Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis

Luca Massarelli
University of Rome, Sapienza
massarelli@diag.uniroma1.it

Giuseppe A. Di Luna
CINI,
National Laboratory of Cybersecurity
g.a.diluna@gmail.com

Fabio Petroni*
Facebook AI Research
petronif@acm.org

Leonardo Querzoni
University of Rome, Sapienza
querzoni@diag.uniroma1.it

Roberto Baldoni**
Italian Presidency of Ministry Council
baldoni@diag.uniroma1.it

***Abstract*—In this paper we investigate the use of graph embedding networks, with unsupervised features learning, as neural architecture to learn over binary functions.**

We propose several ways of automatically extract features from the *control flow graph* (CFG) and we use the structure2vec graph embedding techniques to translate a CFG to a vectors of real numbers. We train and test our proposed architectures on two different binary analysis tasks: binary similarity, and, compiler provenance. We show that the unsupervised extraction of features improves the accuracy on the above tasks, when compared with embedding vectors obtained from a CFG annotated with manually engineered features (i.e., ACFG proposed in [39]).

We additionally compare the results of graph embedding networks based techniques with a recent architecture that do not make use of the structural information given by the CFG, and we observe similar performances. We formulate a possible explanation of this phenomenon and we conclude identifying important open challenges.

I. Introduction

The current evolution trends in all industrial sectors, favored by phenomena like the Internet of Things and evolutionary programs like Industry 4.0, strongly push for a large number of heterogeneous common-off-the-shelf devices to be integrated within the IT systems of large organizations. While this approach is supposed to improve efficiency by cutting costs and enabling new "smart" production processes, it raises several concerns related to the security of such systems. Security teams that work to guarantee the safe and correct functioning of such systems, need to monitor a large number of interconnected devices running closed-source software, without any clue about their internals (but for the often scarce and rarely updated documentation provided by vendors). This problem is today raising the demand for new methodologies and tools to analyze software in its binary form, the only form such software is often available to security analysts.

The academic and industrial communities working in the areas of software engineering and programming languages have recently started to investigate new promising approaches based on the statistical modelling of software source code. The idea at the heart of these approaches is to leverage the huge, readily available source base that is represented by open-source software, to train models though machine learning techniques. A recent survey [2] provides a detailed overview of this, so called, Big Code movement. A fertile line in this area of research is to use and adapt solutions from Natural Language Processing (NLP) that, coupled with deep neural networks, have been extremely successful in providing state-of-the-art performance for several tasks: translation of source code between different languages [12], detection of duplicate semantic [10], variable misuses detection [3] and many others [5], [11], [17]. Unfortunately, such techniques are tailored to the analysis of source code and cannot be immediately used on binary code. The source code compilation process, for instance, destroys variables and functions naming, as well as variable types, thus hampering the applicability of Big Code techniques. We believe that transferring the Big Code knowledge in the vast field of binary analysis is of urgent interest, and such knowledge can be a precious aid in the design of new, effective, and, easy to use binary analysis tools.

To the best of our knowledge, very few works have used NLP techniques to analyse binary code [14], [21], [40]. Notably, [21] and [39] propose techniques based on embedding the CFG in order to solve the binary similarity task. In [39], as in [23], the CFG is first transformed into an ACFG, that is CFG where the blocks are vector manual engineered features, and then such ACFG is embedded, using the graph embedding architecture proposed in [16], into a vector.

Building upon the results of these previous works, in this paper we investigate several techniques to represent the blocks of the CFG, without using manual selected features, and we

∗ This paper was researched and written prior to joining Facebook
∗∗ On leave from University of Rome, Sapienza

Workshop on Binary Analysis Research (BAR) 2019
24 February 2019, San Diego, CA, USA
ISBN 1-891562-58-4
https://dx.doi.org/10.14722/bar.2019.23020
www.ndss-symposium.org

examine how these techniques perform when they are used end-to-end with a graph embedding architecture. In fact, a really successful recent trend in machine learning is to design network architectures that introduce the smallest possible human bias and that automatically learn the representations needed from raw data [8], [30]. We argue that manually selecting vertex features is prone to injecting human bias and could potentially fail in capturing non-obvious syntactic and semantic structure in the binary code.

Our final aim is to understand the suitability of supervised architectures working on graph embeddings for common binary analysis tasks. To this end we consider tasks that are markedly different from each others, having the specific purpose of testing the flexibility and the usefulness of various graph embeddings techniques, and, at large, on the use of the bare CFG as an aid to learn the syntactic structure of binaries by deep neural networks [3].

Specifically, we test our solutions on the tasks of binary similarity and compiler provenance:

1) Binary Similarity —In this task we train our model to map CFGs into vectors of numbers (i.e. *embeddings*), in such a way that similar graphs result in similar vectors. We assume that two CFGs are similar if they are derived from the same source code. As already pointed out in [39], this assumption does not make the problem trivial. The complexity lies in the fact that, starting from the same source code, widely different binaries can be generated by different compilers with several optimisation parameters. To make things more challenging the same source code could be compiled targeting different architectures that use completely different instruction sets (in particular we consider AMD64 and ARM as target architectures for our study). The binary similarity task is known to have several practical use cases in the field of security like vulnerability detection in closed-source software and the phylogenic analysis of malware.
2) Compiler Provenance —In this task we train our model to map a CFG into a class representing the compiler family that generated it. This classification task identifies the toolchain used to generate unknown binaries and produces information that is required by specific library detection tools such as IDA FLIRT[4].

The main contributions of our work are the following:

- we describe a general network architecture for calculating binary function embeddings starting from the corresponding CFGs that extends the one introduced in [39] (Section IV);
- we introduce several designs for the *Vertex Features Extractor*, a component that associates a feature vector to each vertex in the CFG. These designs make use of unsupervised learning techniques, i.e. they do not introduce any human bias (Section IV-A);
- we report on an experimental evaluation conducted on these different feature extraction solutions considering the binary similarity task and the compiler provenance task; we show that unsupervised feature extraction is better, on both tasks, than manually engineered features (Section V);
- we discuss our findings in Section VI. We note that despite taking into account the syntactic structure of code using the CFG our techniques underperform or have comparable performances, on both task, when compared with a solution [29] that examine sequentially all the disassembled instructions, without information on the control flow given by the CFG. We discuss our hypothesis on this phenomena, giving a possible explanation on the shortcomings of blindly embedding the CFG.

Finally, in the conclusions (Section VII) we propose two interesting open challenges in this field.

II. Related work

Several works have investigated the use of deep neural networks for binary analysis tasks like finding functions [7], [24], [35]. At the best of our knowledge the first work that has introduced NLP concepts in the field of binary analysis is [14]. The paper introduces the use of the distributed representation of instructions, that is word2vec [30], for the purpose of learning function signature in binaries using a Recurrent Neural Network (RNN). We are not aware of works that have investigated the use of graph embedding neural networks automatically extracting the features from CFG blocks. In the following we details the literature of the two tasks that we investigate, specifically focusing on works using deep neural networks.

A. Binary Similarity based on embeddings

Among all works on binary similarity [19] [27] [31] the most related are the ones that propose embeddings for binary similarity, or that use deep neural networks. We can divide the works in single-architecture (that are able to compute the similarity only for binaries compiled for the same architecture, i.e. AMD64) and cross-architecture (that are able to compute the similarity among binaries compiled for two or more architectures, i.e. ARM and AMD64).

a) Single-Architecture solutions: — Recently, [21] proposed a function embedding solution named *Asm2Vec*. Asm2Vec builds a series of instruction traces by performing random-walks over a function CFG, and then it embeds such traces using a variation of the PV-DM model [28] for natural language processing.

b) Cross-Architecture solutions: — Feng et al. [23] computed the embeddings by using a clustering approach: they first obtained clusters of training functions, then, they used centroids of such clusters and a feature encoding technique to associate an embedding vector to each function. Xu et al. [39] proposed function embeddings that are computed using a deep neural network. Interestingly, [39] shows that the proposed architecture, namely *Gemini*, outperforms [23].

[3] The works on source code have shown that, in most cases, being aware of the syntactic structure of code leads to better performances [2]. The majority of them takes into account the syntactic structure by training the model on the AST or some derived graph [3], [5]

[4]https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml

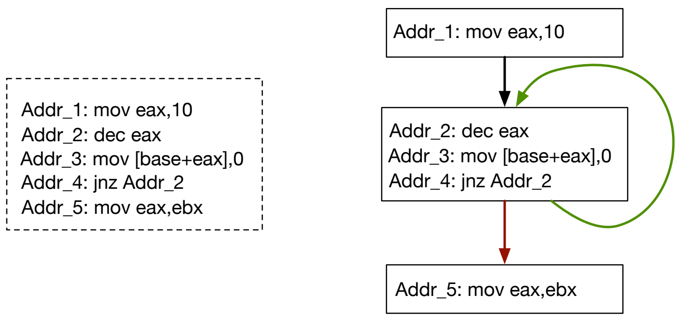


Fig. 1. Assembly code on the left and corresponding cfg graph on the right.

In Gemini the CFG of a function is first transformed into an *annotated CFG* [22] [32], a graph containing manually selected features, and then embedded into a vector using the graph embedding model of [16].

Finally, in [40] a recurrent neural network based on LSTM (Long short-term memory) is used to solve a sub-task of binary similarity: finding similar CFG blocks.

B. Compiler Provenance

The compiler provenance problem was first afforded by Rosenblum et al. [34]. They proposed a solution to infer the compiler that produces a given executable using *idioms*, short sequences of instructions that can be matched into an executable. In [33] the same authors outperformed their previous solution combining idioms with *graphlets*. Rahimian et al. [32] proposed an approach based on annotated control flow graph (ACFG) to recover the compiler provenance of a binary, however their approach seems to be less accurate than [33]. Recently, Chen et al [13] proposed a deep neural network that recovers the optimization level of different functions compiled with gcc; they show that the learned model is explicable as it learn common compiler convention.

III. Problem definition and solutions overview

Binary similarity and compiler provenance — We say that two binary functions f_1^s, f_2^s are similar, $f_1 \sim f_2$, if they are the result of compiling the same original source code s with different compilers. We see the compiler c has a deterministic transformation mapping a source code s into a binary function f_c^s. For us a compiler is the specific software, e.g. gcc-5.4.0, together with the flags that influence the compiling process, e.g. the optimization flags -O$[0, ..., 3]$. With the term *compiler family*, we mean the specific compiler without considering the version number, i.e. the family of `gcc-5.4.0` is `gcc`, the family of `clang-3.4` is `clang`.

In the compiler provenance task we are given a set of possible compilers $C : \{c_1, c_2, \ldots\}$, and a a binary function $f_{c_j}^s$ and we have to guess the compiler family $F(c_j) | c_j \in C$.

Control Flow Graph — Given a binary function, we use its representation as *control flow graph* (CFG) [4]. A control flow graph is a directed graph whose vertices are sequences of assembly instructions, and whose arcs represent the execution flow among vertices. In Figure 1 there is a binary code snippet and the corresponding CFG. We reduce the binary similarity problem to the one of finding similar CFGs. Thus when we say that two CGFs are similar we mean that the corresponding functions are similar (the same holds for dissimilar CFGs).

We denote a CFG as $g = (V, E)$ where V is a set of vertices and E a set of edges. $\mathcal{N}(v_i)$ is the set of neighbors of vertex $v_i \in V$. Vector x_i is a d-dimensional features vector associated with vertex v_i, and I_{v_i} is the set of instructions contained in vertex v_i. Without loss of generality, we assume that all vertices contain the same number of instructions m[5].

Overview of the solution — Our aim is to feed a CFG to a deep neural network to solve the tasks of binary similarity and compiler provenance. This is achieved using an embedding neural network that maps a CFG g into an *embedding vector* $\vec{g} \in \mathbb{R}^n$. The first component of this network is a *Vertex Features Extraction* mechanism that automatically generates a feature vector x_i for a vertex v_i in the CFG using the set of instructions I_{v_i}.

For the binary similarity task we want that vector $\vec{g}$ preserves the structural similarity relation between CFGs, and thus between binary functions. That is, two vectors representing similar CFGs should be close in the metric space. For the task of compiler provenance we want that vector $\vec{g}$ contains structural information about the compiler that generated the CFG, in such a way that $\vec{g}$ can be feed to a classifier, a deep feed-forward neural network in our case.

We give a general overview of the approaches analyzed for the Vertex Feature Extraction component, that can be divided in the two broad families, namely (A) manual feature engineering and (B) unsupervised feature learning.

a) Manual Feature Engineering (MFE): This is the approach defined in [39] and represents the baseline in our tests. The feature vector x_i of vertex v_i is computed by counting the number of instructions belonging to predefined classes (e.g. transfer instructions), together with the number of strings and constants referred in v_i, and the offspring and betweenness centrality of v_i in the CFG. Note that this mechanism is not modified by the training procedure of the neural network, i.e. it is not *trainable*.

b) Unsupervised Feature Learning: The main idea of this family of approaches is to map each instruction $\iota \in I_{v_i}$ to vectors of real numbers $\vec{\iota}$, using the *word2vec* model [30], an extremely popular feature learning technique in NLP. We use a large corpus of instructions to train our instruction embedding model (see Section V-A) and we call our mapping *instruction2vec* (i2v). Note that such instruction vectors can be both kept *static* throughout training or updated via back-propagation (*non-static*) by the network. To generate a single feature vector x_i for each vertex v_i we considered two different strategies to aggregate the instruction embeddings:

- i2v_attention: the main idea of this aggregation strategy is to use an attention mechanism to learn which instructions are the most important for the classifier according to their position. In particular, we include in the network a vector a that contains weights. The features x_i of vertex v_i are computed as a weighted

[5]Concretely speaking, this is achievable by adding `NOP` padding instructions to vertices that contain less than m instructions.

mean of the vectors $\vec{\iota}$ of instructions $\iota \in I_{v_i}$. Note that the weights vector a is trained end-to-end with the other network hyper-parameters. **Rationale:** this strategy takes inspiration from recent works in neural machine translation [6]. The presence of vector a allows the network to automatically decide the importance of instructions relatively to their position inside each vertex.

- i2v_RNN: the feature x_i of v_i is the last vector of the sequence generated by a Recurrent Neural Network (RNN) that takes as input the sequence of vectors $\vec{\iota}$ of instructions $\iota \in I_{v_i}$. The RNN we consider in this work is based on GRU cell [15]. This mechanism is trainable, the weights of the RNN are updated with the training of Structure2Vec. **Rationale:** this method generates a vector representation that takes into account the order of the instructions in the input sequence.

IV. Graph Embedding Neural Network

We denote the entire network that compute the embedding of a CFG graph as *graph embedding neural network*. The graph embedding neural network is the union of two main elements: (1) the *Vertex Feature Extraction* component, that is responsible for associating a feature vector with each vertex in g, and (2) the Structure2Vec network, that combines such feature vectors through a deep neural architecture to generate the final embedding vector of g. See Figure 2 for a schematic representation of the overall architecture of the graph embedding network, where the Vertex Feature Extraction component refers to an Unsupervised Feature Learning implementation.

A. Vertex Features Extraction

The Vertex Feature Extraction is the component where we focus the attention of this paper and where we propose most of our contributions. The goal of this component is to generate a vector x_i from each vertex v_i in the CFG g. We considered several solutions to implement this component. As baseline we consider the approach based on manual feature engineering proposed in [39]. Moreover, we investigated solutions based on unsupervised feature learning (or representation learning), borrowing models and ideas from the natural language processing community. These techniques allow the network to automatically discover the representations needed for the feature vectors from raw data. This replaces manual feature engineering and allows the network to both learn the features and use them to generate the final graph embedding. We will show in Section V that this approach leads to performance improvements of the overall graph embedding network. Finally, we tested a solution that executes the different part of the CFG g using a synthetic input and samples the results to generate the feature vector x_i, this solution is only discussed in the appendix for space constraint and because is the one that has shown the worst performances in our evaluation.

Manual feature engineering (MFE)

As in [39] for each block we use the following features:

1) Number of constants;
2) Number of strings;
3) Number of transfer instructions (e.g. *MOV*);
4) Number of calls;
5) Number of instructions;
6) Number of arithmetic instructions (e.g. *ADD*);
7) Vertex off-springs;
8) Vertex betweenness centrality;

The first six features are related to the code of the block. Instead, the last two features depend on the CFG, hence they bring some information about the structure of the control flow graph inside each vertex.

Unsupervised feature learning

This family of techniques aim at discovering low-dimensional features that capture the underline structure of the input data. The first step of these solutions consist in associating an embedding vector with each instruction ι contained in I_{v_i}. In order to achieve this we train an embedding model i2v using the skip-gram method outlined in the paper that introduces word2vec technique for computing word embeddings [30]. The idea of the skip-gram model is to use the current instruction to predict the instructions around it. A similar approach has been used also in [14].

We use the mnemonics and the operands of each assembly instruction as tokens to train the i2v model. Note that, we filter the operands of each instruction and we replace all base memory addresses with the special symbol `MEM` and all immediates whose absolute value is above some threshold (we use 5000 in our experiments, see Section V-A) with the special symbol `IMM`. The motivation behind this choice is that we believe that using raw operands is of small benefit, e.g. the relative displacement given by a jump is useless (e.g., instructions do not carry with them their memory address), and, on the contrary, it may decrease the quality of the embedding by artificially inflating the number of different instructions. As example the instruction `mov EAX,6000` becomes `mov EAX,IMM`, `mov EAX,[0x3435423]` becomes `mov EAX,MEM`, while the instruction `mov EAX,[EBP−8]` is not modified. Intuitively, the last instruction is accessing a stack variable different from `mov EAX,[EBP−4]`, and this information remains intact with our filtering.

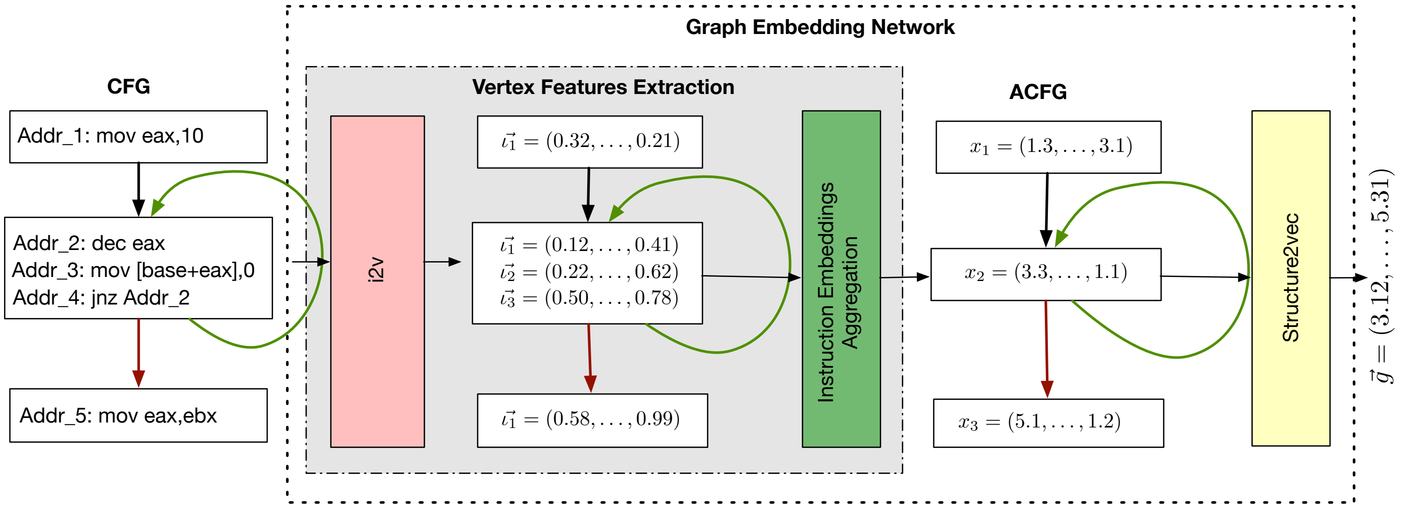
After obtaining an embedding for each instruction we still need to aggregate such vectors in order to obtain a single feature vector x_i to associate with vertex v_i. In this paper, we investigate two different instruction embeddings aggregation techniques: i2v_attention, i2v_RNN.

i2v_attention — In i2v_attention we compute a weighted average of the instructions I_v using an end-to-end trained vector a that associates a different weight with each instruction position. The feature vector x_i is:

$$x_i = \frac{\sum_{j=1}^{m} a[j] \cdot \vec{\iota_j}}{|| \sum_{j=1}^{m} a[j] \cdot \vec{\iota_j} ||} \qquad (1)$$

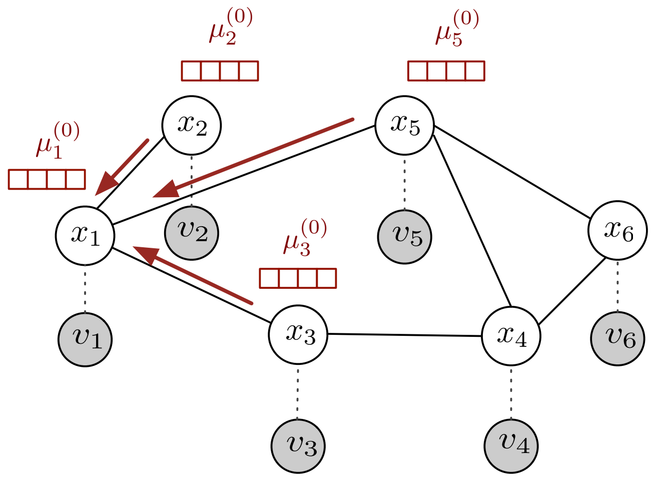
where $a[j]$ is the j-th component of vector a.

i2v_RNN — To fully take into consideration the instruction position we also considered a solution that incorporates a Recurrent Neural Network (RNN) [26] into the overall network architecture. This RNN is trained end-to-end, takes in input all the instruction embedding vectors in order, i.e. $(\vec{\iota_1}, ..., \vec{\iota_m})$ and

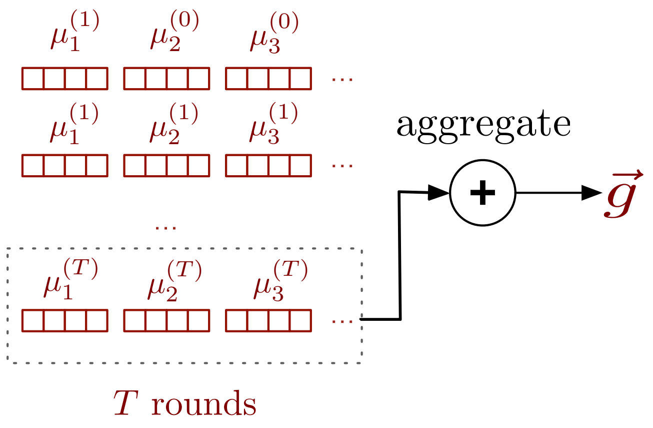


Graph Embedding Network
CFG
Addr_1: mov eax,10
Addr_2: dec eax
Addr_3: mov [base+eax],0
Addr_4: jnz Addr_2
Addr_5: mov eax,ebx
Vertex Features Extraction
i2v
$\vec{\iota_1} = (0.32, \dots, 0.21)$
$\vec{\iota_1} = (0.12, \dots, 0.41)$
$\vec{\iota_2} = (0.22, \dots, 0.62)$
$\vec{\iota_3} = (0.50, \dots, 0.78)$
$\vec{\iota_1} = (0.58, \dots, 0.99)$
Instruction Embeddings Aggregation
ACFG
$x_1 = (1.3, \dots, 3.1)$
$x_2 = (3.3, \dots, 1.1)$
$x_3 = (5.1, \dots, 1.2)$
Structure2vec
$\vec{g} = (3.12, \dots, 5.31)$

Fig. 2. Graph Embedding Neural Network Architecture. The vertex feature extractor component refers to the Unsupervised Feature Learning case.



(a) Vertex-specific μ_v vectors are updated according to graph topology and vertex features.



$\mu_1^{(1)}$ $\mu_2^{(0)}$ $\mu_3^{(0)}$ ...
$\mu_1^{(1)}$ $\mu_2^{(1)}$ $\mu_3^{(1)}$ ...
...
$\mu_1^{(T)}$ $\mu_2^{(T)}$ $\mu_3^{(T)}$ ...
T rounds
aggregate
$\vec{g}$

(b) The final graph embedding $\vec{g}$ is obtained by aggregating together the vertex μ vectors after T rounds.

Fig. 3. Structure2Vec Deep Neural Network

generates m outputs and m hidden states $(h^{(1)}, ..., h^{(m)})$. The final feature vector x_i is simply the last hidden state of the RNN, that is:

$$x_i = h^{(m)} \tag{2}$$

B. *Structure2Vec deep neural network*

The Structure2Vec component is based on the approach of [16] using the parameterization of [39]. In order to compute the embedding of the graph g, a p-dimensional vector μ_i is associated with each vertex v_i. The μ vectors are dynamically updated during the network training following a synchronous approach based on rounds. We refer with $\mu_i^{(t)}$ to the μ vector associated with vertex v_i at round t.

We aggregate vertex-specific μ vectors and features following the topology of the input graph g. After each step, the network generates a new μ vector for each vertex in the graph taking into account both the vertex features and graph-specific characteristics, see Figure 3.

In particular, the vertex vector μ_i is updated at each round as follows:

$$\mu_i^{(t+1)} = \mathcal{F}\Big(x_{v_i}, \sum_{u_j \in \mathcal{N}(v_i)} \mu_j^{(t)}\Big), \forall v_i \in V \tag{3}$$

The vertex μ vectors at round zero $\mu^{(0)}$ are randomly initialized and $\mathcal{F}$ is a nonlinear function:

$$\mathcal{F}\Big(x_{v_i}, \sum_{u_j \in \mathcal{N}(v_i)} \mu_j^{(t)}\Big) = \tanh\Big(W_1 x_{v_i} + \sigma\big(\sum_{u_j \in \mathcal{N}(v_i)} \mu_j^{(t)}\big)\Big) \tag{4}$$

where W_1 is a $d \times p$ matrix, $tanh$ indicates the hyperbolic tangent function and σ is a nonlinear function:

$$\sigma(y) = P_1 \times \text{ReLU}(P_2 \times ...\text{ReLU}(P_\ell \times y)) \tag{5}$$

The function $\sigma(y)$ is an ℓ layers fully-connected neural network, parametrized by ℓ matrices $P_i (i = 1, ..., \ell)$ of dimension $p \times p$. ReLU indicates the rectified linear unit function, i.e., $\text{ReLU}(x) = max\{0, x\}$.

The final graph embedding $\vec{g}$ is obtained by aggregating together the vertex μ vectors after T rounds, as follows:

$$\vec{g} = W_2 \sum_{v_i \in V} \mu_i^{(T)} \quad (6)$$

where W_2 is another $p \times p$ matrix used to transform the final graph embedding vector.

V. Evaluation

We conducted an experimental study on datasets extracted from collections of real binary code. Before delving in the evaluation of Task 1 - binary similarity (in Section V-B) and Task 2 - compiler provenance (in Section V-C), we discuss implementation details common to both tasks.

A. Implementation details

We developed a prototype implementation of the graph embedding neural network and of all the different vertex feature extraction solutions described in IV using Python and the Tensorflow [1] framework[6]. For static analysis of binaries we used the angr framework [36]. We trained the network using a batch size of 250, learning rate 0.001, *Adam* optimizer, feature vector size $|x_i| = 100$, function embeddings of dimension $p = 64$ (this is the same dimension of μ vectors), number of rounds $T = 2$, and a number of layers in Structure2Vec $\ell = 2$. These values have been chosen with an exhaustive grid search over the hyperparameters space. Tensorflow requires training batches of uniform dimension, therefore we manipulate each vertex v_i to contain the same number of instructions; i.e. we fix the length of I_{v_i} to 150, this is done by either padding with a special instruction or by truncation. Padding vectors contain all zeros. We fix the maximum number of vertices in each CFG to 150, removing CFGs larger than this threshold. In our dataset less than 4% of the graphs were above such threshold. Finally, the RNN used in i2v_RNN is a multi-layer network with 2 layers and GRU cell.

1) i2v details; datasets and training: For i2v we used the word2vec skip-gram implementation of [37]. The model parameters are: embedding size 100, window size 8, and word frequency 8.

Assembly source codes corpus to train i2v — We decided to capture the semantics and syntactic of the two architectures by building two different models, one for each instruction set. For each model we built two training corpora, one for AMD64 model and one for ARM model, by collecting the assembly code of a large number of functions. We built the corpora by disassembling several unix executables and libraries using IDA PRO[7], the libraries and the executables have been randomly sampled from repositories of linux distributions. We avoided multiple inclusion of functions by using a duplicate detection mechanism; we tested the uniqueness of a function computing an hash of all instructions, where instructions operands containing immediate and memory locations are replaced with a special symbol. From 2.52 GBs of AMD64 binaries we obtained the assembly of 547K unique functions. From 3.05 GBs of ARM binaries we obtained the assembly of 752K unique functions. Overall the AMD64 corpus contains 86 millions of assembly lines while the ARM corpus contains 104 millions of assembly lines.

To validate the benefits of an instruction embedding model we tested also what happens with random instruction embeddings. In particular, we associate a random vector to each instruction appearing more than 8 times in the training documents described above. All instructions appearing less than 8 times are mapped into the same random vector.

B. Task 1: Binary Similarity

Learning function embeddings: the siamese architecture

In this task we train the architecture of Section IV to generate CFG embeddings that preserve the similarity of the original CFGs. To do so we use, as in [39], a pairwise approach called *siamese network* [9]. This approach uses two identical graph embedding networks (i.e., the two networks share all the parameters) and join them with a similarity score. In this way the final output of the siamese architecture will represent the similarity score between the two input graphs. Technically speaking, from a pair of input graphs $< g_1, g_2 >$, first we obtained two vectors $< \vec{g_1}, \vec{g_2} >$ using the same graph embedding network, and then, these vectors are compared using cosine similarity:

$$\text{similarity}(\vec{g_1}, \vec{g_2}) = \frac{\sum_{i=1}^{p} \left(\vec{g_1}[i] \cdot \vec{g_2}[i] \right)}{\sqrt{\sum_{i=1}^{p} \vec{g_1}[i]} \cdot \sqrt{\sum_{i=1}^{p} \vec{g_2}[i]}} \quad (7)$$

where $\vec{g}[i]$ indicates the i-th component of the vector $\vec{g}$.

To train the network we require in input a set of K CFGs pairs, $< \vec{g_1}, \vec{g_2} >$, with ground truth labels $y_i \in \{+1, -1\}$, where $y_i = +1$ indicates that the two input graphs are similar and $y_i = -1$ otherwise. The training of the siamese network is done by minimizing the least squares objective function:

$$J = \sum_{i=1}^{K} \Big(\text{similarity}(\vec{g_1}, \vec{g_2}) - y_i \Big)^2 \quad (8)$$

Dataset, Test methodology, and, performance measure

Dataset — To align our experimental evaluation with state-of-the-art studies we built the OpenSSL Dataset in the same way as the one used in [39]. In particular, the dataset consists of a set of 95535 graphs generated from all the binaries included in two versions of Openssl (v1_0_1f - v1_0_1u) that have been compiled for x86 and ARM using gcc-5.4 with 4 optimization levels (i.e., -O[0-3]). The resulting binaries have been disassembled using angr[8] [36] discarding all the functions that angr was not able to disassemble.

[6]The source code of our prototype is available here: https://github.com/lucamassarelli/Unsupervised-Features-Learning-For-Binary-Similarity

[7]We used IDA PRO because of its performance in disassembling executables. However, the prototype we will publicly release is compatible with open source alternatives like angr and Radare2.

[8]angr is a framework for static and symbolic analysis of binaries

Index	Vertex Feature Extractor	Instruction Representation	Val Auc	Test Auc
1	MFE	-	*94.6%*	*95.0%*
2	i2v_attention	i2v	**96.1%**	**95.6%**
3		Random Embedding	88.7%	88.1%
4	i2v_RNN	i2v	94.5%	93.3%
5		Random Embedding	93.7%	93.2%

TABLE I. **Task 1 - Binary Similarity:** TEST OVER OPENSSL DATASET. IN ITALIC WE REPORT THE RESULTS OBTAINED REPRODUCING [39]. IN BOLD WE REPORT OUR BEST RESULTS.

Test methodology — We designed the methodology used in our tests following the one in [39]. More precisely, we generate our training and test pairs as reported in [39]. In order to train and test our system, we create a certain number of pairs using the OpenSSL Dataset. The pairs can be of two kinds: similar pairs, obtained pairing together two CGFs originated by the same source code, and dissimilar pairs, obtained by randomly pairing CFGs that do not derive from the same source code. In particular, for each CFG in our dataset we create two pairs, a similar pair, associated with training label $+1$ and a dissimilar pair, training label -1, obtaining a total number of pairs K that is twice the total number of CFGs. We split these pairs in three sets: train, validation, and test. As in [39] pairs are partitioned preventing that two similar CFGs are in different sets (this is done to avoid that the network sees during the training phase graphs similar to the ones on which it will be later validated or tested). The split is 80%-10%-10%. We train our models for 50 epochs (an epoch represents a complete pass over the whole training set) and we compute performance metrics on the validation set for all the epochs. Then, we use the model hyper-parameters that led to the best performance on the validation set to compute a final performance score on the test set. In each epoch we regenerate the training pairs, that is we create new similar and dissimilar pairs using the graphs contained in the training split. We pre-compute the pairs used in each epoch, in such a way that each method is tested on the same data. Note that, we do not regenerate the validation and test pairs.

We used this static train/validation/test split in our first set of experiments to understand which vertex feature extraction model performs the best; then we performed an additional set of experiments comparing the best performing solution with the baseline approach using 5-folds cross validation. In the 5-folds cross validation we partitions the dataset in 5 sets; for all possible set union of 4 partitions we train the classifiers on such union and then we test it on the remaining partition.

Performance Measure — We test our system using the standard *Receiver Operating Characteristic* (ROC) curve [25]. Specifically, we use the area under the ROC curve, or AUC (Area Under Curve), as evaluation metric.

Results

We present the results of two sets of experiments where we compare the considered vertex features extraction solutions on the OpenSSL Dataset.

Table I shows the results of our first set of experiments, conducted on the fixed train/test/validation split of the OpenSSL Dataset. The entries in the last two columns shows the best AUC value obtained on the validation set and the corresponding AUC value for the test set. First note that the best performing vertex feature extraction solution is i2v_attention with pre-trained i2v embeddings (i.e., index 2 - 95.6% AUC on test set). This highlights the benefits of adopting an unsupervised approach to feature learning with respect to a manual feature selection approach. Moreover, all solutions benefit from the pre-trained i2v models. In fact, pre-trained i2v vectors led to a significant boost in performance up to 7% with respect to randomly chosen instruction vectors. This confirms that a distributed representation of assembly instructions is indeed beneficial.

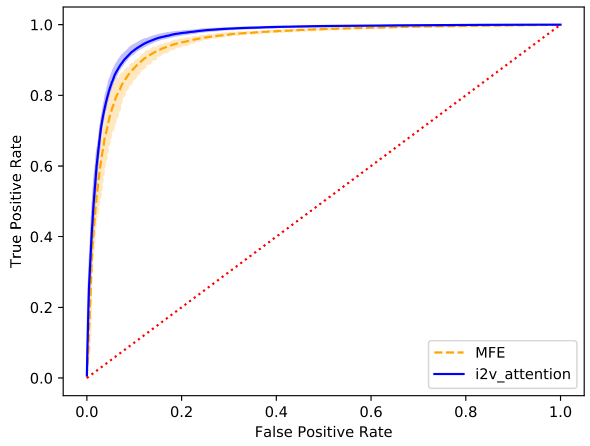


Fig. 4. ROC curves for the comparison between MFE and i2v_attention with static pre-trained i2v embeddings, using 5-fold cross validation. The lines represent the ROC curves obtained by averaging the results of the five runs; the dashed line is the average for MFE, the continuous line the average for i2v_attention. For both MFE and i2v_attention we color the area between the ROC curves with minimum AUC and the maximum AUC. The average AUC of i2v_attention is 96.4%, the one of MFE 94.8%.

The i2v_RNN solution, despite its greater theoretical representation power, performs worse than i2v_attention. This is probably due to the large number of parameters that the network has to train; the considered dataset might contain not enough training points to accurately train all of the parameters or simply the network is over-complex for this specific task.

As discussed in Section V-C, we performed a 5-fold cross validation between the two best performing models, MFE and i2v_attention.

Figure 4 shows the average ROC curves of the five runs. The MFE results are reported with an orange dashed line

Index	Vertex Feature Extractor	Instruction Representation	Val Accuracy	Test Accuracy
1	MFE	-	80.3%	81.2%
2	i2v_attention	i2v	93.1%	93.9%
3	i2v_RNN	i2v	**95.7%**	**95.9%**

TABLE II. **Task 2 - Compiler Provenance:** TEST OVER RESTRICTED COMPILER DATASET. IN BOLD WE REPORT THE BEST RESULTS.

Family	Precision	Recall	F1-Score	Support
clang	98%	97%	98%	63382
gcc	98%	99%	99%	119595
icc	98%	98%	98%	45015
Weighted Total	98%	98%	98%	227992

TABLE III. **Task 2 - Compiler Provenance:** RESULTS OF I2V_RNN OVERCOMPILERS DATASET. WE REPORT PRECISION, RECALL AND F1 SCORE FOR EACH COMPILER FAMILY IN THE DATASET. THE OVERALL ACCURACY IS 98.2%

while we used a continuous blue line for the i2v_attention results. For both solutions we additional highlighted the area between the ROC curves with minimum AUC and maximum AUC in the five runs. The better prediction performance of the i2v_attention solution are clearly visible; the average AUC obtained by MFE is 94.8% with a standard deviation of $0,6$ over the five runs, while the average AUC of i2v_attention is 96.4% with a standard deviation of $0,2$. Overall we observed an improvement of almost 2% of i2v_attention with respect to MFE. This clearly confirms the benefit of learning vertex features with respect to manually engineering them. Note also that the standard deviation of the AUC values over the five runs is smaller for i2v_attention than for MFE. As a final remark on the comparison, [39] states that removing the betweenness centrality from the features used by MFE slightly improves the performance of MFE. We tried to confirm this but we found no improvement.

C. Task 2: Compiler Provenance

In the compiler provenance task our purpose is to reconstruct the compiler family that has generated a certain binary function. In our tests we use 3 widely known compiler families: clang, gcc and icc.

Classifying graph embedding vectors

For this task we train the architecture of Section IV end-to-end with a feed-forward two layer neural networks. Specifically, starting from the graph embedding vector $\vec{g}$ we obtain a vector of classification probabilities as follows:

$$p = \text{softmax}(W_{out} \cdot \text{ReLU}(W_{hidden} \cdot \vec{g}))$$

The classifier is trained by minimizing the standard cross-entropy loss. Note that in this case there is no need for a siamese architecture.

Dataset, Test methodology, and, performance measure

Dataset — We built two datasets for this task: a Restricted Compiler Dataset that is used as screening dataset to select the best performing model, and a Compilers Dataset that is used to evaluate the best performing model.The details of the datasets are:

- Restricted Compiler Dataset. We built the dataset compiling different open-source projects: binutils-2.30, ccv-0.7, coreutils-8.29, curl-7.61.0. gsl-2.5, libhttpd-2.0, openmpi-3.1.1, openssl-1.1.1, valgrind-3.13.0. Each project has been compiled for AMD64 with three compilers: gcc-3.4 and gcc-5.0 and clang-3.9 and all 4 optimizations flags. After this step we disassembled the binaries using radare2. The number of binary functions inside this dataset is 452598.
- Compilers Dataset. We built the dataset using AMD64 binaries. Specifically, we compiled the following projects: binutils-2.30, ccv7.0, coreutils-8.29, curl-7.61.0, ffmpeg-4.0.2, gdb-8.2, gsl-2.5, libhttpd-2.0, openssl-1.1.1-pre8, postgresql-10.4, valgrind-3.13.0, using the following compilers: gcc-3.4, gcc-4.7, gcc-4.8, gcc-4.9, gcc-5.0, clang-3.8, clang-3.9, clang-4.0, clang-5.0, icc-17 and icc-19 with all 4 optimization flags. We disassembled the binaries using radare2 and we removed duplicated or really similar functions by considering two functions identical if they have the same instructions without considering the specific values of immediate operands and memory location accesses. The number of binary functions in the dataset is 1587648.

Test methodology and classifier parameters — For each function we used as ground-truth the compiler family that generated the function. We split each dataset in train set, validation set and test set (the split is 70%-15%-15%). We trained our models for 50 epochs (an epoch represents a complete pass over the whole training set; for each epoch we shuffle the training samples) and we computed the performance metric on the validation set for all the epochs.

Then we took the model that has the best accuracy on the validation split, and we computed the final performance score on the test split. The hidden layer of the feed forward classifier has size 3000. For i2v_RNN we use cells of depth 1.

Performance Measure — We tested our multi-class classifier using the standard measures: precision, recall, and f1-score.

For the selection of the best model we considered the overall accuracy, defined as the fraction of predictions our model got right.

Results

The result of our screening test are in Table II. It is possible to see that all the unsupervised extraction techniques (i.e., i2v_attention and i2v_RNN) outperform MFE by a large margin, achieving a more than 10% boost in performance. Between the two, the i2v_RNN performs better than i2v_attention, achieving an accuracy of $95,9\%$ vs $93,9\%$.

We believe that this big difference in performance could be attributed to the inability of the vertex feature vector defined in [39] to capture compiler specific signals. Such vertex features are, in fact, hand crafted for the binary similarity problem (e.g., number of strings and constants in the block) and fail to generalize to other use cases. Our method, instead, is able to learn directly from the data what are the latent features that help the most in the specific task considered, and therefore it can automatically adapts to the specifics of the problem.

In Table III we additionally report the performances on Compilers Dataset, where the i2v_RNN reaches a test accuracy of $98,2\%$. Note that this result is not far from the one reported in the literature for the state-of-the-art [33], where an accuracy of 98% is achieved for the task of family identification for binary functions. Unfortunately, we were not able to perform a direct comparison with [33]: their source code is not available; additionally, the reconstruction of their specific dataset is not possible (versions of software included in the dataset are not specified, and they used compilers that are not anymore commercially available; e.g., Visual Studio 2003).

Finally, we performed a test on Compilers Dataset using random instructions embeddings instead of pre-trained ones (i.e., i2v). In this case i2v_RNN reaches a test accuracy of $95,2\%$ this confirms the benefits of pre-training the representations for assembly instructions, that in this case leads to a 3% improvement.

VI. Discussion

Our unsupervised approach to vertices feature extraction in the CFG achieves better performance than the original approach based on manual feature engineering proposed in [39] on both considered tasks.

We performed a further comparison with the SAFE architecture proposed by us in a previous paper [29]. SAFE does not use the CFG but a self-attentive recurrent neural network that parses all instructions according to their addresses. Interestingly, despite SAFE not using any control flow information, it shows comparable or better performance than our methods, (AUC of 99% on the OpenSSL Dataset in task 1, and, accuracy of 97.5% on the Compilers Dataset in task 2). This raises interesting questions on the usefulness of using graph embedding networks on CFGs.

We believe that a binary function has some peculiar characteristics that cannot be captured by representing it as a simple graph. One such characteristic is, for instance, the fact that the CFG is nothing but a representation of all possible execution paths obtained by varying the input of the binary function. Just one of this path will be followed during the execution of the function when its inputs and its context are defined. We believe that by exploiting this consideration better binary function representation can be obtained. In this regards is worth mentioning the attempt done by the authors of [21], where they explicitly consider traces generated by random walks on a CFG in order to compute the final embedding vector, but focusing just on the AMD64 architecture. We believe that an investigation of other graph embedding techniques is needed to confirm, or confute, our hypothesis on the negligible usefulness of applying a graph embedding network to CFGs.

VII. Conclusions and Open Challenges

In this work we show that associating features to the vertices of the control flow graph (CFG) in an unsupervised fashion, taking inspiration from natural language embedding techniques, is beneficial for the binary similarity task, as well as for the task of compiler provenance. Our experimental results show that for the first task our model reaches an AUC of 96.4% vs 94.8% achieved annotating the CFG with manually engineered features. For the second task we show an even bigger boost in performance, from an accuracy of 81.2% (manual feature engineering) to 95.9% (unsupervised approach). These results clearly highlight the benefit of a vertex representation for the CFG that is automatically learn from the data.

Is interesting to notice that our experimental evaluation additionally show that a recent approach that does not make use of the concept of control flow graph, but consider the binary as a flat sequence of assembly instructions, achieve performances that are comparable (in some cases even slightly better) to those achieved by graph embedding networks techniques. In the light of these results we believe that it would be interesting to study ad-hoc techniques for embedding instructions flows.

Open Challenges — During the writing of this manuscript we found the following worthwhile challenges:

- **Analogy Dataset.** A common approach to evaluate the performance of distributed representation models for words is to use a dataset of analogies [30]. When we consider embeddings of assembly instructions (i.e., our i2v encoding) such standardized dataset does not exist. Creating a dataset of analogies would permit to compare different distributed representation models and to pick the one that is able to better capture the semantic relationship between instructions.
- **Function Embedding Benchmark.** At the best of our knowledge, the current literature in binary analysis is developing and testing embedding techniques to solve a specific task at hand [21], [39], [40]. However, proper comparison of functions embedding techniques needs a general, and standardized, evaluation benchmark. This would test the intrinsic quality of the embeddings and their usefulness for new, possibly unknown, downstream tasks.
  Such benchmarks are already a standard in the NLP community, as example see the GLUE benchmark [38].

Acknowledgment

This work has been partially founded by a grant by the Italian Presidency of the Council of Ministers, by CINI with the FilieraSicura project, by the PANACEA Horizon 2020 research and innovation programme under the Grant Agreement no 826293 and by the University of Rome "La Sapienza" with the Calypso project. The authors would also like to thank Google for providing cloud computing resources through the Education Program and NVIDIA Corporation for the donation of a GPGPU.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*, 2016, pp. 265–283.

[2] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. A. Sutton, "A survey of machine learning for big code and naturalness," *CoRR*, vol. abs/1709.06182, 2017. [Online]. Available: http://arxiv.org/abs/1709.06182

[3] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, vol. 1711.00740, 2017.

[4] F. E. Allen, "Control flow analysis," *SIGPLAN Notice*, vol. 5, no. 7, pp. 1–19, 1970.

[5] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *CoRR*, vol. 1808.01400, 2018.

[6] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[7] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *Proceedings of 23rd USENIX Security Symposium, (USENIX Security)*. USENIX, 2014, pp. 845–860.

[8] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[9] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," in *Proceedings of the 6th International Conference on Neural Information Processing Systems, (NIPS)*, 1994, pp. 737–744.

[10] N. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," *CoRR*, vol. abs/1710.06159, 2017. [Online]. Available: http://arxiv.org/abs/1710.06159

[11] S. Chakraborty, M. Allamanis, and B. Ray, "Tree2tree neural translation model for learning source code changes," *CoRR*, vol. 1810.00314, 2018.

[12] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *CoRR*, vol. 1802.03691, 2018.

[13] Y. Chen, Z. Shi, H. Li, W. Zhao, Y. Liu, and Y. Qiao, "Himalia: Recovering compiler optimization levels from binaries by deep learning," in *Proceedings of SAI Intelligent Systems Conference*. Springer, 2018, pp. 35–47.

[14] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proceedings of 26th USENIX Security Symposium, (USENIX Security)*, 2017, pp. 99–116.

[15] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[16] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proceedings of the 33rd International Conference on Machine Learning, (ICML)*, 2016, pp. 2702–2711.

[17] H. Dam, T. Pham, S. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C. Kim, "A deep tree-based model for software defect prediction," *CoRR*, vol. abs/1802.00921, 2018. [Online]. Available: http://arxiv.org/abs/1802.00921

[18] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*, 2016, pp. 266–280.

[19] ——, "Similarity of binaries through re-optimization," in *ACM SIGPLAN Notices*, vol. 52, no. 6, 2017, pp. 79–94.

[20] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the 14th international conference on Tools and Algorithms for the Construction and Analysis of Systems, (TACAS)*, 2008, pp. 337–340.

[21] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *(to Appear) Proceedings of 40th Symposium on Security and Privacy, (SP)*, 2019.

[22] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.

[23] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security, (CCS)*. ACM, 2016, pp. 480–491.

[24] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "Lemna: Explaining deep learning based security applications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 364–379.

[25] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl, "Evaluating collaborative filtering recommender systems," *ACM Transactions on Information Systems*, vol. 22, no. 1, pp. 5–53, 2004.

[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[27] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proceedings of the 10th Working Conference on Mining Software Repositories, (MSR)*, 2013, pp. 329–338.

[28] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31th International Conference on Machine Learning, (ICML)*, 2014, pp. 1188–1196.

[29] L. Massarelli, G. A. D. Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Safe: Self-attentive function embeddings for binary similarity," 2018.

[30] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems, (NIPS)*, 2013, pp. 3111–3119.

[31] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the 34th IEEE Symposium on Security and Privacy, (SP)*, 2015, pp. 709–724.

[32] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi, "Bincomp: A stratified approach to compiler provenance attribution," *Digital Investigation*, vol. 14, pp. S146–S155, 2015.

[33] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 100–110.

[34] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2010, pp. 21–28.

[35] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks." in *Proceedings of 24th USENIX Security Symposium, (USENIX Security)*. USENIX, 2015, pp. 611–626.

[36] Y. Shoshitaishvili, C. Kruegel, G. Vigna, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy, (SP)*, 2016, pp. 138–157.

[37] TensorFlow Authors, "Word2vec skip-gram implementation in tensorflow," in *https://www.tensorflow.org/tutorials/representation/word2vec*, last accessed 07/2018.

[38] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.

[39] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security, (CCS)*, 2017, pp. 363–376.

[40] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *arXiv preprint arXiv:1808.04706*, 2018.

VIII. Appendix

A. An Additional Vertex Features Extractor Technique, Sampling

Apart from the vertex features extractions discussed in Section IV-A we also tested a Sampling strategy that leverages the symbolic execution of codes[9].

- Sampling: We represent each block as a set of multivariable functions, and then compute features as sampling of these functions over random inputs. This mechanism is not trainable. **Rationale:** this method is based on the assumption, common to [31], that sampling a sequence of instructions captures a semantic that cannot be captured by static analysis.

In our test Sampling performed really badly. We include its details in the appendix. We believe that is a worth mentioning negative result.

1) Sampling details: Given a vertex of the CFG, we partition it in sequences of instructions that update independent memory locations, these are the *strands* defined in [18]. We construct an extended version of the CFG that we call SCFG. The SCFG contains all strands of the CFG as vertices and its topology is built as follows: for each vertex v_i of the CFG we define a total order on the vertex's strands. We take all predecessors of vertex v_i in the CFG and in the SCFG we add an edge from each predecessor's last strand to the first strand in v_i. Finally, we create an oriented path connecting all strands in v_i according to their order.

To transform a strand in a set of functions we execute it symbolically, using the angr framework. Obtaining multivariable functions expressed as formulas for the Z3 solver [20]: Each function defines the value of a memory location that is wrote without being read after (output), as combination of a set of inputs (an input is a memory location that is read before any write on it and that concurs to the value of the output).

Finally, we compute a feature vector from each strand by sampling and averaging the outputs of the strand's functions over 100 randoms inputs.

Note that functions are in general non symmetric; This can create a problem since two semantically equivalent strands may have different ordering of their inputs. We address this problem by symmetrising the function:

Given a function $z : \mathbb{R}^n \to \mathbb{R}$ we define its symmetrisation as:

$$z'(\vec{b}) = \frac{1}{|\Pi(\vec{b})|} \sum_{\vec{b'} \in \Pi(\vec{b})} z(\vec{b'}) \tag{9}$$

where $\Pi(\vec{b})$ is the set of all the possible vectors obtained by permuting the components of vector $\vec{b}$. The above implies that we have to compute z on each permutation of its inputs. This is a costly operation, the subdivision in strands it is beneficial: subdividing a vertex in smaller units reduces the number of variables in each of the segments that are symbolically executed. Unfortunately, it is not enough. Therefore, we cap the number of inputs to 5 (this threshold includes almost all functions in our test dataset). More precisely, we order the inputs in an arbitrary way and we forces all inputs from position 6 on to value zero. Combining symmetrisation and the average of all functions outputs we obtain a feature array of size 100 for each vertex of the SCFG.

2) Sampling performances on binary similarity: The Sampling solution performs particularly poorly, achieving an AUC score on the test set of 0.710. The reason behind this poor performance is probably due to increased complexity of the SCFG graph (recall that SCFG represents the program flow among strands, and this leads to a graph that has more edges and a more intricate structure that the CF), and/or to the poor quality of the features obtained by sampling. However, a further investigation is needed to confirm that such sampling strategy is indeed a failure.

We did not test Sampling for the compiler provenance case, the rationale is that the execution output, in correct binaries, should be compiler invariant. Thus Sampling is an intrinsically bad feature to detect binary artefacts that do not change the execution semantic.

[9]We implement this approach using angr as symbolic execution engine