it to analyze more complex and larger-scale vulnerability datasets.

Conclusion

We introduced Bin2Vec, a new model for learning distributed representations of binary executable programs. Our learned representation has strong potential to be used in the context of a wide variety of binary analysis tasks. We demonstrate this by putting our learned representations to use for classification in two semantically different tasks - algorithm classification and vulnerability discovery. We show that for both tasks our proposed representation achieves better qualitative and quantitative performance in comparison to state-of-the-art approaches, including inst2vec and common machine learning baselines.

Table 4 Asm2vec Precision@15 on Juliet Test Suite

CWE-ID	P@15	CWE-ID	P@15	CWE-ID	P@15
CWE121	0.72	CWE197	0.71	CWE476	0.73
CWE122	0.61	CWE23	0.68	CWE563	0.63
CWE124	0.63	CWE36	0.68	CWE590	0.64
CWE126	0.69	CWE369	0.54	CWE606	0.56
CWE127	0.66	CWE400	0.56	CWE617	0.64
CWE134	0.66	CWE401	0.50	CWE680	0.75
CWE190	0.53	CWE415	0.66	CWE690	0.55
CWE191	0.59	CWE416	0.77	CWE758	0.72
CWE194	0.71	CWE427	0.47	CWE761	0.55
CWE195	0.72	CWE457	0.69	CWE762	0.58

Acknowledgements
Not applicable.

Authors' contributions
C.H. and E.K. had the initial idea. Sh.Ar., C.H., E.K. and A.G. designed the study. Sh.Ar., C.H. and E.K. came up with evaluation scenarios. Sh.Ar. and C.H. implemented the solution. Sh.Ar. and A.G. designed machine learning experiments. Sh.Ar. performed original experiments. Sh.Ar. and C.H. analyzed the results and drafted the paper. A.G. and E.K. revised the draft. S.A performed additional experimentation during revision. The author(s) read and approved the final manuscript.

Funding
The authors are grateful for the funding to Information Sciences Institute of University of Southern California.

Availability of data and materials
All of the data used in this study is publicly available.

Declarations

Competing interests
The authors declare that they have no competing interests.

Received: 5 May 2020 Accepted: 11 April 2021
Published online: 01 July 2021

References

Aafer Y, Du W, Yin H (2013) Droidapiminer: Mining api-level features for robust malware detection in android. In: Zia TA, Zomaya AY, Varadharajan V, Mao ZM (eds). Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers, Springer, vol 127. pp 86–103. https://doi.org/10.1007/978-3-319-04283-1_6

Abu-El-Haija S, Perozzi B, Kapoor A, Alipourfard N, Lerman K, Harutyunyan H, Steeg GV, Galstyan A (2019) MixHop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. PMLR Long Beach Calif USA Proc Mach Learn Res 97:21–29

Allamanis M, Barr ET, Devanbu PT, Sutton CA (2018) A survey of machine learning for big code and naturalness. ACM Comput Surv 51(4):81:1–81:37. https://doi.org/10.1145/3212695

Andriesse D, Chen X, van der Veen V, Slowinska A, Bos H (2016) An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: USENIX. In: USENIX Association, Austin. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse

Ben-Nun T, Jakobovits AS, Hoefler T (2018) Neural code comprehension: A learnable representation of code semantics. In: Bengio S, Wallach HM, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R (eds). Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December, 2018, Montréal, Canada. pp 3589–3601. http://papers.nips.cc/paper/7617-neural-code-comprehension-a-learnable-representation-of-code-semantics

(2017) 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings, OpenReview.net(Bengio Y, LeCun Y, eds.) https://openreview.net/group?id=ICLR.cc/2017/conference

Boland T, Black PE (2012) Juliet 1.1 C/C++ and java test suite. IEEE Comput 45(10):88–90. https://doi.org/10.1109/MC.2012.345

Cha SK, Avgerinos T, Rebert A, Brumley D (2012) Unleashing mayhem on binary code. In: IEEE Symposium on Security and Privacy, SP 2012, 21-23 May, 2012, San Francisco, California, USA, IEEE Computer Society. pp 380–394. https://doi.org/10.1109/SP.2012.31

Chen J, Ma T, Xiao C (2018) Fastgcn: Fast learning with graph convolutional networks via importance sampling. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings, OpenReview.net. https://openreview.net/forum?id=rytstxWAW

Ding SHH, Fung BCM, Charland P (2019) Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019, IEEE. pp 472–489. https://doi.org/10.1109/SP.2019.00003

Eschweiler S, Yakdan K, Gerhards-Padilla E (2016) discovRE: Efficient cross-architecture identification of bugs in binary code. In: 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016, The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/discovre-efficient-cross-architecture-identification-bugs-binary-code.pdf

Feng Q, Zhou R, Xu C, Cheng Y, Testa B, Yin H (2016) Scalable graph-based bug search for firmware images. In: Weippl ER, Katzenbeisser S, Kruegel C, Myers AC, Halevi S (eds). Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. ACM. pp 480–491. https://doi.org/10.1145/2976749.2978370

Goseva-Popstojanova K, Perhinschi A (2015) On the capability of static code analysis to detect security vulnerabilities. Inform Softw Technol 68:18–33. https://doi.org/10.1016/j.infsof.2015.08.002

Grieco G, Grinblat GL, Uzal LC, Rawat S, Feist J, Mounier L (2016) Toward large-scale vulnerability discovery using machine learning. In: Bertino E, Sandhu R, Pretschner A (eds). Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA, March 9-11, 2016. ACM. pp 85–96. https://doi.org/10.1145/2857705.2857720

(2017) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017(Guyon I, von Luxburg U, Bengio S, Wallach HM, Fergus R, Vishwanathan SVN, Garnett R, eds.), Long Beach

Hamilton WL, Ying Z, Leskovec J (2017) Inductive representation learning on large graphs. In: (Guyon et al. 2017). pp 1024–1034. http://papers.nips.cc/paper/6703-inductive-representation-learning-on-large-graphs

He J, Ivanov P, Tsankov P, Raychev V, Vechev MT (2018) Debin: Predicting debug information in stripped binaries. In: Lie D, Mannan M, Backes M, Wang X (eds). Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. ACM. pp 1667–1680. https://doi.org/10.1145/3243734.3243866

Hindle A, Barr ET, Gabel M, Su Z, Devanbu PT (2016) On the naturalness of software. Commun ACM 59(5):122–131. https://doi.org/10.1145/2902362

Kang B, Yerima SY, Sezer S, McLaughlin K (2016) N-gram opcode analysis for android malware detection. IJCSA 1(1):231–255. https://doi.org/10.22619/ijcsa.2016.1001011

Karbab EB, Debbabi M, Derhab A, Mouheb D (2018) Maldozer: Automatic framework for android malware detection using deep learning. Digit Inv 24:S48—S59. https://doi.org/10.1016/j.diin.2018.01.007

Kipf TN, Welling M (2017) Semi-supervised classification with graph convolutional networks. In: (Bengio and LeCun 2017). https://openreview.net/forum?id=SJU4ayYgl

Kolosnjaji B, Zarras A, Webster GD, Eckert C (2016) Deep learning for classification of malware system call sequences. In: Kang BH, Bai Q (eds). AI 2016: Advances in Artificial Intelligence - 29th Australasian Joint Conference, Hobart, TAS, Australia, December 5-8, 2016, Proceedings, Springer, Lecture Notes in Computer Science, vol. 9992. pp 137–149. https://doi.org/10.1007/978-3-319-50127-7_11

Lee T, Choi B, Shin Y, Kwak J (2018) Automatic malware mutant detection and group classification based on the n-gram and clustering coefficient. J Supercomput 74(8):3489–3503. https://doi.org/10.1007/s11227-015-1594-6

Li Y, Gu C, Dullien T, Vinyals O, Kohli P (2019b) Graph matching networks for learning the similarity of graph structured objects. In: Chaudhuri K, Salakhutdinov R (eds). Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA, PMLR, Proceedings of Machine Learning Research, vol. 97. pp 3835–3845. http://proceedings.mlr.press/v97/li19d.html

Li Y, Tarlow D, Brockschmidt M, Zemel RS (2016) Gated graph sequence neural networks. In: Bengio Y, LeCun Y (eds). 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings. http://arxiv.org/abs/1511.05493

Li B, Zhang Y, Yao J, Yin T (2019a) MDBA: detecting malware based on bytes n-gram with association mining. In: 26th International Conference on Telecommunications, ICT 2019, Hanoi, Vietnam, April 8-10, 2019. IEEE. pp 227–232. https://doi.org/10.1109/ICT.2019.8798828

Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) Vuldeepecker: A deep learning-based system for vulnerability detection. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018, The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf

Lin Z, Feng M, dos Santos CN, Yu M, Xiang B, Zhou B, Bengio Y (2017b) A structured self-attentive sentence embedding. In: (Bengio and LeCun 2017). https://openreview.net/forum?id=BJC_jUqxe

Lin G, Zhang J, Luo W, Pan L, Xiang Y (2017a) POSTER: vulnerability discovery with function representation learning from unlabeled projects. In: (Thuraisingham et al. 2017). pp 2539–2541. https://doi.org/10.1145/3133956.3138840

Liu Z, Chen C, Li L, Zhou J, Li X, Song L, Qi Y (2019) Geniepath: Graph neural networks with adaptive receptive paths. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. AAAI Press. pp 4424–4431. https://doi.org/10.1609/aaai.v33i01.33014424

Mikolov T, Chen K, Corrado G, Dean J (2013a) Efficient estimation of word representations in vector space. In: Bengio Y, LeCun Y (eds). 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings. http://arxiv.org/abs/1301.3781

Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J (2013b) Distributed representations of words and phrases and their compositionality. In: Burges CJC, Bottou L, Ghahramani Z, Weinberger KQ (eds). Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8 2013, Lake Tahoe, Nevada, United States. pp 3111–3119. http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality

Monti F, Boscaini D, Masci J, Rodolà E, Svoboda J, Bronstein MM (2017) Geometric deep learning on graphs and manifolds using mixture model cnns. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017. IEEE Computer Society. pp 5425–5434. https://doi.org/10.1109/CVPR.2017.576

Mou L, Li G, Zhang L, Wang T, Jin Z (2016) Convolutional neural networks over tree structures for programming language processing. In: Schuurmans D, Wellman MP (eds). Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, AAAI Press 2016, Phoenix. pp 1287–1293. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775

Murtaza SS, Khreich W, Hamou-Lhadj A, Bener AB (2016) Mining trends and patterns of software vulnerabilities, Vol. 117. https://doi.org/10.1016/j.jss.2016.02.048

Pang Y, Xue X, Namin AS (2015) Predicting vulnerable software components through n-gram analysis and statistical feature selection. In: Li T, Kurgan LA, Palade V, Goebel R, Holzinger A, Verspoor K, Wani MA (eds). 14th IEEE International Conference on Machine Learning and Applications, ICMLA 2015, Miami, FL, USA, December 9-11, 2015. IEEE. pp 543–548. https://doi.org/10.1109/ICMLA.2015.99

Perozzi B, Al-Rfou R, Skiena S (2014) Deepwalk: online learning of social representations. In: Macskassy SA, Perlich C, Leskovec J, Wang W, Ghani R (eds). The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014. ACM. pp 701–710. https://doi.org/10.1145/2623330.2623732

Rawat S, Mounier L (2012) Finding buffer overflow inducing loops in binary executables. In: Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012. IEEE. pp 177–186. https://doi.org/10.1109/SERE.2012.30

Ray B, Hellendoorn V, Godhane S, Tu Z, Bacchelli A, Devanbu PT (2016) On the "naturalness" of buggy code. In: Dillon LK, Visser W, Williams L (eds). Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. ACM. pp 428–439. https://doi.org/10.1145/2884781.2884848

Santos I, Penya YK, Devesa J, Bringas PG (2009) N-grams-based file signatures for malware detection(Cordeiro J, Filipe J, eds.)

Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng S, Hauser C, Krügel C, Vigna G (2016) SOK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. IEEE Computer Society. pp 138–157. https://doi.org/10.1109/SP.2016.17

Simonyan K, Zisserman A (2015) Very deep convolutional networks for large-scale image recognition. In: Bengio Y, LeCun Y (eds). 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings. http://arxiv.org/abs/1409.1556

Theisen C, Herzig K, Morrison P, Murphy B, Williams LA (2015) Approximating attack surfaces with stack traces. In: Bertolino A, Canfora G, Elbaum SG (eds). 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2. IEEE Computer Society. pp 199–208. https://doi.org/10.1109/ICSE.2015.148

Thuraisingham BM, Evans D, Malkin T, Xu D (2017) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. ACM. https://doi.org/10.1145/3133956

Velicheti LMR, Feiock DC, Peiris M, Raje RR, Hill JH (2014) Towards modeling the behavior of static code analysis tools. In: Abercrombie RK, McDonald JT (eds). Cyber and Information Security Research Conference, CISR '14, Oak Ridge, TN, USA, April 8-10, 2014. ACM. pp 17–20. https://doi.org/10.1145/2602087.2602101

Velickovic P, Cucurull G, Casanova A, Romero A, Liò P, Bengio Y (2017) Graph attention networks. CoRR:abs/1710.10903. http://arxiv.org/abs/1710.10903

White M, Tufano M, Vendome C, Poshyvanyk D (2016) Deep learning code fragments for code clone detection. In: Lo D, Apel S, Khurshid S (eds). Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016. ACM. pp 87–98. https://doi.org/10.1145/2970276.2970326

Wu S, Wang P, Li X, Zhang Y (2016) Effective detection of android malware based on the usage of data flow apis and machine learning. Inform Softw Technol 75:17–25. https://doi.org/10.1016/j.infsof.2016.03.004

Wüchner T, Ochoa M, Pretschner A (2015) Robust and effective malware detection through quantitative data flow graph metrics. In: Almgren M, Gulisano V, Maggi F (eds). Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings, Springer, Lecture Notes in Computer Science, vol 9148. pp 98–118. https://doi.org/10.1007/978-3-319-20550-2_6

Xu X, Liu C, Feng Q, Yin H, Song L, Song D (2017) Neural network-based graph embedding for cross-platform binary code similarity detection. In: (Thuraisingham et al. 2017). pp 363–376. https://doi.org/10.1145/3133956.3134018

Yamaguchi F, Golde N, Arp D, Rieck K (2014) Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. IEEE Computer Society. pp 590–604. https://doi.org/10.1109/SP.2014.44

Zalewski M (2017) American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/

Zhang Y, Shen D, Wang G, Gan Z, Henao R, Carin L (2017) Deconvolutional paragraph representation learning. In: (Guyon et al. 2017). pp 4169–4179. http://papers.nips.cc/paper/7005-deconvolutional-paragraph-representation-learning

Zhou Y, Liu S, Siow JK, Du X, Liu Y (2019) Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks(Wallach HM, Larochelle H, Beygelzimer A, d'Alché-Buc F, Fox EB, Garnett R, eds.), Vancouver. http://papers.nips.cc/paper/9209-devign-effective-vulnerability-identification-by-learning-comprehensive-program-semantics-via-graph-neural-networks

Zuo F, Li X, Young P, Luo L, Zeng Q, Zhang Z (2019) Neural machine translation inspired binary code similarity comparison beyond function pairs. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019, The Internet Society. https://www.ndss-symposium.org/ndss-paper/neural-machine-translation-inspired-binary-code-similarity-comparison-beyond-function-pairs/

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.