Towards Learning Representations of Binary Executable Files for Security Tasks

Shushan Arakelyan, Christophe Hauser, Erik Kline, Aram Galstyan
Information Sciences Institute, University of Southern California
4676 Admiralty Way #1001
Marina Del Rey, California 90292
shushan@isi.edu

Abstract
Tackling binary analysis problems has traditionally implied manually defining rules and heuristics. As an alternative, we are suggesting using machine learning models for learning distributed representations of binaries that can be applicable for a number of downstream tasks. We construct a computational graph from the binary executable and use it with a graph convolutional neural network to learn a high dimensional representation of the program. We show the versatility of this approach by using our representations to solve two semantically different binary analysis tasks – algorithm classification and vulnerability discovery. We compare the proposed approach to our own strong baseline as well as published results, and demonstrate improvement on state of the art methods for both tasks.

Introduction
In recent years we have seen a big surge in applications of machine learning 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.

For many of the security problems, researchers are dealing directly with binary executable files. Two of the main related challenges are the size of binary programs and the absence of high-level semantic structure in binary (executable) code: the compilation process of going from source code to binary executable programs preserves only basic low-level instructions and data representations understood by the target CPU. 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 models has a high potential for improving performance while keeping the analysis fast.

Using machine learning requires obtaining a 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. 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. Word2vec, 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 embeddings 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 are useful for a range of new computer vision tasks.

We see an important argument for learning 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 an ongoing effort in natural language processing. 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++ programs
– 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. We propose a graph-based representation for binaries, that when used with a Graph Convolutional Network (GCN) (Kipf and Welling 2017), captures the semantics of the program.

Our main contributions are: (i) To the best of our knowledge we are the first to suggest distributed representation learning model for binary executable programs that can be fine-tuned to different downstream tasks; (ii) For that 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 - algorithm classification and vulnerability discovery, and show improvement on state of the art on both.

Related Work

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 program binary 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 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 literature for malware discovery and analysis (??), as well as vulnerability discovery (??). Additionally, there have been efforts focusing on extracting relevant API calls or using traces of system calls to detect malware (??).

Graph-based features Many solutions rely on extracting 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, discoverRE (??), among other things, uses closeness of control flow graphs to compute similarity between functions. Genius (??) 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 (??).

Learned Features

Besides manually crafting the representations it is also possible to employ neural models for that purpose. Among other things 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 (??; Ray et al. 2016) 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 (??). Some of previous vulnerability detection efforts also use RNNs on lines of assembly code (Li et al. 2018). More recently, INNEREYE used neural machine translation for binary code similarity detection (??).

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 (Xu et al. 2017; Li et al. 2019; Ding, Fung, and Charland 2019). Researchers have also used Conditional Random Fields on an enhanced Control Flow Graph for attempting to recover the debug information of the binary program (He et al. 2018).

Task Description

We evaluate the performance of our proposed representations on two independent tasks. In the first, we test the proposed representations for algorithm recognition. 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 semi-structurally 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: Algorithm Classification

Algorithm classification is crucial for semantic analysis of code. It can be used for creating assisting tools for security researchers to understand and analyse 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.

Written independently by different students, programs solving the same problem have differences in implementation, in data structures used, and in ways the code was split into functions. In some rare cases, the same problem can be solved by more than one algorithm.
In this task, we test the ability of the model to capture the higher-level semantic similarity, and to take into account program behavior, 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 the system 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. It should be emphasised that vulnerability discovery is unrelated to malware analysis, which is the study of programs specifically crafted to behave in a malicious way. Our work excludes malware, and focuses on benign programs.

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 discovering 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.\

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 networks work. We also describe the baseline model that we use for evaluation and comparison.

Program Graphs

We start by disassembling the binary program and constructing a control flow graph (CFG). A CFG is a representation of the program, where each node is a linear sequence of commands (representing a so-called basic block). A basic block does not contain any control-flow instructions, jumps or jump targets within it. Jump targets appear only at the beginning of the block and jumps - at the end. We use static inter-procedural CFGs in our work, which we construct using the angr library (Shoshitaishvili et al. 2016). After constructing the CFG we lift all basic blocks to VEX Intermediate Representation (IR). The fact that each block is executed linearly allows us to unfold the instructions within each block and represent them as a directed, computational tree, similar to an Abstract Syntax Tree (AST). Every node of the resulting tree corresponds to a constant, a register, a temporary or an operation. The edges of the tree are directed from the argument to the instruction. Within each block we reuse nodes that correspond to addresses, temporaries, constants and registers to tie together potentially related computations (thus preserving information about local data-flow within the scope of each basic block). The VEX IR provides a Static Single Assignment form (SSA). This means an assembly instruction is lifted to IR instructions operating on temporary variables which are each used only once. However, VEX does not track instances of different definitions/uses of the same register across instructions on the basic block level, which we implemented to ensure we do not introduce fake data-dependence edges. In our implementation, if an instruction overrides (redefines) the content of register eax, our analysis renames it to eax_1 and so on. Thus, we do not reuse the same node for eax and eax_1.

Within each block computations do not necessarily all depend on each other. We may have chunks of code that can be reordered inside the block without affecting the final result. In this case our described approach produces a forest of computations. To connect these trees we add virtual Source and Sink nodes at the beginning and at the end of each block as a parent, or correspondingly a child, for all the trees generated from that block. Table 1 demonstrates how the same code looks in C, assembly and VEX IR. Figure 1 shows how IR is translated to a graph. We additionally remove redundant edges and nodes, particularly, Iex_Const node that follows every constant, and chains of Iex_WrtTmp → ‘t%’ → Iex_RdTmp, a more concise resulting graph is shown in Figure 2. After constructing graphs for every block of CFG, we connect virtual Sink and Source nodes following the topology the blocks had in the CFG.

Before feeding the constructed graph into the neural network, we remove SSA indices for temporary variables and registers to reduce sparsity. The remaining names of instructions, registers, constants and generic ‘t’ for temporaries are encoded in a one-hot manner and used as a features matrix.

<table>
<thead>
<tr>
<th>Source code</th>
<th>Decompiled assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>// init a and b</td>
<td>mov edx, dword ptr [rbp-8]</td>
</tr>
<tr>
<td>...</td>
<td>add edx, eax</td>
</tr>
<tr>
<td>...</td>
<td>sub edx, eax</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Lifted to VEX IR

<table>
<thead>
<tr>
<th>Source</th>
<th>Decompiled assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: c = a - b</td>
<td>t4 = GET(i64(dbs)) t7 = 32Uto64(t3)</td>
</tr>
<tr>
<td>2: t3 = 64to32(t4)</td>
<td>8: PUT(c, dep1) = t7</td>
</tr>
<tr>
<td>3: t6 = GET(i64(iax))</td>
<td>9: t6 = 32Uto64(t5)</td>
</tr>
<tr>
<td>4: t5 = 64to32(t6)</td>
<td>10: PUT(c, dep2) = t8</td>
</tr>
<tr>
<td>5: t0 = Sub32(t3, t5)</td>
<td>11: t9 = 32Uto64(t0)</td>
</tr>
<tr>
<td>6: PUT(c, ap) = 0x7</td>
<td>12: PUT(dbs) = t9</td>
</tr>
</tbody>
</table>

Table 1: The table demonstrates a line of source code that performs subtraction, how it is decomposed into assembly (after compilation), and how it is lifted to VEX IR.
ular and frequently used for a wide range of security tasks,
other task-independent model. Since NLP methods are pop-
Baseline
olution for the optimization. Our GCN had three graph convo-
diction. We use the cross-entropy error as the objective func-
Graph Convolutional Networks
The model we used for learning representations is a multi-
layer Graph Convolutional Neural Network (GCN) (Kipf
and Welling 2017), which has achieved state-of-the-art per-
ficiency on a number of benchmark graph datasets. GCN
consists of a few stacked graph convolutional layers. Intu-
itively, k-th graph convolutional layer filters and propagates
information from k-th neighborhood of the node. GCN uses
the adjacency matrix of the graph A and its feature matrix
X to generate representations of each node in the graph,
Z. Additional details and intuitions as to why convolving
on a graph in this manner creates a good representation of
its nodes are presented in the original GCN paper. Here we
briefly review the propagation rule and the final form of the
model.

The propagation through (l + 1)-st layer looks like:

\[
H^{(l+1)} = \text{ReLU}(D^{\frac{1}{2}} \hat{A} D^{\frac{1}{2}} H^{(l)} W^{l})
\]

where \(\hat{A}\) is the adjacency matrix 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.

Propagating information through graph convolutional lay-
ers outputs a representation for each individual node. To get
the representation of the entire graph, we aggregated the rep-
resentations of every node in the graph via summation.
We present a schematic illustration of this process in Figure 3.

The resulting aggregated representation is passed through
another fully-connected layer, followed by a softmax which
is defined like \(\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum \exp(x_j)}\), for the final pre-
diction. We use the cross-entropy error as the objective
function for the optimization. Our GCN had three graph con-
volutional layers and produced representations of size 64.

Baseline
We wanted to compare our proposed representation with an-
other task-independent model. Since NLP methods are pop-
ular and frequently used for a wide range of security tasks,
we chose a bag-of-words representation to be used with a
Support Vector Machine (SVM) classifier. In particular, we
used the SVM with a Gaussian kernel.

For bag-of-words representation we chose to use VEX
IR from angr. We considered each line of IR to be a sin-
gle “word”. Using IR offers a more consistent comparison
with our proposed representation. Additionally, we found
empirically that using lines of IR as features works better
than both using lines of assembly code, and tokenized as-
sembly. Vocabulary for the bag-of-words was obtained from
the training data. We used frequency thresholding to remove
infrequent entries and reduce data sparsity. Those frequen-
cies were empirically tuned on validation data.

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 De-
bian operating system, using gcc 8.3, without any optimiza-
tion 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 into their corresponding
problems.

The second dataset we used is the Juliet C/C++ test
suite (Boland and Black 2012). This is a synthetically gen-
erated dataset, created to facilitate research of vulnerabil-
ity 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
\textit{OMITGOOD} and \textit{OMITBAD} macros, surrounding vul-
nerable and non-vulnerable functions correspondingly. We
compiled the dataset twice - once with each macro, to gen-
erate 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 ex-
amples, that we omitted. Note that even though our approach
is not platform-specific, in this work we limit our experimen-
tation to Linux only.

Most CWE-IDs had too few examples to train a classi-
ifier on and/or to report any meaningful statistics on. We also
omitted any CWE-IDs that had less than 100 files in their
test set after splitting the data with 70:15:15 ratio for train-
ing, validation and test, because for those cases any reported
result would be too noisy.

As a result, we experimented on vulnerabilities from 30
CWE-IDs. We trained a separate classifier for every indi-
individual CWE-ID, which was required because files in each 
CWE-ID directory may or may not contain other vulnera-

ty 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 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 
train set for training and extracting some additional helper 
structures, such as vocabulary for the bag of words mod-
els and counting frequencies for thresholding in neural net-
work 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 test set. The ex-
periments are cross-validated and averaged over 5 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 train 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 neural network representation, we follow similar logic 
and use the train set to find and remove infrequent node la-


beles. Here too the exact threshold is decided via experimen-

tation 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 hexadeci-


mal.

Task 2. Experimental Setup

In the vulnerability discovery experiments, we train a sepa-
rate classifier for each of 30 different CWE-IDs. 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 train-
ing the models and validation sets for grid search of the


penalty parameter C in SVMs. We report the performance 
of the best model on test sets.

Here we reuse some statistics obtained on the first dataset, 
in particular, we reuse frequency thresholds and bag-of-
words vocabularies. We do this because some CWE-IDs 
have much larger number of binary executables than others. 
To account for this we would need to search for frequency 
thresholds in a very large range. Since we need to train a sepa-
rate classifier for each CWE-ID, so 30 SVM classifiers and 
30 NN classifiers in total, that would lead to a huge search 
space at the phase of the model selection. Additionally, de-
pending on how many examples we have in the training set, 
the expressive power of the model would vary a lot.

Evaluation and Results

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

Task 1

Table 2 contains quantitative evaluation of our represen-
tation for the Tasks 1. Our proposed representation out-
performs our own SVM baseline, TBCNN model(Mou 
et al. 2016), and current state-of-the-art for this task - 
inst2vec(Ben-Nun, Jakobovits, and Hoefler 2018). We man-
age to reduce the error by more than 40%, thus setting a new 
state-of-the-art result. It should be additionally mentioned

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM on VEX IR</td>
<td>0.93</td>
</tr>
<tr>
<td>TBCNN</td>
<td>0.94</td>
</tr>
<tr>
<td>inst2vec</td>
<td>0.9483</td>
</tr>
<tr>
<td>Ours</td>
<td>0.97</td>
</tr>
</tbody>
</table>

Table 2: Accuracy obtained for the task of online judge problem classification. We show results reported for TBCNN and inst2vec, though original splits for their experiments are not available.
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.¹

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 hand, 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 4 contains evaluation of our representation 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 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, (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 where the commercial tool scores a few percents higher, and, again, CWE-ID590.

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**

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 the program in Juliet test suite is relevant for vulnerability discovery task, 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 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.

Conclusion

In this paper we proposed a method for learning distributed representations for binary executable programs. Our learned representation has a potential to be used for 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, compared to other state of the art methods, including common machine learning baselines.

References


