Function Representations for Binary Similarity

Luca Massarelli†, Giuseppe Antonio Di Luna†, Fabio Petroni∗,
Leonardo Querzoni†, Roberto Baldoni†
†: University of Rome Sapienza. {massarelli, diluna, querzoni, baldoni}@diag.uniroma1.it.
∗: Facebook AI Research, petronif@acm.org.

***Abstract*—The binary similarity problem consists in determining if two functions are similar considering only their compiled form. Advanced techniques for binary similarity recently gained momentum as they can be applied in several fields, such as copyright disputes, malware analysis, vulnerability detection, etc. In this paper we describe SAFE, a novel architecture for function representation based on a self-attentive neural network. SAFE works directly on disassembled binary functions, does not require manual feature extraction, is computationally more efficient than existing solutions, and is more general as it works on stripped binaries and on multiple architectures. Results from our experimental evaluation show how SAFE provides a performance improvement with respect to previous solutions. Furthermore, we show how SAFE can be used in widely different use cases, thus providing a general solution for several application scenarios.**

***Index Terms*—Binary Analysis, Binary Similarity, Deep Learning, Malware**

I. Introduction

In the last years there has been an exponential increase in the creation of new content. Software was no exception to this trend. As an example, the number of apps available on the Google Play Store increased from 30K in 2010 to 3 millions in 2018[1]. This increase directly leads to more vulnerabilities as reported by CVE[2] that witnessed a 120% growth in the number of discovered vulnerabilities from 2016 to 2017. At the same time complex software find application in new devices: the *Internet of Things* has multiplied the number of architectures on which the same program has to run and COTS software components are increasingly integrated into closed-source products.

This multidimensional increase in quantity, complexity and diffusion of software makes the resulting infrastructures difficult to manage and control, as part of their internals are often inaccessible for inspection to their administrators. As a consequence, system integrators are looking forward to novel solutions that take into account such issues and provide functionalities to automatically analyze software artifacts in their compiled form (binary code). One prototypical problem in this regard, is the one of *binary similarity* [2], [3], [4], where the goal is to find similar functions in compiled code fragments. This problem has been recently subject to a lot of attention [5], [6], [7] due to its centrality in several tasks, such as the discovery of known vulnerabilities in large collection of software, dispute on copyright matters, analysis and detection of malicious software [8], etc.

In this paper, coherently with [9], [10], we focus on a specific version of the binary similarity problem in which we define two binary functions to be similar if they are compiled from the same source code. As already pointed out in [10], this assumption does not make the problem trivial.

Inspired by [9] we look for solutions that solve the binary similarity problem using *embeddings*, i.e. vectors of numbers that embed binary functions characteristics while preserving a similarity metric. Function embeddings can be pre-computed, and checking their similarity is relatively cheap and fast (we consider the scalar product of two constants size vectors as a constant time operation), thus providing an important performance boost for several use cases. Furthermore, as we will show, embeddings can be used as input to other machine learning algorithms, that can in turn cluster functions, classify them, etc.

Current solutions that adopt this approach, come with several shortcomings. Firstly, they [10] use manually selected features to calculate the embeddings, introducing potential bias in the resulting vectors. Such bias stems from the possibility of overlooking important features (that don't get selected), or including features that are expensive to process while not providing noticeable performance improvements. Secondly, they [11] assume that call symbols to dynamically linked libraries are available in binary functions (such as libc, msvc, etc.), while this is not true for statically linked and stripped binaries, statically linked libraries or in partial binary fragments (e.g. extracted during volatile memory forensic analyses). Finally, they usually work only on specific CPU architectures [11].

Considering these shortcomings, in this paper we describe SAFE: Self-Attentive Function Embeddings, a solution we designed to overcome all of them. In particular, we considered two specific goals: (i) design a solution to quickly generate embeddings for several hundreds of binaries and (ii) that could be applicable to a vast majority of cases, i.e. able to work with stripped binaries with statically linked libraries, and on multiple architectures (in particular we consider AMD64 and ARM as target platforms for our study). The core of SAFE is based on recent advancements in the area of natural language processing. Specifically, we designed SAFE on a Self-Attentive Neural Network recently proposed in [12].

We extensively tested SAFE showing that it provides better performance than previous state-of-the-art systems with similar requirements. Specifically, we compare it with the recent Gemini [10], showing a performance improvement on several

A preliminary version of this paper has been published in [1]

[1]www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[2]www.cvedetails.com/browse-by-date.php

metrics that ranges from 6% to 29% depending on the task at hand. Furthermore, we also tested SAFE in several other tasks related to several diverse application scenarios to test its general applicability.

We investigate the possibility of semantically classifying (i.e. identifying the general semantic behavior of) binary functions by clustering similar embeddings. To the best of our knowledge, we are the first to investigate the feasibility of this task through machine learning tools, and to perform a quantitative analysis on this subject. The results are encouraging showing a 95% of classification accuracy for 4 different broad classes of algorithms (namely *Encryption*, *Sorting*, *Mathematical* and *String Manipulation* functions). We applied our semantic classifier to known malwares, and we were able to accurately recognize with it functions implementing encryption algorithms. Moreover, we used a recent dataset [13] of malwares from APT groups, and we used SAFE embedding to build a classifier associating malwares and APT with an f1 score of 90%.

Finally, we decided to test SAFE on a task that is completely unrelated to the binary similarity one to assess the ability to abstract from the specific features needed to solve a particular class of problems. For this purpose, we decided to study the compiler provenance task (i.e., to understand which compiler produced a given binary function [14]). This problem can be seen as the dual of binary similarity: in one task the network has to focus on the differences introduced by compilers, while in the other it has to do the opposite. Our tests show an accuracy of 97.4% for compiler family classification, reaching performances that are comparable with the state-of-the-art [14].

A. Contributions

The main contributions of our work are:

- we describe SAFE, a general architecture for calculating binary function embeddings starting from disassembled binaries;
- we publicly release SAFE source code and the datasets used for its training and evaluation[3];
- we apply SAFE to the problem of identifying vulnerable functions in binary code, a common application task for binary similarity solutions; also in this task SAFE provides better performance than state-of-the-art solutions.
- we show that embeddings produced by SAFE can be used to automatically classify binary functions in semantic classes. On a dataset of 15K functions, we can recognize whether a function implements an encryption algorithm, a sorting algorithm, generic math operations, or a string manipulation, with an accuracy of 95%.
- we use SAFE to classify and cluster malwares accordingly to the APT group that developed them, reaching an f1-score of 90%.
- we provide some insights on the embedding vector space through a qualitative analysis.

[3]The source code of our prototype and the datasets are publicly available at the following address: https://github.com/gadiluna/SAFE

This paper is an extended version of [1] where we originally introduced SAFE. The main contributions of this paper are the investigation of the compiler provenance problem through the SAFE model (discussed in Section VII) and the application of SAFE function embeddings to the problem of APTs (Advanced Persistent Threats) malware classification (discussed in Section VI-E). These two new contributions enlarge the scope of our research on SAFE, proving its general applicability in several contexts where the analysis of binary code is relevant.

The remainder of this paper is organized as follows. Section II discusses related work, followed by Section III where we define the problem and report an overview of the solution we tested. In Section IV we describe in detail SAFE, and in Section V we provide implementation details and information on the training. In Section VI we describe the experiments we performed and report their results. Finally, in Section VIII we discuss the speed of SAFE.

II. Related Work

The existing literature in the field of binary similarity can be broadly divided in two large families: works that use and works that do not use function embeddings. Within these groups some works proposed cross-platform solutions, while other focus on single-platform proposals. In this related work section we focus on the papers that are most similar to our approach. The interested reader can find a detailed and recent survey in [15].

a) Works that do not use embeddings: A family of works is based on matching algorithms for function CFGs. In Bindiff [2] matching among vertices is based on the syntax of code, and it is known to perform poorly across different compilers (see [5]). Pewny et al. [16] proposed a solution where each vertex of a CFG is represented with an expression tree; similarity among vertices is computed by using the edit distance between the corresponding expression trees. Other works use different solutions that do not rely on graph matching. David and Yahav [17] introduced the concept of *tracelets*, while a related concept, namely *strands*, was used by David et al. [5]. In the latter paper functions are divided in pieces of independent code, the strands. The matching between functions is based on how many statistically significant strands are similar. Intuitively, a strand is significant if it is not statistically common. Strand similarity is computed using an SMT-solver to assess semantic similarity. A similar approach is followed by *Binjuice* [18] in which each block of a CFG is interpreted and substituted with an abstract representation of its semantic. The use of the CFG topology for obtaining a measure of similarity has been also used in the field of malware analysis, see [8].

All these solutions were designed around matching procedures that work *pair-to-pair*, and they cannot be adapted to pre-compute a constant size signature of a binary function on which similarity can be assessed.

Egele et al. in [7] proposed to derive a signature from features collected during multiple independent executions of the target function. Specifically, each function is executed multiple times in a random environment. During the executions some features are collected and then used to match similar functions. This solution can be used to compute a signature for each function. However, it needs to execute a function multiple times, that is both time-consuming and difficult to perform in the cross-platform scenario. Furthermore, it is not clear if the features identified in [7] are useful for cross-platform comparison. Finally, Khoo et al. [3] proposed a matching approach based on *n-grams* computed on instruction mnemonics and *graphlets*. Even if this strategy does produce a signature, it cannot be immediately extended to cross-platform similarity.

Pewny et al. [19] proposed a graph-based methodology that uses the function's CFG: the binary code is transformed in an intermediate representation where the semantic of each CFG vertex is computed by using a sampling of code executions with random inputs. Feng et al. [20] proposed a solution where each function is expressed as a set of conditional formulas; then it uses integer programming to compute the maximum matching between formulas. Both solutions, [19] and [20], are cross-platform but only allow *pair-to-pair* checks.

David et al. [6] propose to transform binary code to an intermediate representation that is then partitioned in *strands* (slices of independent code). Strands with the same semantics will have similar representations, and this similarity is then transferred to functions. This approach can generate variable-sized signatures for functions.

b) Works based on embeddings: Ding et al. recently proposed in [11] a function embedding solution called *Asm2Vec* that is based on the PV-DM model [21] for natural language processing.

Operatively, Asm2Vec computes the CFG of a function, and then it performs a series of random walks on top of it. Essentially, each random walk is the sequence of instructions that are executed on a certain execution path of the CFG.

This set of random walk is interpreted as a paragraph of text on which PV-DM is run. PV-DM transforms the paragraph into an embedding vector following a philosophy similar to word2vec [22]. It computes the embedding paragraph vector that maximizes the probability of predicting an assembly token given its context and the paragraph vector itself. We stress that PV-DM is an unsupervised technique, which means that the embedding architecture provided by Asm2Vec cannot be trained specifically for a certain downstream task (such as compiler provenance). Asm2Vec outperforms several state-of-the-art solutions in the field of binary similarity, but still has some important limitations: firstly it requires *libc* call symbols to be present in the binary code as tokens to produce the embedding of a function and, secondly, it can only generate single-platform embeddings (it cannot be used for cross-architecture similarity). Moreover, it has the performance overhead of performing random walks on graphs.

Feng et al. [9] introduced a solution that uses a clustering algorithm over a set of functions to obtain centroids for each cluster. Then, these centroids and a configurable feature encoding mechanism are used to associate an embedding with each function. This solution is not based on deep neural networks, but it has been the first to propose the concept of embedding in the field of binary similarity.

Xu et al. [10] proposed an architecture called *Gemini*, where function embeddings are computed using a deep neural network. Interestingly, [10] shows that Gemini outperforms [9] both in terms of accuracy and performance (measured as time required to train the model). In Gemini the CFG of a function is first transformed into an *annotated CFG*, a graph containing manually selected features, and then embedded into a vector using the graph embedding model of [23]. The manual features used by Gemini do not need call symbols. A more detailed comparison between our architecture and Gemini is reported in Section IV.

The recent, VulSeeker [24] builds on top of Gemini, and extends the annotated CFG to a *labeled Semantic Flow Graph*, that is a graph mixing the CFG with edges from the data flow graph. Apart from this difference, VulSeeker uses the exact same approach of Gemini (the same learning architecture and training methodology). We remark that in VulSeeker, as in Gemini, sequences of assembly instructions are abstracted away and substituted with manually selected features. On the performance side, VulSeeker shows some improvements on the results of Gemini, in the vulnerability search case and on the binary similarity test used also by Gemini.

In [25] we proposed a variation of Gemini where manual features are replaced with an unsupervised feature learning mechanism that provides a small advantage in terms of accuracy.

Finally, in InnerEye [26] the authors propose the use of a recurrent neural network based on LSTM (Long short-term memory) to solve a specific subtask of binary similarity that is the one of finding similar CFG blocks.

Operatively, each CFG block is interpreted as a linear sequence of assembly instructions, such instructions are given as input to a LSTM network. The last internal state of the LSTM is the embedding vector of the sequence. The training is done similarly to Gemini using a siamese architecture.

There are several main differences between [26] and our work. The first one is that they only design and evaluate their architectures on the task of finding similar blocks, that is their solution creates embeddings for the CFG blocks. It is not clear how these embeddings should and could be aggregated to find a representation of the entire function. Therefore, their solution is not suited for our task. Secondly, the LSTM architecture does not cope well with long sequences (see [12]), while our self-attentive architecture has been explicitly designed to address such problem (more details on this in Section IV).

Recently, a system named DeepBinDiff [27] has been proposed to solve the related problem of finding the differences between two binaries (note that these are complete binaries and not just binary functions). DeepBinDiff uses a complex approach based on deep neural networks and greedy graph matching. It first computes a program-wide inter-procedural control flow graph, it then uses such a graph to compute both the embeddings of each assembly token, and the embedding of each basic block. The philosophy to generate a block level embedding is similar to the one used by InnerEye. DeepBinDiff leverages these embedding and a greedy graph matching algorithm between the ICFG of the two programs to be tested. The deep learning procedure used by DeepBinDiff

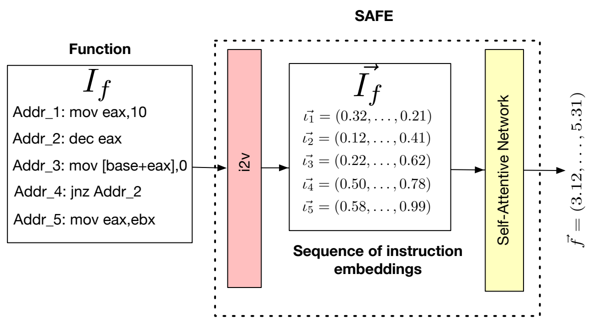

SAFE
Function
I_f
Addr_1: mov eax,10
Addr_2: dec eax
Addr_3: mov [base+eax],0
Addr_4: jnz Addr_2
Addr_5: mov eax,ebx
i2v
$\vec{I_f}$
$\vec{\iota_1} = (0.32, \ldots, 0.21)$
$\vec{\iota_2} = (0.12, \ldots, 0.41)$
$\vec{\iota_3} = (0.22, \ldots, 0.62)$
$\vec{\iota_4} = (0.50, \ldots, 0.78)$
$\vec{\iota_5} = (0.58, \ldots, 0.99)$
Sequence of instruction embeddings
Self-Attentive Network
$\vec{f} = (3.12, \ldots, 5.31)$

Fig. 1. Architecture of SAFE. The vertex feature extractor component refers to the Unsupervised Feature Learning case.

is unsupervised, as the one followed by Asm2Vec. We remark that DeepBinDiff uses information such as call symbols and strings, that would be stripped away or obfuscated in case of *unfriendly binaries* (such as a statically linked malware), and that DeepBinDiff is a single architecture solution.

III. Problem Definition and Solution Overview

We say that two binary functions f_1, f_2 are similar, $f_1 \sim f_2$, if they are the result of compiling the same original source code s with different compilers. Essentially, a compiler c is a deterministic transformation that maps a source code s to a corresponding binary function f^s. In this paper we consider as a compiler the specific software, e.g. gcc-5.4.0, together with the parameters that influence the compiling process, e.g. the optimization flags -O$[0, ..., 3]$.

We indicate with $I_{f_1} : (\iota_1, \iota_2, \iota_3, \ldots, \iota_m)$, the list of assembly instructions composing function f_1. Our aim is to represent f_1 as a vector in $\mathbb{R}^n$. This is achieved with an embedding model that maps I_{f_1} to an *embedding vector* $\vec{f_1} \in \mathbb{R}^n$, preserving structural similarity relations between binary functions.

A. SAFE Overview.

We use an embedding model structured in two phases; in the first phase the *Assembly Instructions Embedding* component, transforms a sequence of assembly instructions I_f in a sequence of vectors, in the second phase a *Self-Attentive Neural Network*, transforms a sequence of vectors in a single embedding vector. See Figure 1 for a schematic representation of the overall architecture of our embedding network.

*Assembly Instructions Embedding (*i2v*):* — In the first phase of our strategy we map each instruction $\iota \in I_f$ to a vector of real numbers $\vec{\iota}$, using the word2vec model [22]. Word2vec is an extremely popular feature learning technique in natural language processing. We use a large corpus of instructions to train our instruction embedding model (see Section V), we call our mapping instruction2vec (i2v). The final outcome of this step is a sequence of vectors $\vec{I_f}$.

Self-Attentive Network: — For our Self-Attentive Network we use the network recently proposed in [12]. In this network, a bi-directional recurrent neural network is fed with the sequence of assembly vectors. Intuitively, for each instruction vector $\vec{\iota_i}$ the RNN computes a summary vector taking into account the instruction itself and its context in I_f. The final embedding of $\vec{I_f}$ is a weighted sum of all summary vectors. The weights of such summation are computed by a two-layers fully-connected neural network.

We selected the Self-Attentive Network for two reasons. First, it shows state-of-the art performance on natural language processing tasks [12]. Secondly, it suffers less of the long-memory problem[4] of classic RNNs: in the Self-Attentive case, the RNN computes only a local summary of each instruction. Our research hypothesis is that it would behave well over the long sequences of instructions composing binary functions; and this hypothesis is indeed confirmed in our experiments (see Section VI).

IV. Details of the SAFE, Function Embedding Network

*Assembly Instructions Embedding (*i2v*):* — The first step of our solution consists in associating an embedding vector to each instruction ι contained in I_f. We achieve it by training the embedding model i2v using the skip-gram method [22]. The idea of skip-gram is to use the current instruction to predict the instructions around it. A similar approach has been used also in [28].

We train the i2v model using assembly instructions as tokens (i.e., a single token includes both the instruction mnemonic and the operands). Similarly to [29] we reduce our instruction vocabulary to a reasonable size. This is a standard practice in natural language processing to allow efficient computation of the softmax function over all the tokens [30]. In particular, we examine the operands and replace all 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) with the special symbol `IMM`. The interested reader can find in [29] an experimental evaluation of similar pre-processing techniques, that has shown the beneficial impact of such step. A similar normalisation step is also used by the recent [27].

As example refer to the instructions in Figure 1 instruction `mov EAX,10` becomes `mov EAX,10`; while `mov [ADDR+EAX],0` becomes `mov [EAX+MEM],0`, and `JNZ ADDR_2` becomes `JNZ MEM`. Note that in our architecture the network does not see the address associated with each instruction, i.e. it does not know which instruction is pointed by `ADDR_2`, therefore it is of no benefit to show explicitly the address.

As another example consider 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 pre-processing. We do not perform anti-aliasing steps to normalise registers aliases.

Self-Attentive Network: — We based our Self-Attentive Network on the one proposed by [12] (see Figure 2). We compute embedding $\vec{f}$ of a function f by using the sequence of instruction vectors $\vec{I_f} : (\vec{\iota_1}, \ldots, \vec{\iota_m})$. These vectors are fed

[4]Classic RNNs do not cope well with really long sequences.

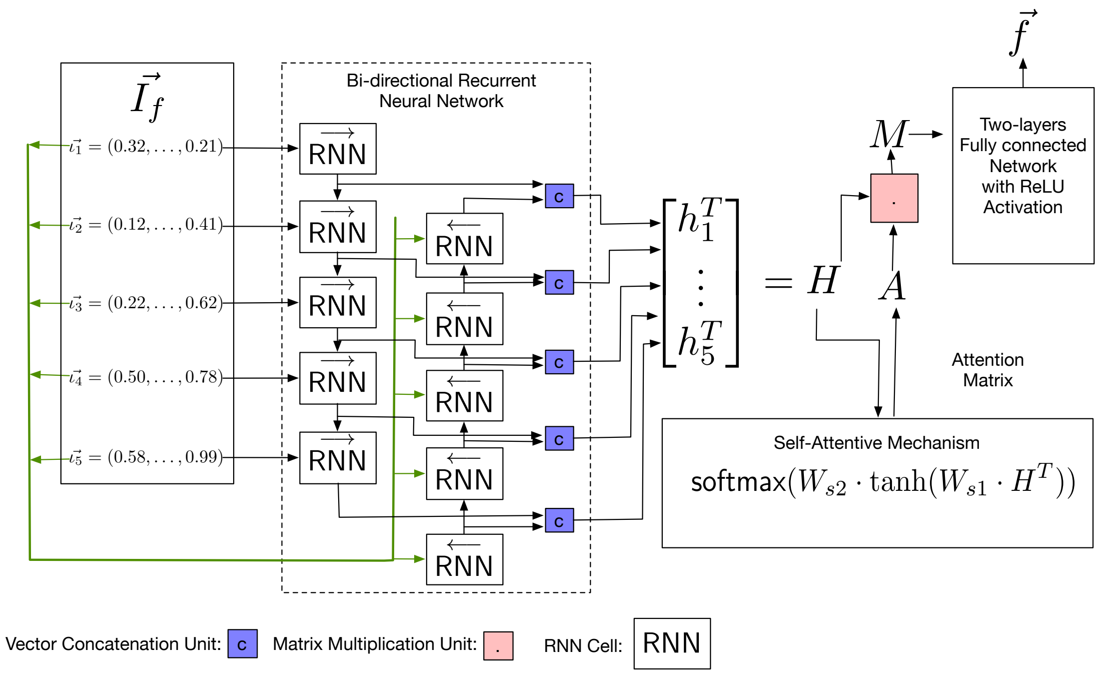

$\vec{I_f}$
$\vec{\iota_1} = (0.32, \ldots, 0.21)$
$\vec{\iota_2} = (0.12, \ldots, 0.41)$
$\vec{\iota_3} = (0.22, \ldots, 0.62)$
$\vec{\iota_4} = (0.50, \ldots, 0.78)$
$\vec{\iota_5} = (0.58, \ldots, 0.99)$
Bi-directional Recurrent Neural Network
RNN
c
h_1^T ⋮ h_5^T $= H$
A
M
Two-layers Fully connected Network with ReLU Activation
$\vec{f}$
Attention Matrix
Self-Attentive Mechanism
$\mathrm{softmax}(W_{s2} \cdot \tanh(W_{s1} \cdot H^T))$
Vector Concatenation Unit:
Matrix Multiplication Unit:
RNN Cell:

Fig. 2. Self-Attentive Network: detailed architecture.

into a bi-directional neural network, obtaining for each vector $\vec{\iota_i} \in \vec{I_f}$ a *summary* vector of size u:

$$h_i = \overrightarrow{\mathsf{RNN}}(\overrightarrow{h_{i-1}}, \iota_i) \oplus \overleftarrow{\mathsf{RNN}}(\overleftarrow{h_{i+1}}, \iota_i)$$

where $\oplus$ is the concatenation operand, $\overrightarrow{\mathsf{RNN}}$ (resp., $\overleftarrow{\mathsf{RNN}}$) is the forward (resp., backward) RNN cell, and $\overrightarrow{h_{i-1}}, \overleftarrow{h_{i+1}}$ are the forward and backward states of the RNN (we set $\overrightarrow{h_{-1}} = \overleftarrow{h_{n+1}} = 0$). The state of each RNN cell has size $\frac{u}{2}$.

From these summary vectors we obtain a $m \times u$ matrix H. Matrix H has as rows the summary vectors. An attention matrix A of size $r \times m$ is computed using a two-layers neural network: $A = \mathsf{softmax}(W_{s2} \cdot \tanh(W_{s1} \cdot H^T))$ where W_{s1} is a weight matrix of size $d_a \times u$ and the parameter d_a is the *attention depth* of our model. The matrix W_{s2} is a weight matrix of size $r \times d_a$ and the parameter r is the number of *attention hops* of our model.

The embedding matrix of our sequence is: $B = (b_1, b_2, \ldots, b_u) = AH$ and it has fixed size $r \times u$. In order to transform the embedding matrix into a vector $\vec{f}$ of size n, we flatten the matrix M and we feed the flattened matrix into a two-layers fully connected neural network with ReLU activation function: $\vec{f} = W_{out2} \cdot \mathsf{ReLU}(W_{out1} \cdot (b_1 \oplus b_2 \ldots \oplus b_u))$ where W_{out1} is a weight matrix of size $e \times (r+u)$, and W_{out2} a weight matrix of size $n \times e$.

Learning Parameters Using Siamese Architecture: we learn the network parameters:

$$\Phi = \{W_{s1}, W_{s2}, \overrightarrow{\mathsf{RNN}}, \overleftarrow{\mathsf{RNN}}, W_{out1}, W_{out2}\}$$

using a pairwise approach, a technique also called *siamese network* in the literature [31] (the same technique is also used in [10]). The main idea is to join two identical function embedding networks with a similarity score (with identical we mean that the networks share the same parameters). The final output of the siamese architecture is the similarity score between the two input graphs.

In more details, from a pair of input functions $< f_1, f_2 >$ two vectors $< \vec{f_1}, \vec{f_2} >$ are obtained by using the same function embedding network. These vectors are compared using cosine similarity as distance metric, with the following formula:

$$\mathrm{similarity}(\vec{f_1}, \vec{f_2}) = \frac{\sum_{i=1}^{n} \left(\vec{f_1}[i] \cdot \vec{f_2}[i] \right)}{\sqrt{\sum_{i=1}^{n} \vec{f_1}[i]} \cdot \sqrt{\sum_{i=1}^{n} \vec{f_2}[i]}} \quad (1)$$

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

To train the network we require in input a set of K functions pairs, $< \vec{f_1}, \vec{f_2} >$, with ground truth labels $y_i \in \{+1, -1\}$, where $y_i = +1$ indicates that the two input functions are similar and $y_i = -1$ otherwise. Then using the siamese network output, we define the following objective function:

$$J = \sum_{i=1}^{K} \left(\mathrm{similarity}(\vec{f_1}, \vec{f_2}) - y_i \right)^2 + \|(A \cdot A^T - I)\|_F$$

The objective function J is minimized by using, for instance, stochastic gradient descent. The term $\|(A \cdot A^T - I)\|_F$ is introduced to penalize the choice of the same weights for each attention hops in matrix A (see [12]).

Comparison with Gemini architecture:— As described in the related work section Gemini transforms a CFG into a vector. A CFG is a representation of a binary function as a graph where blocks of instructions are connected accordingly to the possible execution flow, see Figure 3. In its first step Gemini transforms a CFG into an annotated CFG, this is done by pre-processing each block of instructions transforming it into a vector of manually selected features (such as numbers of arithmetic instructions, numbers of calls, ...). The annotated

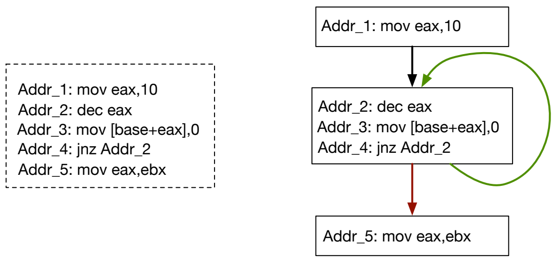


Fig. 3. CFG example: linear code on left and CFG and right.

CFG is then embedded into a vector using a *structure2vec* [23] neural network. Structure2vec takes into account the graph structure of the annotated CFG: it computes an embedding for each node in the network by multiple rounds in which it averages its value with the ones of its neighbours, and then it computes a cumulative graph embedding by aggregating the embeddings of the nodes.

We can highlight two main differences between our approach and Gemini. Gemini uses a manual annotation approach to transform a sequence of instructions into a vector, while we let the network decides how to do such transformation.The advantage is that we are not injecting a human bias in the feature selection, for example preferring arithmetic instructions over others. The self-attentive mechanism allows the network to automatically focus on the sequences of instructions that are more important.

The second difference is that our network does not use the structural information of the CFG. While this allows for better speed, it has the potential drawback of ignoring possible important information. However, from our experimental evaluation it seems that a self-attentive architecture reaches performance that are better or on par with structure2vec.

V. Implementation Details and Training

Implementation Details and i2v setup: — We developed a prototype implementation of SAFE using Python and the Tensorflow [32] framework. [5]. For static analysis of binaries we used the ANGR framework [33], radare2 and IDA Pro[6]. To train the network we used a batch size of 250, learning rate 0.001, Adam optimizer. In our SAFE prototype we used the following parameters: the RNN cell is the GRU cell [34]; the u value is 100, $r = 10$, $d_a = 250$, $e = 2000$, $n = 100$.

We decided to truncate the number of instructions inside each function to the maximum value of $m = 150$, this represents a good trade-off between training time and accuracy, the great majority of functions in our datasets is below this threshold (more than 90% of the functions).

a) I2v model: – We trained two i2v models using the two training corpora described below. One model is for the instruction set of ARM and one for AMD64. With this choice we tried to capture the different syntaxes and semantics of these two assembly languages. The model that we use for i2v (for both versions AMD64 and ARM) is the skip-gram implementation of word2vec provided in TensorFlow [35]. We used as parameters: embedding size 100, window size 8 and word frequency 8.

We collected the assembly code of a large number of functions, and we used it to build two training corpora for the i2v models, one for the i2v AMD64 model and one for the i2v ARM model. We built both corpora by disassembling several UNIX executables and libraries using IDA PRO. The libraries and the executables have been randomly sampled from repositories of Debian packages.

We avoided multiple inclusion of common functions and libraries by using a duplicate detection mechanism; we tested the uniqueness of a function computing an hash of all function instructions, where instructions are filtered by replacing the operands containing immediate and memory locations with a special symbol.

From 2.52 GBs of AMD64 binaries we obtained the assembly code of 547K unique functions. From 3.05 GBs of ARM binaries we obtained the assembly code of 752K unique functions. Overall the AMD64 corpus contains 86M assembly code lines while the ARM corpus contains 104M assembly code lines.

Training Single and Cross Platform Models: — We trained SAFE models using the same methodology of Gemini, see [10]. We trained both a single and a cross platform models that were then evaluated in several tasks (see Section VI for the results).

We considered two different datasets:

AMD64multipleCompilers Dataset: – This is dataset has been obtained by compiling the following libraries for AMD64: binutils-2.30, ccv0.7, coreutils-8.29, curl-7.61.0, gsl-2.5, libhttpd-2.0, openmpi-3.1.1, openssl-1.1.1-pre8, valgrind-3.13.0. The compilation has been done using 3 different compilers, clang-3.9, gcc-5.4, gcc-3.4[7] and 4 optimization levels (i.e., -O[0-3]). The compiled object files have been disassembled with ANGR, obtaining a total of 452598 functions.

AMD64ARMOpenSSL Dataset: – To align our experimental evaluation with state-of-the-art studies we built the AMD64ARMOpenSSL Dataset in the same way as the one used in [10]. In particular, the AMD64ARMOpenSSL Dataset consists of a set of 95535 functions generated from all the binaries included in two versions of Openssl (v1_0_1f - v1_0_1u) that have been compiled for AMD64 and ARM using gcc-5.4 with 4 optimizations levels (i.e., -O[0-3]). The resulting object files have been disassembled using ANGR; we discarded all the functions that ANGR was not able to disassemble.

Training: – We generate our training and test pairs as reported in [10]. The pairs can be of two kinds: similar pairs, obtained pairing together two binary functions originated by the same source code, and dissimilar pairs, obtained pairing randomly functions that do not derive from the same source code. Specifically, for each function in our datasets we create two pairs, a similar pair, associated with training label $+1$ and a dissimilar pair, training label -1; obtaining a total number of

[5]The source code and the datasets used in the evaluation are available at https://github.com/gadiluna/SAFE

[6]We designed our system to be compatible with several disassemblers, including two opensource solutions.

[7]Note that gcc-3.4 has been released more than 10 years before gcc-5.4.

pairs that is twice the total number of functions. The functions in AMD64multipleCompilers Dataset are partitioned in three sets: train, validation, and test (75%-15%-15%). The functions in AMD64ARMOpenSSL Dataset are partitioned in two sets: train and test (80%-20%), in this case we do not need the validation set because in Task 1 Section VI-A we will perform a cross-validation. The test and validation pairs will be used to assess performances in Task 1, see Section VI-A. As in [10], pairs are partitioned preventing that two similar functions are in different partitions (this is done to avoid that the network sees during training functions similar to the ones on which it will be validated or tested).

We train our models for 50 epochs (an epoch represents a complete pass over the whole training set). In each epoch we regenerate the training pairs, that is we create new similar and dissimilar pairs using the functions 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.

Inlining and function boundaries: – During the compilation process, if an high optimisation level is selected O2,O3, it is possible that simple functions are "inlined". That is if a function B calls a function A, the optimizer may decide to inline the assembly code of function A in B, obtaining a resulting function B(A). In this case, we decided to use the convention that only the matches between functions similar to B and function B(A) is correct. This means that our results could slightly underrepresent the actual performance of SAFE, since one could argue that B(A) is also similar to A, since it is performing also the functionalities of A.

A clarification is needed on how we detect functions inside a binary. During the compilation process the compiler is free to decide the layout of the binary, at function level this means that the compiler decides where to place the function, and it could even partition a single function in several locations. Therefore, to disassemble a function is necessary to detect its boundaries. This problem is orthogonal to our. In this paper we relay on the disassembler to find the correct boundaries. Papers have shown [36], [37] that modern disassemblers have around 94% of accuracy in detecting boundaries. In case of an occasional miss, we could lose the function or get a function with a corrupted assembly code. We accept this last outcome since it is not frequent and a limited quantity of corrupted data does not jeopardize the training.

VI. Evaluation of Representation Learning using SAFE

In this section we evaluate SAFE as a representation learning model (training it to create embedding vectors that are then tested on several different tasks). Regarding the representation learning power, we evaluate SAFE performances on several tasks, comparing our evaluation with the state of the art system Gemini[8]

[8]Gemini has not been distributed publicly. We implemented it using the information contained in [10]. For Gemini the parameters are: function embeddings of dimension 64, number of rounds 2, and a number of layers 2. These parameters are the ones that give the best performance for Gemini, according to our experiments and the one in the original Gemini paper.

- **Task 1 - Single Platform and Cross Platform Models Tests**: we test our single platform and cross platform models following the same methodology of [10]. We achieve a performance improvement of 6.8% in the single platform case and of 4.4% in the cross platform case. We remark that in these tests our models behave almost perfectly (within 1% from what a perfect model may achieve).
- **Task 2 - Function Search**: in this task we are given a certain binary function and we search for similar on a large dataset created using several compilers (including compilers that were not used in the training phase). We achieve a precision above 80% for the first 15 results, and a recall of 47% in the first 50 results.
- **Task 3 - Vulnerability Search**: in this task we evaluate our system on a use-case scenario in which we search for vulnerable functions. Our tests on several vulnerabilities show a recall of 84% in the first 10 results.
- **Task 4 - Semantic Classification**: in this task we classify the semantic of binary functions using the embeddings built with SAFE. We reach an accuracy of 95% on our test dataset. Moreover, we test our classifier on real-world malware, showing that we can identify encryption functions.
- **Task 5 - Advanced Persistent Threat (APT) Classification**: Leveraging the learned function embeddings, we build a classifier that given a malware, it outputs the possible APT group behind its development. We reach an f1 score of 89%

Finally, in Section VI-F we discuss the robustness of SAFE in the scenario of statically linked and stripped binaries.

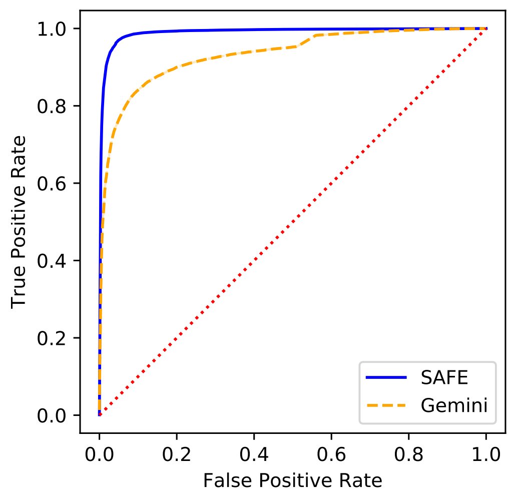
A. Task 1 - Single and Cross Platform tests

In this task we evaluate the performance of SAFE using the same testing methodology of Gemini. We split our dataset as discussed in Section V-0a.

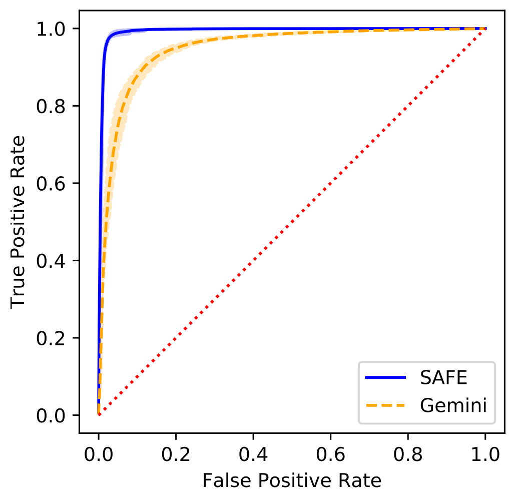
We perform two disjoint tests. (i) Using AMD64multipleCompilers Dataset, we first compute performance metrics on the validation set for all the epochs, then we use the model hyper parameters that led to the best performance on the validation set to compute a final performance score on the test set. (ii) Using AMD64ARMOpenSSL Dataset, we perform a 5-fold cross validation: we partition 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. The reported results are the average of 5 independent runs, one for each possible fold chosen as test set. This approach is more robust than a fixed train/validation/test split since it reduces the variability of the results.

As in [10], we measure the performance using the *Receiver Operating Characteristic* (ROC) curve [38]. Following the best practices of the field we measure the *area under the ROC curve*, or AUC (Area Under Curve). Loosely speaking, higher the AUC value, better the predictive performance of the algorithm.

Results:

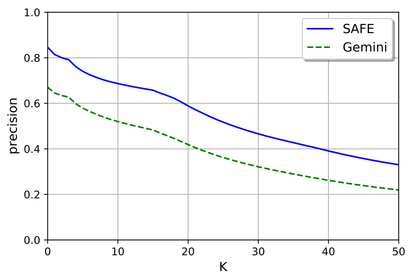


(a) AMD64multipleCompilers Dataset.

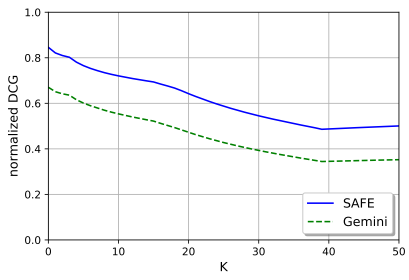


(b) AMD64ARMOpenSSL Dataset.

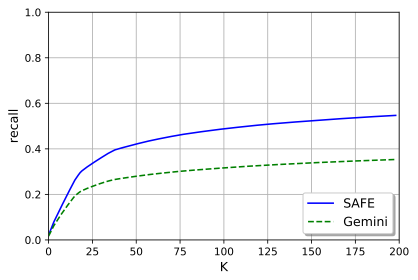
Fig. 4. ROC curves for Task 1 - Validation and Test of Single Platform and Cross Platform Models.



(a) Precision for the top-k answers with $k \leq 50$.



(b) nDCG for the top-k answers with $k \leq 50$.



(c) Recall for the top-k answers with $k \leq 200$.

Fig. 5. Results for Task 2 - Function Search, on AMD64FuntionSearch Dataset (581K functions) average on 160K queries.

AMD64multipleCompilers Dataset: – The results for the single platform case are in Figure 4a. Our AUC is 0.99, the AUC of Gemini is 0.932.

AMD64ARMOpenSSL Dataset: – We compare ourselves with Gemini in the cross-platform case. The results are in Figure 4b and they show the average ROC curves on the five runs of the 5-fold cross validation. The Gemini results are reported with orange dashed line while we use a continuous blue line for our results. For both solutions we additional highlighted the area between the ROC curves with minimum AUC maximum AUC in the five runs. The better prediction performance of SAFE is clearly visible; the average AUC obtained by Gemini is 0.948 with a standard deviation of 0.006, while the average AUC of SAFE is 0.992 with a standard deviation of 0.002. The average improvement with respect to Gemini is of 4.4%.

We also compared SAFE and VulSeeker [24] using AMD64ARMOpenSSL Dataset. We used the pretrained model provided in their repository[9]. Results show that VulSeeker reaches an AUC of 0.85%, lower than the one reached by Gemini.

[9]https://github.com/buptsseGJ/VulSeeker

B. Task 2 - Function Search

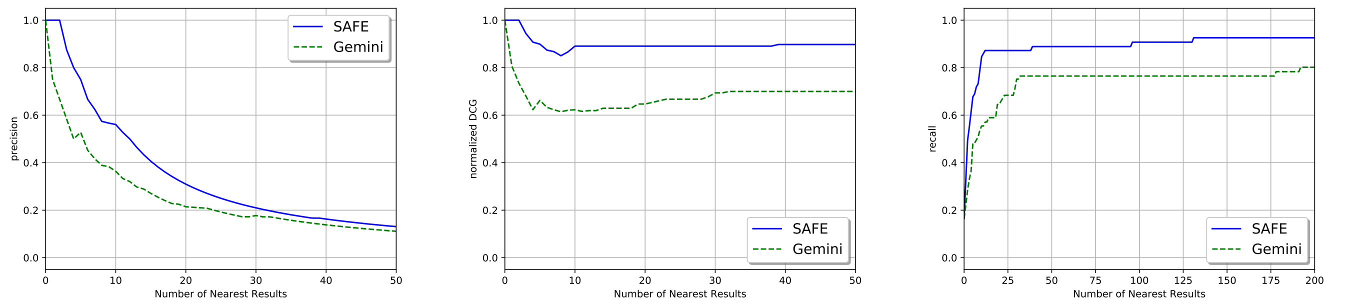
In this task we evaluate the function search capability of the model trained on AMD64multipleCompilers Dataset. We take a target function f, we compute its embedding $\vec{f}$ and we search for similar functions in the AMD64FuntionSearch Dataset (details of this dataset are given below). Given the target $\vec{f}$, a search query returns $R_{\vec{f}} : (r_1, r_2, \ldots, r_k)$, that is the ordered list of the k nearest embeddings in AMD64FuntionSearch Dataset.

We built AMD64FuntionSearch Dataset by compiling multiple projects [10] for AMD64 using 10 compilers[11] For each compiler we used all 4 optimization levels. We took the object files, i.e. we did not create the executable by linking objects file together, and we disassembled them with radare2, obtaining a total of 581640 functions. For each function the AMD64FuntionSearch Dataset contains an average number of 33 similars. We do not reach an average of 48[12] similars because of inlining and disassembler fails on some functions.

[10] adns-1.5.0, dietlibc-0.34, ffmpeg-4.0.2, glib-2.5.0, libogg-1.3.4, musl-1.2.1, nginx-052, postgresql_9.6, tcc-0.9.27

[11]gcc-3.4, gcc-4.7, gcc-4.8, gcc-4.9, gcc-5.4, gcc-6, gcc-7, clang-3.8, clang-3.9, clang-4.0, clang-5.0, clang-6.0.

[12]48= 12 compilers $\times$ 4 optimizations level



(a) Precision for the top-k answers with $k \leq 50$. (b) nDCG for the top-k answers with $k \leq 50$. (c) Recall for the top-k answers with $k \leq 200$.

Fig. 6. Results for Task 3 - Vulnerability Search.

We compute the usual measures of *precision*, fraction of similar functions in $R_{\vec{f}}$ over all functions in $R_{\vec{f}}$, and *recall*, fraction of similar functions in $R_{\vec{f}}$ over all similar functions in the dataset. Moreover, we also compute the *normalised Discounted Cumulative Gain (nDCG)* [39]:

$$nDCG(R_{\vec{f}}) = \frac{\sum_{i=1}^{k} \frac{isSimilar(r_i, \vec{f})}{\log(1+i)}}{IdealDCG_k}$$

Where $isSimilar$ is 1 if r_i is a function similar to $\vec{f}$ or 0 otherwise, and, $IdealDCG_k$ is the Discounted Cumulative Gain of the optimal query answering. This measure is between 0 and 1, and it takes into account the ordering of the similar functions in $R_{\vec{f}}$, giving better results to responses that put similar functions first. As an example let us suppose we have two results for the same query: $(1,1,0,0)$ and $(1,0,0,1)$ (where 1 means that the corresponding index in the result list is occupied by a similar function and 0 otherwise). These results have the same precision (i.e., $\frac{1}{2}$), but nDCG scores the first better.

Results: – our results on precision, nDCG and recall are reported in Figure 5. Performances were calculated by averaging the results of 160K queries. The queries are obtained by sampling, in AMD64FuntionSearch Dataset, 10K functions for each compiler and optimization level in the set {clang-4.0,clang-6.0,gcc-4.8,gcc-7} × {O0,O1,O2,O3}.

Precision: The results are reported in Figure 5a. The precision is above or around 70% for $k \in [0, 10]$, and it is above 60% for $k \in [0, 20]$. The increase of performance on Gemini is around 10% on the entire range considered. Specifically at $k \in \{10, 20, 30, 40, 50\}$ we have values $\{69\%, 61\%, 48\%, 40\%, 34\%\}$ for SAFE and $\{53\%, 43\%, 33\%, 27\%, 22\%\}$ for Gemini.

nDCG: The tests are reported in Figure 5b. Our solution has a performance above 70% for $k \in [0, 15]$. This implies that we have a good order of the results and the similar functions are among the first results returned. The value is always above or around 50%. There is a clear improvement with respects to Gemini, the increase is around 10% on the entire range considered. Specifically, at $k \in \{10, 20, 30, 40, 50, 100, 200\}$ we have values $\{73\%, 66\%, 55\%, 49\%, 50\%, 54\%, 57\%\}$ for SAFE and $\{56\%, 48\%, 40\%, 34\%, 35\%, 37\%, 39\%\}$ for Gemini.

Recall: The tests are reported in Figure 5c. We have a recall at $k = 40$ of 40% (vs. 28% Gemini), the recall at $k = 200$ is 56% (vs. 45% Gemini). Specifically at $k \in \{10, 20, 30, 40, 50, 100, 200\}$ we have values $\{17\%, 30\%, 36\%, 40\%, 42\%, 49\%, 55\%\}$ for SAFE and $\{13\%, 22\%, 25\%, 27\%, 28\%, 32\%, 35\%\}$ for Gemini.

C. Task 3 - Vulnerability Search

In this task we evaluate our ability to look up for vulnerable functions on a dataset specifically designed for this purpose. The methodology and the performance measures of this test are the same of Task 2.

The dataset used is the vulnerability dataset of [5]. It contains several vulnerable binaries compiled with 11 compilers in the families of clang, gcc and icc. The total number of different vulnerabilities is 8[13]. We disassembled the dataset with ANGR, obtaining 3160 binary functions. The average number of vulnerable functions for each of the 8 vulnerabilities is 7.6; with a minimum of 3 vulnerable functions and a maximum of 13[14]. We performed a lookup for each of the 8 vulnerabilities, computing the precision, nDCG, and recall on each result. Finally, we averaged these performance over the 8 queries.

Results: – the results of our experiments are reported in Figure 6. We can see that SAFE outperforms Gemini for all values of k in all tests. Our nDCG is very large, showing that SAFE effectively finds most of the vulnerable functions in the nearest results. For $k = 10$ we reach a recall of 84%, while Gemini reaches a recall of 55%. For $k = 15$ our recall is 87% (vs. 58% recall of Gemini, with an increment of performance of 29%), and we reach a maximum of 88% (vs. 76% of Gemini). One of the reasons why the accuracy quickly decreases is that, on average, we have 7.6 similar functions; this means that even a perfect system at $k = 20$ will have an accuracy that is less than 50%. This metric problem is not shared by the nDCG reported in Figure 6b, recall that the nDCG is normalized on the behaviour of the perfect query answering system. During our tests we have seen that on the infamous hearthbleed vulnerability we have an ideal behaviour,

[13]cve-2014-0160, cve-2014-6271, cve-2015-3456, cve-2014-9295, cve-2014-7169, cve-2011-0444, cve-2014-4877, cve-2015-6862.

[14]Some vulnerable functions are lost during the disassembling process

TABLE I
NUMBER OF FUNCTION FOR EACH CLASS IN THE SEMANTIC DATASET

Class	Number of functions
S (Sorting)	4280
E (Encryption)	2868
SM (String Manipulation)	3268
M (Math)	4742
Total	15158

SAFE found all the 13 vulnerable functions in the first 13 results, while Gemini had a recall at 13 around 60%.

D. Task 4 - Semantic Classification

Loosely speaking, a function f can be seen as an implementation of an algorithm. We can partitions algorithms into *classes*, where each class is a group of algorithms solving related problems. In this paper we focus on four classes {E (Encryption), S (Sorting), SM (String Manipulation), M (Mathematical)}. A function belongs to class E if it is the implementation of an encryption algorithm (e.g., AES, DES); it belongs to S class if it implements a sorting algorithm (e.g., bubblesort, mergesort); it belongs to SM class if it implements an algorithm to manipulate a string (e.g., string reverse, string copy); it belongs to M class if it implements math operations (e.g., computing a bessel function); We say that a classifier, recognizes the semantic of a function f, with f taken from one of the aforementioned classes, if it is able to guess the class to which f belongs.

Test description and dataset: – In Task 4 we evaluate the semantic classification using the embeddings computed with the model trained on AMD64multipleCompilers Dataset. We calculate the embeddings for all functions in Semantic Dataset (details on the dataset below). We split our embeddings in train set and test set and we train and test an SVM classifier using a 10-fold cross validation. We use an SVM classifier with kernel rbf, and parameters $C = 10$ and $\gamma = 0.01$. We compare our embeddings with the ones computed with Gemini.

The Semantic Dataset has been generated from a source code collection containing 443 functions that have been manually annotated as implementing algorithms in one of the 4 classes: E (Encryption), S (Sorting), SM (String Manipulation), M (Mathematical). Semantic Dataset contains multiple functions that refer to different implementations of the same algorithm. We compiled the sources for AMD64 using the 12 compilers and 4 optimizations used for AMD64FuntionSearch Dataset, we took the object files and after disassembling them with ANGR we obtained a total of 15158 binary functions, see details in Table I. It is customary to use auxiliary functions when implementing complex algorithms (e.g. a swap function used by a quicksort algorithm). When we disassemble the Semantic Dataset we take special care to include the auxiliary functions in the assembly code of the caller. This step is done to be sure that the semantic of the function is not lost due to the scattering of the algorithm semantic among helper functions. Operatively, we include in the caller all the callees up to depth 2. As performance measures we considered precision, recall and F-1 score.

TABLE II
RESULTS OF SEMANTIC CLASSIFICATION USING EMBEDDINGS COMPUTED WITH SAFE MODEL AND GEMINI. THE CLASSIFIER IS AN SVM WITH KERNEL *rbf*, $C = 10$ AND $gamma = 0.01$.

Class	Emb. Model	Precision	Recall	F1
E (Encryption)	**SAFE**	**0.92**	**0.94**	**0.93**
	Gemini	0.82	0.85	0.83
M (Math.)	**SAFE**	**0.98**	**0.95**	**0.96**
	Gemini	0.96	0.90	0.93
S (Sorting)	**SAFE**	**0.91**	**0.93**	**0.92**
	Gemini	0.87	0.92	0.89
SM (String Manipulation)	**SAFE**	**0.98**	**0.97**	**0.97**
	Gemini	0.90	0.89	0.89
Weighted Average	**SAFE**	**0.95**	**0.95**	**0.95**
	Gemini	0.89	0.89	0.89

Results: – The results of our semantic classification tests are reported in Table II. First and foremost, we have a strong confirmation that is indeed possible to classify the semantic of the algorithms using function embeddings. The use of an SVM classifier on the embedding vector space leads to good performance. There is a limited variability of performances between different classes. The classes on which SAFE performs better are SM and M. We speculate that the moderate simplicity of the algorithms belonging to these classes creates a limited variability among the binaries. The M class is also one of the classes where the Gemini embeddings are performing better, this is probably due to the fact that one of the manual features used by Gemini is the number of arithmetic assembly instructions inside a code block of the CFG. By analyzing the output of the classifier we find out that the most common error, a mistake common to both Gemini case and SAFE, is the confusion between encryption and sorting algorithms. A possible explanation for this behaviour is that simple encryption algorithms, such as RC5, share many similarities with sorting algorithms (e.g., nested loops on an array). Finally, we can see that, in all cases, the embeddings computed with our architecture outperform the ones computed with Gemini; the improvement range is between 10% and 2%. The average improvement, weighted on the cardinality of each class, is around 6%.

Ablation study on callee inclusion: – We assessed the impact of inserting the callee on the caller function by performing an experiment without such inclusion. In this case the weighted average of the performances of safe goes from 95% to 90% with an average loss of performances of 5% percent. The new f1-scores are: E 87%, M 95%, S 86%, SM 91%. As we can see the classes that are more impacted are E and S, a likely reason is that these functionalities are usually implemented using helper functions.

Qualitative Analysis of the Embeddings: – We performed a qualitative analysis of the embeddings produced with SAFE. Our aim is to understand how the network captures the information on the inner semantics of the binary functions, and how it represents such information in the vector space.

To this end we computed the embeddings for all functions in Semantic Dataset. In Figure 7 we report the two-

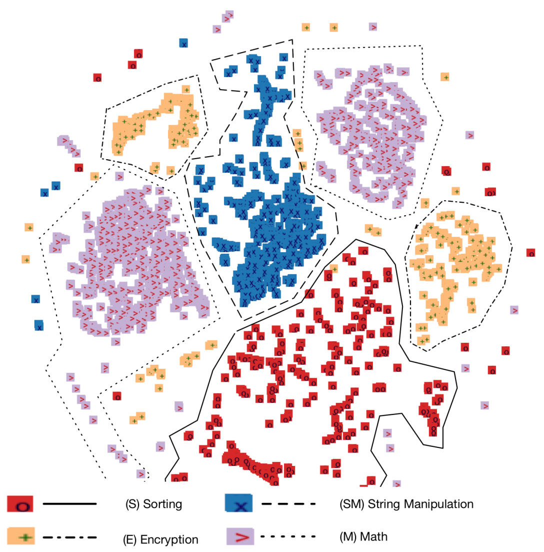


Fig. 7. 2-dimensional visualization of the embedding vectors for all binary functions in Semantic Dataset. The four different categories of algorithms (Encryption, Sorting, Math and String Manipulation) are represented with different symbols and colors.

TABLE III
APT CLASSIFICATION RESULTS. AVERAGE OF OUR 5-FOLD CROSS VALIDATION TESTS. THE SUPPORT IS A DECIMAL NUMBER BEING AN AVERAGE OVER 20 TESTS. SVM CLASSIFIER WITH KERNEL *rbf*, $C = 10$ AND $\gamma = 0.01$.

	f1-score	precision	recall	support
APT28	0.845	0.735	1.000	9.560
APT29	0.876	0.878	0.875	28.209
APT30	0.945	0.984	0.910	17.354
Carbanak	0.664	0.527	0.901	7.585
Desert Falcon	0.775	0.670	0.937	5.862
Hurricane Panda	0.976	0.960	0.992	60.002
Lazarus Group	0.872	0.986	0.783	13.910
Mirage	0.740	0.681	0.812	8.446
Patchwork	0.953	0.971	0.936	98.595
Sandworm	0.624	1.000	0.455	17.774
Shiqiang	0.718	0.625	0.858	3.758
Transparent Tribe	0.852	0.794	0.927	9.483
Violin Panda	0.854	0.768	0.969	3.210
Volatile Cedar	0.964	1.000	0.934	7.529
Winnti Group	0.905	0.903	0.910	34.857
macro avg	0.837	0.832	0.880	326.134
weighted avg	0.898	0.917	0.899	326.134
accuracy	0.899			

dimensional projection of the 100-dimensional vector space where binary functions embeddings lie, obtained using the *t-SNE* [15] visualisation technique [40]. From Figure 7 is possible to observe a quite clear separation between the different classes of algorithms considered. We believe this behaviour is really interesting and it further confirms our quantitative experiments on semantic classification.

Real use case of Task 4 - Detecting encryption functions in Windows Malware: – We tested the semantic classification on a real use case scenario. We trained a new SVM classifier using the semantic dataset with only two classes, encryption and non-encryption. We then used this classifier to analyze two samples of window malware found in famous malware repositories: the *TeslaCrypt* and *Vipasana* ransomwares. We disassembled the samples with radare2, we included in the caller the code of the callee functions up to depth 2. We processed the disassembled functions with our classifier, and we selected only the functions that are flagged as encryption with a probability score greater than 96%. Finally, we manually analyzed the malware samples to assess the quality of the selected functions.

TeslaCrypt[16]. On a total of 658 functions, the classifier flags the ones at addresses 0x41e900, 0x420ec0, 0x4210a0, 0x4212c0, 0x421665, 0x421900, 0x4219c0. We confirmed that these are either encryption (or decryption) functions or helper functions directly called by the main encryption procedures.

Vipasana [17]. On a total of 1254 functions, the classifier flags the ones at addresses 0x406da0, 0x414a58, 0x415240. We confirmed that two of these are either encryption (or decryption) functions or helper functions directly called by the main encryption procedures. The false positive is 0x406da0.

As final remark, we want to stress that these malware are for windows and they are 32-bit binaries, while we trained our entire system on ELF executables for AMD64. This shows that our model is able to generate good embeddings also for cases that are largely different from the ones seen during training.

E. Task 5 - APT Classification

Dataset: – We use the APT dataset from [13]. The dataset contains the mapping between malware and APT developers, it has been created by using public reports[18] of APT activites. From the reports the authors have obtained an attribution for each malware sample in their dataset; they have taken special care to remove malware for which the attribution was doubtful or shared among more than one APT.

We extracted a set of malware and the corresponding APT developers, and successfully disassembled, using IDA pro, 1656 malware. For each malware all functions recognised by IDA as library functions have been discarded. Finally, using SAFE model trained on AMD64multipleCompilers Dataset, we computed the embeddings of all the functions containing more than 20 instructions. We released the code used in our experiments[19].

Program Representation: – Embeddings are representations of functions, but to classify a malware we need to represent a whole program. For us a program P_j is a set of functions embeddings $P_j = \{\vec{f_0}, ..., \vec{f_{n_j}}\}$. A function that indicates whether P_j contains function $\vec{f}$ can be defined as:

$$\alpha(\vec{f}, P_j) = \begin{cases} 1 \text{ if } \exists \;\; \vec{f_x} \in P_j \;\text{ s.t. }\; \vec{f} \cdot \vec{f_x} > \tau \\ 0 \text{ else} \end{cases} \tag{2}$$

[15]We used the TensorBoard implementation of *t-SNE*

[16]Sample available at https://github.com/ytisf/theZoo/tree/master/malwares/Binaries/Ransomware.TeslaCrypt – Hash: *3372c1eda...4a370*

[17]Sample available at https://github.com/ytisf/theZoo/tree/master/malwares/Binaries/Ransomware.Vipasana – Hash: *0442cfabb...4b6ab*

[18]The sources reported in [13] are: MITRE ATT&CK, PT Groups and Operations, APT Notes, MISP Galaxy Cluster.

[19]https://github.com/lucamassarelli/safe_apt_classification

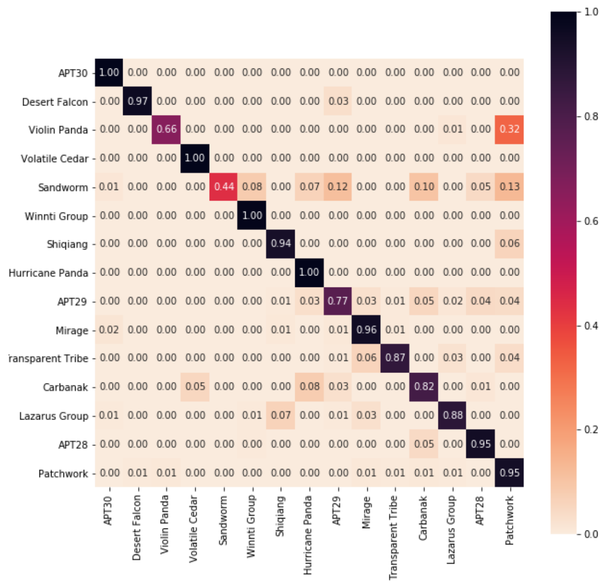


Fig. 8. Confusion Matrix. Average of our 5-fold cross validation tests.

Where τ is a threshold (equal to 0.95 in our experiments). Given α the similarity between two programs is:

$$S(P_i, P_j) = \frac{\sum_{\forall \vec{f} \in P_i} \alpha(\vec{f}, P_j)}{\max(|P_i|, |P_j|)} \tag{3}$$

Classify programs: – The non-trivial difficulty is to deal with a representation of programs that has a well defined similarity metric (see above), but where programs are not points in a metric space. The latter is a desirable requirement to use features based classification techniques. From our similarity measure we obtained a feature-based representation of programs using a known trick [41]. We randomly sample a subset of programs and use them as a "*base*" and then we represent any other program as a vector of distances from base programs. On such vectors we trained an SVM classifier with kernel *rbf*, $C = 10$ and $\gamma = 0.01$.

Test setting and results: – We split randomly our dataset in five different folds. The distribution of classes between the 5 different fold is the same. In each experiment we use one fold as test, one fold as base and the 3 remaining fold for training the classifier. We tested all possible 20 combinations. Results in Table III are the average over all the experiments, while in Figure 8 there is the average of the confusion matrices of our experiments.

From the average confusion matrix we can see that there are specific APT on which our classifier performs badly. The most egregious example are Violin Panda, Sandworm, and Carbanak. For Violin Panda, the nationality of the group has been attributed to China, and it is mostly confused with Patchwork. Patchwork is a group known to copy code from public sources. The second reason is that we found our classifier to be unbalanced toward Patchwork, this is probably due to the highest number of samples available for this APT.

For Sandworm, the group is allegedly based in Russia and nationally sponsored. It is confused with Patchwork (for the same reasons explained above); with APT29, that is also a group of russian origin and nationally, and with Carbanak. The similarity with APT29 could be explained by code sharing.

Carbanak is confused only with APTs groups related to Russia (Sandworm, APT28, APT29), there are reports indicating that the part of their codebase was leaked by russian ip-addresses (https://www.entrada.co.ke/2019/05/07/kaspersky-lab-carbanak-source-code-leak-whats-next/).

In general, we can observe that russian attributed APTs are more prone to be confused between each other (see APT28, APT29).

F. Robustness to static linking and stripping

We tested SAFE when applied to statically linked and stripped binaries. This has been done to confirm that our solution is agnostic with respect to debug symbols and static linking. We compiled the botnet Mirai with multiple compilers and optimization levels; using the flag "-static" to statically link the libraries inside the final executable. After compilation we obtained a stripped copy removing all references to symbols in the binary using the unix strip command. At the end of this process we have two datasets, one containing functions of the stripped binaries and one with functions from unstripped binaries. We perform three different function-search experiments aimed at understanding if there is a difference of results when using unstripped or stripped functions.

We selected a subset of 800 functions to be used as queries, for each of this source function we obtained a set of binary functions taking the source compiled with 4 compilers and 4 optimizations levels. In each experiment we search using these query function to find their similars (this is analogous to Task 2). What we variate is the combination between stripped/unstripped functions used for the query over a database of stripped/unstripped functions. In the first experiment we use unstripped query functions over an unstripped dataset. In the second experiment we strip the query functions and we search over an unstripped dataset. In the third we strip both the queries and the dataset. In all cases we obtain the same results for recall, nDCG and precision. This confirms that SAFE is not sensible to symbols in the binary and static linking, as a matter of fact the embeddings computed for stripped and unstripped functions are equals.

VII. Evaluation of SAFE on Compiler Provenance

We decided to test SAFE on a task that is completely unrelated to the binary similarity one to assess the ability to abstract from the specific features needed to solve a particular class of problems. For this purpose we investigated the compiler provenance task (i.e., to understand which compiler produced a given binary function [14]). This problem can be seen as the dual of binary similarity: in one task the network has to focus on the differences introduced by compilers, while in the other it has to do the opposite. Our tests show an accuracy of 97.4%

for compiler family classification, reaching performances that are comparable with the state-of-the-art [14].

We ran two compiler provenance tests. In one we aim to detect the compiler family that generated a binary (i.e., gcc, clang, icc), in the second the optimisation level (as common in the literature we map $\{O0,O1\}$ to Low optimization level, and $\{O2,O3\}$ to High).

DNN details: – For this problem we train SAFE end-to-end with a feed-forward two-layers neural network. Given the embedding vector $\vec{f}$ we obtain a vector of classification probabilities as follows:

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

The loss function is the usual cross-entropy.

Dataset, methodology and performance measure: – We built a dataset using AMD64 binaries. The dataset contains 11 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, compiled with 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 obtained binaries and removed duplicates, totalising 1587648 binary functions.

We randomly split the dataset in train set, validation set and test set (the split is 70%-15%-15%). We keep functions generated by the same source in a single split.

We trained our models for 50 epochs and computed the performance metric on the validation set for all the epochs.

The model with the best accuracy on the validation split is used to compute the final performance score on the test split. 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. The hidden layer of the feed forward classifier has size 3000.

Results: – The results are reported in Tables IV, V. For the compiler family classification task we obtain an overall weighted accuracy of 97.4% (Table IV). The breakdown of classification performance is reported in the confusion matrices of Figure 9. In Figure 9a there is the confusion matrix for the compiler family identification, as we can see the classifier has more problem to discriminate between clang and gcc than icc. This is not surprising as they are both open-source projects that use similar techniques in the code generation. We think that the classifier is less precise on clang than gcc due to the difference in the support size. This result is inline with the state-of-the-art [14] (reported 98% of accuracy). Unfortunately, a direct comparison with [14] is unfeasible: 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). As reported in [25], Gemini reaches an accuracy of 80% on the compiler family classification task. s For the task of identifying the optimization we reach an overall accuracy of 91% (Table IV); this result beats the one reported by [42] where DNNs are used to identify the optimization level of binaries compiled uniquely with gcc (they report an accuracy of 89% on their dataset). Our accuracy is slightly worse than the one reported by [14] (reported accuracy of 98%).

TABLE IV
COMPILER CLASSIFICATION RESULT

Class	Precision	Recall	F1	Support
clang	0.96	0.97	0.97	66864
gcc	0.98	0.98	0.98	123490
icc	0.97	0.97	0.97	47646
avg / total	0.97	0.97	0.97	238000

TABLE V
OPTIMIZATION RESULT

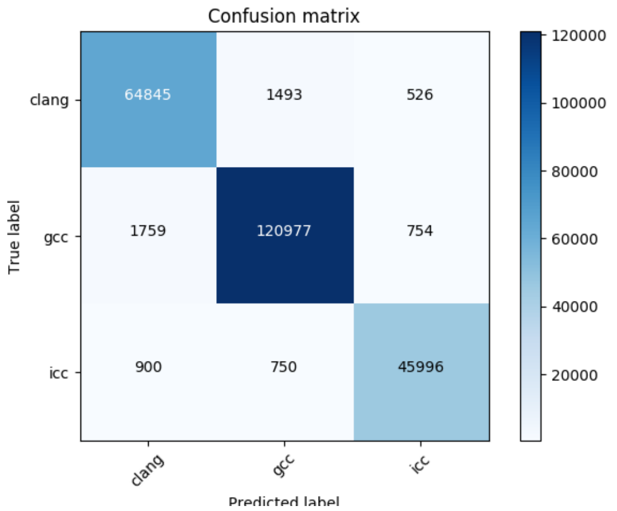
Class	Precision	Recall	F1	Support
H	0.88	0.89	0.89	97090
L	0.93	0.92	0.92	140910
avg / total	0.91	0.91	0.91	238000

VIII. SPEED CONSIDERATIONS.

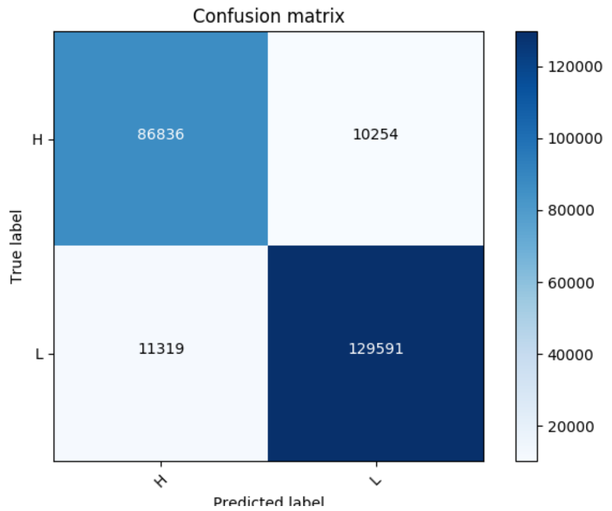
As reported in the introduction, an advantage of SAFE is ditching the use of CFGs. From our tests on radare2 disassembling a function is 10 times faster than asking for its CFG. Once functions are disassembled an Nvidia K80 running our model computes the embeddings of 1000 functions in around 1 second. More precisely, we run our tests on a virtual machine hosted on Google cloud platform. The machine has 8 core Intel Sandy Bridge, 30gb of ram, an Nvidia K80 and SSD hard-drive. We disassembled all object files in postegres 9.6 compiled with gcc-6 for all optimizations. The time needed to disassemble and pre-process 3432 binaries is 235 seconds, the time needed to compute the embeddings of the resulting 32592 functions is 33.3 seconds. The end-to-end time to compute embeddings for all functions in postgres starting from binary files is less than 5 minutes. We repeated the same test with openssl 1.1.1 compiled with gcc-5 for all optimizations. The end-to-end time to compute the embeddings for all functions in openssl is less than 4 minutes. Our implementation of Gemini is up to 10 times slower, it needs 43 minutes for postgres and 26 minutes for openssl.

IX. CONCLUSIONS

In this paper we introduced SAFE an architecture for computing embeddings of functions in the cross-platform case that does not use debug symbols. SAFE does not need the CFG, and this leads to a considerable speed advantage. It creates thousand embeddings per second on a mid-COTS GPU. This considerable speed comes with a significant increase of predictive performances with respect to the state of the art. There are several immediate lines of improvement that we plan to investigate in the immediate future. The first one is to retrain our i2v model to make use of libc call symbols. This will allow us to quantify the impact of such information on embedding quality. We believe that symbols could lead to a further increase of performance, at the cost of assuming



(a) Compiler Family Classification, accuracy 0.97



(b) Optimization Classification, accuracy 0.91

Fig. 9. Confusion matrices of the Compiler Family classification tasks, and Optimization classification test.

more information and the integrity of the binary that we are analyzing.

Acknowledgments. This work has been partially funded by the AXA Postdoctoral Fellowship of Giuseppe Di Luna, by Sapienza project RM11916B75A3293D, and by the PNRM project "SAFE". The authors thank Google for its support through the Academic Research Credits Program and NVIDIA Corporation for the donation of a GPGPU.

References

[1] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 309–329.

[2] T. Dullien, R. Rolles, and R. universitaet Bochum, "Graph-based comparison of executable objects," in *Proceedings of Symposium sur la sécurité des technologies de l'information et des communications, (STICC)*, 2005.

[3] 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.

[4] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digital Investigation*, vol. 12, pp. S61 – S71, 2015.

[5] 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.

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

[7] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proceedings of 23rd USENIX Security Symposium, (USENIX Security)*, 2014, pp. 303–317.

[8] G. Bonfante, M. Kaczmarek, and J. Marion, "Morphological detection of malware," in *2008 3rd International Conference on Malicious and Unwanted Software (MALWARE)*, 2008, pp. 1–8.

[9] 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.

[10] 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.

[11] 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.

[12] Z. Lin, M. Feng, C. Nogueira dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A structured self-attentive sentence embedding," *Arxiv: arXiv:1703.03130*, 2017.

[13] G. Laurenza and R. Lazzeretti, "dAPTaset: a comprehensive mapping of APT-related data," in *Proceedings of the 1st International Workshop on Security for Financial Critical Infrastructures and Services (FINSEC)*, 2019.

[14] 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*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 100–110. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001433

[15] I. U. Haq and J. Caballero, "A survey of binary code similarity," *CoRR*, vol. abs/1909.11424, 2019. [Online]. Available: http://arxiv.org/abs/1909.11424

[16] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference, (ACSAC)*. ACM, 2014, pp. 406–415.

[17] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*, 2014, pp. 349–360.

[18] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, ser. PPREW '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2430553.2430558

[19] 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.

[20] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proceedings of the 12th ACM on Asia Conference on Computer and Communications Security, (ASIA CCS)*. ACM, 2017, pp. 346–359.

[21] 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.

[22] 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.

[23] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent vari-

able models for structured data," in *Proceedings of the 33rd International Conference on Machine Learning, (ICML)*, 2016, pp. 2702–2711.

[24] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 896–899. [Online]. Available: https://doi.org/10.1145/3238147.3240480

[25] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *In Proceedings of the 2nd workshop on binary analysis research, (BAR)*, 2019.

[26] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[27] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/deepbindiff-learning-program-wide-code-representations-for-binary-diffing/

[28] 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.

[29] Z. Xiaochuan, S. Wenjie, P. Jianmin, L. Fudong, and M. Zhen, "Similarity metric method for binary basic blocks of cross-instruction set architecture," in *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2020.

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

[31] 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.

[32] 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.

[33] 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.

[34] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, (EMNLP)*, 2014.

[35] TensorFlow Authors, "Word2vec skip-gram implementation in tensorflow," in *https://www.tensorflow.org/tutorials/representation/word2vec*, last accessed 11/2018.

[36] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. USA: USENIX Association, 2015, p. 611–626.

[37] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 845–860.

[38] 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.

[39] A. Al-Maskari, M. Sanderson, and P. Clough, "The relationship between ir effectiveness measures and user satisfaction," in *Proceedings of the 30th International ACM Conference on Research and Development in Information Retrieval, (SIGIR)*, 2007, pp. 773–774.

[40] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[41] Y. Chen, E. K. Garcia, M. R. Gupta, A. Rahimi, and L. Cazzanti, "Similarity-based classification: Concepts and algorithms," *J. Mach. Learn. Res.*, vol. 10, pp. 747–776, Jun. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1577069.1577096

[42] Y. Chen, Z. Shi, H. Li, W. Zhao, Y. Liu, and Y. Qiao, "Himalia: Recovering compiler optimization levels from binaries by deep learning," in *Intelligent Systems and Applications*, 2019, pp. 35–47.

Luca Massarelli He got a master degree in Physics in 2017. Currently, he is a Ph.D student at Sapienza, University of Rome. Its main research topic is Natural Language Processing techniques applied to binary analysis. During summer 2019 he was research intern in Facebook AI research doing research on neural language modelling.

Giuseppe Antonio Di Luna Giuseppe Antonio Di Luna got his Ph.D. from the Sapienza, University of Rome in 2015, with a thesis on counting in anonymous dynamic networks. After his Ph.D. he did a postdoc at the University of Ottawa, working on fault-tolerant distributed algorithms, distributed robotics, and algorithm design for programmable particles. In 2018 he started a postdoc at the Aix-Marseille University, where he worked on dynamic graphs. Currently, he is working at Sapienza, University of Rome, performing research on applying NLP techniques to the binary analysis domain.

Fabio Petroni Fabio Petroni is a research engineer at Facebook AI Research. Prior to joining Facebook, he was with the R&D department of Thomson Reuters as a research scientist and received PhD and MSc degrees from Sapienza University of Rome. His main areas of expertise are natural language processing, machine learning and distributed systems. Fabio's research experience includes the development of natural language processing models for relational analysis, studying the spreading of misinformation online and developing distributed training techniques, among others.

Roberto Baldoni Roberto Baldoni is Full Professor at the University of Rome "La Sapienza". His research interests include distributed computing, dependable and secure distributed systems, distributed information systems and distributed event-based processing. His research at the University of Rome has been funded along the years by the European Commission, the Ministry of Italian Research, IBM, Microsoft, Finmeccanica and Telecom Italia. In 2010, he received the Science2business Award and the IBM Faculty Award. He is the author of around 150 research papers from theory to practice of distributed systems.

Leonardo Querzoni Leonardo Querzoni is associate professor at Sapienza University of Rome. He got his PhD in 2007 with a thesis on efficient data routing algorithms for publish/subscribe middleware systems. His research interests range from cyber security to distributed systems and focus, in particular, on topics that include binary similarity, distributed stream processing, dependability and security in distributed systems. He authored more than 80 paper published in international scientific journals and conferences. In 2016 he co-authored the Italian National Framework for Cyber Security as member of Cyber Intelligence and Information Security research center at Sapienza University of Rome. In 2017 he got the Test of Time Award from the ACM International Conference on Distributed Event-Based Systems for the paper "TERA: topic-based event routing for peer-to-peer architectures", published in 2007. In 2014 he was general chair for the International Conference on Principles of Distributed Systems, and in 2019 he was the program co-chair for the ACM International Conference on Distributed Event-Based Systems.