SAFE: Self-Attentive Function Embeddings for Binary Similarity

Luca Massarelli†, Giuseppe Antonio Di Luna‡, Fabio Petroni*,
Leonardo Querzoni†, Roberto Baldoni†

†: University of Rome Sapienza. {massarelli, querzoni, baldoni}@diag.uniroma1.it.

‡: CINI, National Laboratory of Cyber Security. g.a.diluna@gmail.com.

∗: Facebook AI Research, petronif@acm.org.

Abstract

The binary similarity problem consists in determining if two functions are similar by only considering 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., and thus have an immediate practical impact. Current solutions compare functions by first transforming their binary code in multi-dimensional vector representations (embeddings), and then comparing vectors through simple and efficient geometric operations. However, embeddings are usually derived from binary code using manual feature extraction, that may fail in considering important function characteristics, or may consider features that are not important for the binary similarity problem. In this paper we propose SAFE, a novel architecture for the embedding of functions 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 (i.e., it does not incur in the computational overhead of building or manipulating control flow graphs), and is more general as it works on stripped binaries and on multiple architectures. We report the results from a quantitative and qualitative analysis that show how SAFE provides a noticeable performance improvement with respect to previous solutions. Furthermore, we show how clusters of our embedding vectors are closely related to the semantic of the implemented algorithms, paving the way for further interesting applications (e.g. semantic-based binary function search)

1 Introduction

In the last years there has been an exponential increase in the creation of new contents. As all products, also software is subject 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 spreads in several 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 in 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 own 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* [3, 13, 18], where the goal is to find similar functions in compiled code fragments.

Binary similarity has been recently subject to a lot of attention [9, 10, 14]. This is due to its centrality in several tasks, such as discovery of known vulnerabilities in large collection of software, dispute on copyright matters, analysis and detection of malicious software, etc.

In this paper, in accordance with [16] and [29], 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 [29], this assumption does not make the problem trivial.

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

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

Inspired by [16] we look for solutions that solve the binary similarity problem using *embeddings*. Loosely speaking, each binary function is first transformed into a vector of numbers (an *embedding*), in such a way that code compiled from a same source results in vectors that are similar.

This idea has several advantages. First of all, once the embeddings are computed, checking the similarity is relatively cheap and fast (we consider the scalar product of two constants size vectors as a constant time operation). Thus we can pre-compute a large collection of embeddings for interesting functions and check against such collection in linear time. In the light of the many use cases, this characteristic is extremely useful. Another advantage comes from the fact that such 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, still come with several shortcomings:

- they [29] use manually selected features to calculate the embeddings, introducing potential bias in the resulting vectors. Such bias stems form 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; for example, including features extracted from the function's *control flow graph* (CFG) imposes a $10\times$ speed penalty with respect to features extracted from the disassembled code[3];
- they [12] assume that call symbols to dynamically linking libraries are available in binary functions (such as libc,msvc,ecc..), while this is not true for binaries that are stripped and statically linked[4] or in partial fragments of binaries (e.g. extracted during volatile memory forensic analyses);
- they work only on specific CPU architectures [12].

Considering these shortcomings, in this paper we introduce SAFE: Self-Attentive Function Embeddings, a solution we designed to overcome all of them. In particular we considered two specific goals:

- design a solution to quickly generate embeddings for several hundreds of binaries;
- design a solution that could be applicable the 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 [20]. The idea is to directly consider the sequence of instructions in the binary function and to model them as natural language. An GRU Recurrent Neural Network (GRU RNN) is then used to capture the sequential interaction of the instructions. In addition, an attention mechanism allows the final function embedding to consider all the GRU hidden states, and to automatically focus (i.e., give more weight) to the portion of the binary code that helps the most in accomplishing the training objective - i.e., recognise if two binary functions are similar.

We also investigate the possibility of semantically classifying (i.e. identifying the general sematic behavior of) binary functions by clustering similar embeddings. At 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). Finally, we also applied our semantic classifier to known malwares, and we were able to accurately recognize with it functions implementing encryption algorithms.

1.1 Contribution

The main contributions of our work are:

- we describe SAFE, a general architecture for calculating binary function embeddings starting from disassembled binaries;

[3] Tests conducted using the Radare2 disassembler [25].

[4] Conversely, recognizing library functions in stripped statically linked binaries is an application of the binary similarity problem without symbolic calls.

- we extensively evaluate 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 [29], showing a performance improvement on several metrics that ranges from 6% to 29% depending on the task at hand;
- 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 apply SAFE to the analysis of known malwares, to identify encryption functions. Interestingly, we achieve good performances: among 10 functions flagged by SAFE as *Encryption*, only one was a false positive.

The remainder of this paper is organized as follows. Section 2 discusses related work, followed by Section 3 where we define the problem and report an overview of the solution we tested. In Section 4 we describe in details SAFE, and in Section 5 we provide implementation details and information on the training. In Section 6 we describe the experiments we performed and report their results. Finally, in Section 7 we discuss the speed of SAFE.

2 Related Work

We can broadly divide the binary similarity literature in works that propose embedding-based solutions, and works that do not.

2.1 Works not based on embeddings

Single Platform solutions — Regarding the literature of binary-similarity for a single platform, a family of works is based on matching algorithms for function CFGs. In Bindiff [13] matching among vertices is based on the syntax of code, and it is known to perform poorly across different compiler (see [9]). Pewny et al. [24] 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 on graph matching. David and Yahav [11] proposed to represent a function as several independent execution traces, called *tracelets*; similar tracelets are then matched by using a custom edit-distance. A related concept is used by David et al. in [9] where functions are divided in pieces of independent code, called *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. Note that all previous solutions are 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 [14] proposed a solution where 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 [14] are useful for cross-platform comparison. Finally, Khoo et al. [18] 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.

Table 1: Notation.

s	source code
c	compiler
f^s	binary function compiled from source code s.
$\vec{f}$	embedding vector of function f
I_{f_i}	list of instructions in function f_i
m	number of instructions in a function
ι	instruction in I_{f_i}
$\vec{\iota}$	embedding vector of ι
$\vec{I_{f_i}}$	list of instruction embeddings in function f_i
i2v	instruction embedding model (instruction2vector)
$a[i]$	i-th component of vector a
h_i	i-th hidden state of the Bi-directional Recurrent Neural Network (RNN)
n	dimension of vector $\vec{f_i}$
u	dimension of the state h_i
tanh	hyperbolic tangent function
softmax	softmax function
ReLU	rectified linear unit function
W_{s1}	$d_a \times u$ weights matrix of the attention mechanism
W_{s2}	$r \times d_a$ weights matrix of the attention mechanism
W_{out1}	$e \times (m + u)$ weights matrix of the output layer of the Self-Attentive net.
W_{out2}	$n \times e$ weights matrix of the output layer of the Self-Attentive network
B	$r \times u$ embedding matrix
d_a	attention depth - Parameters of the function embedding network
r	number of attention hops - Parameters of the function embedding network
y_i	ground truth label associated with the i-th input pair of functions
Φ	network hyper parameters
J	network objective function
k	number of results of a function search query

Cross-Platform solutions — Pewny et al. [23] proposed a graph-based methodology, i.e. a matching algorithm on the CFGs of functions. The idea is to transform the binary code in an intermediate representation; on such representation the semantic of each CFG vertex is computed by using a sampling of the code executions using random inputs. Feng et al. [15] 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. Note that both, [23] and [15] allow *pair-to-pair* check only.

David et al. [10] propose to transform binary code to an intermediate representation. Then, functions were partitioned in slices of independent code, called *strands*. An involved process guarantees that strands with the same semantics will have similar representations. Functions are deemed to be similar if they have matching of significant strands. Note that this solution does generate a signature as a collection of hashed strands. However, it has two drawbacks: the first is that the signature is not constant-size but it depends on the number of strands contained in the function. The second drawback is that is not immediate to transform such signatures into embeddings that can be directly fed to other machine learning algorithms.

2.2 Based on embeddings

The most related to our works are the ones that propose embeddings for binary similarity. Specifically, the works that target cross-platform scenarios.

Single-Platform solutions — Recently, [12] proposed a function embedding solution called *Asm2Vec*. This solution is based on the PV-DM model [19] 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. Asm2Vec outperforms several state-of-the-art solutions in the field of binary similarity. Despite being a really

promising solution, Asm2vec does not fullfill all the design goals of our system: firstly it requires libc call symbols to be present in the binary code as tokens to produce the embedding of a function; secondly it is only suitable for single-platform embeddings.

Cross-Platform solutions — Feng et al. [16] introduced a solution that uses a clustering algorithm over a set of functions to obtain centroids for each cluster. Then, they used these centroids and a configurable feature encoding mechanism to associate a numerical vector representation with each function. Xu et al. [29] proposed an architecture called *Gemini*, where function embeddings are computed using a deep neural network. Interestingly, [29] shows that Gemini outperforms [16] 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 [8]. The manual features used by Gemini do not need call symbols. At the best of our knowledge Gemini is the state-of-the-art solution for cross-platform embeddings based on deep neural networks that works on cross-platform code without call symbols. In an unpublished technical report [4] we proposed a variation of Gemini where manual features are replaced with an unsupervised feature learning mechanism. This single change led to a 2% performance improvement over the baseline represented by Gemini.

Finally, in [30] the author propose the use of a recurrent neural network based on LSTM (Long short-term memory) to solve a subtask of binary similarity that is the one of finding similar CFG blocks.

3 Problem Definition and Solution Overview

For clarity of exposition we summarize all the notation used in this paper in Table 1. Let us first define the similarity problem.

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. 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.

Function Semantic. Loosely speaking, a function f can be seen as an implementation of an algorithm. We can partitions algorithms in *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.

3.1 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

5.1), 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 [20]. In this Self-Attentive 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 [20]. Secondly, it suffers less of the long-memory problem[5] 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 6).

4 Details of the SAFE, Function Embedding Network

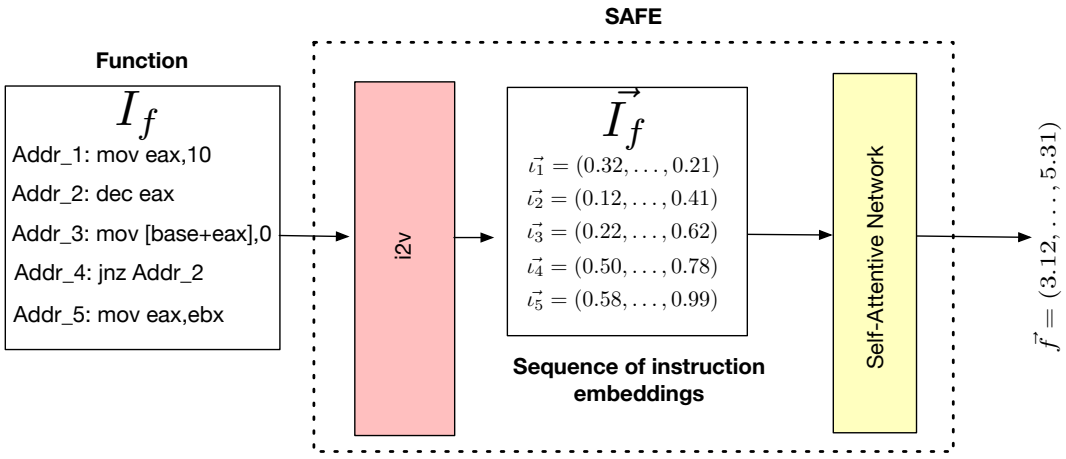


Figure 1: Architecture of SAFE. The vertex feature extractor component refers to the Unsupervised Feature Learning case.

We denote the entire embedding network as SAFE: Self-Attentive Function Embeddings.

4.1 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 [7].

We train the i2v model using assembly instructions as tokens (i.e., a single token includes both the instruction mnemonic and the operands). We do not use the raw instruction but we filter it as follows. We examine the operands and 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 5.1) with the special symbol `IMM`. We do this filtering because we believe that using raw operands is of small benefit; for instance, the 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`

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

`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.

4.2 Self-Attentive Network

We based our Self-Attentive Network on the one proposed by [20]. The overall structure is detailed in 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 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.

Intuitively, when $r = 1$, A collapses into a single attention vector, where each each value is the weight a specific summary vector. When $r > 1$, A becomes a matrix and each row is an independent attention hop. Loosely speaking, each hops weights the attention of a different aspect of the binary function.

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 flattening 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 [5]. 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 (see Figure 3).

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:

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

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

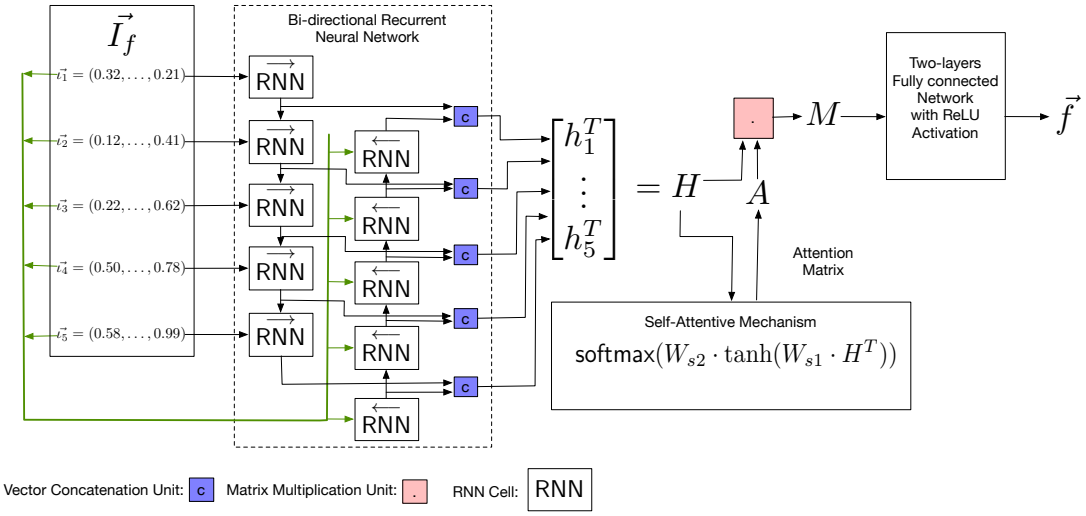


Figure 2: Self-Attentive Network: detailed architecture.

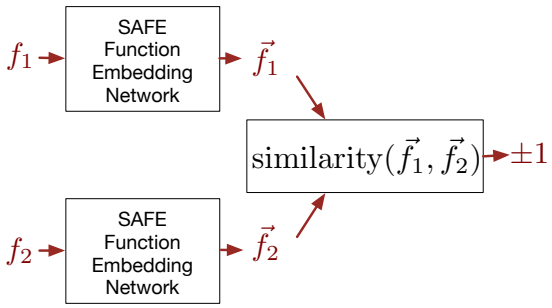


Figure 3: Siamese network.

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:

$$J = \sum_{i=1}^{K} \Big(\text{similarity}(\vec{f}_1, \vec{f}_2) - y_i\Big)^2 + \|(A \cdot A^T - I)\|_F$$

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 [20]).

5 Implementation Details and Training

In this section we first discuss the implementation details of our system, and then we explain how we trained our models.

5.1 Implementation Details and i2v setup

We developed a prototype implementation of SAFE using Python and the Tensorflow [1] framework. For static analysis of binaries we used the ANGR framework [27], radare2 [25][6] and IDA Pro. To train the network we used a batch size of 250, learning rate 0.001, Adam optimizer.

[6] We build our system to be compatible with two opensource leader disassemblers.

In our SAFE prototype we used the following parameters: the RNN cell is the GRU cell [6]; 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).

5.1.1 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 sintaxes 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 [28]. We used as parameters: embedding size 100, window size 8 and word frequency 8.

Datasets for training the i2v models We collected the assembly code of a large number of functions, and we used it to build two training corpora, one for the i2v AMD64 model and one for the i2v ARM model. We built both corpora by dissasembling 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 86 millions assembly code lines while the ARM corpus contains 104 millions assembly code lines.

5.2 Training Single and Cross Platform Models

We trained SAFE models using the same methodology of Gemini, see [29]. We trained both a single and a cross platform models that were then evaluated in several tasks (see Section 6 for the results).

5.2.1 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 [29]. 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.

5.2.2 Training

We generate our training and test pairs as reported in [29]. 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 pairs that is twice the total number of functions.

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

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 6.1 we will perform a cross-validation.

The test and validation pairs will be used to assess performances in Task 1, see Section 6.1.

As in [29], 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.

6 Evaluation

We perform an extensive evaluation of SAFE investigating its performances on several tasks:

- **Task 1 - Single Platform and Cross Platform Models Tests**: we test our single platform and cross platform models following the same methodology of [29]. 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). This task is described in Section 6.1.
- **Task 2 - Function Search**: in this task we are given a certain binary function and we have to search for similes 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. Section 6.2 is devoted to Task 2.
- **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 is the focus of Section 6.3.
- **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 malwares, showing that we can identify encryption functions. Task 4 is explained in Section 6.4.

During our evaluation we compare safe with Gemini [8]

6.1 Task 1 - Single and Cross Platform tests

In this task we evaluate the performance of SAFE using the same testing methodology of Gemini. We use the test split and the validation split computed as discussed in Section 5.2.1.

6.1.1 Test methodology

We perform two disjoint tests.

- On AMD64multipleCompilers Dataset, 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.

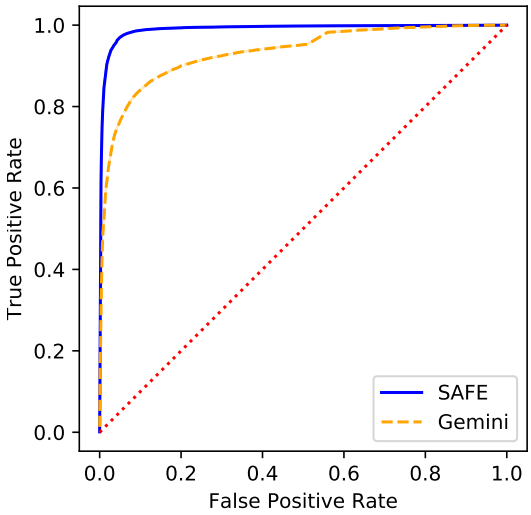
[8] Gemini has not been distributed publicly. We implemented it using the information contained in [29]. For Gemini the parameters are: function embeddings of dimension 64, number of rounds 2, and a number of layers 2. These parameters are the ones that give the better performance for Gemini, according to our experiments and the one in the original Gemini paper.

- On 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.

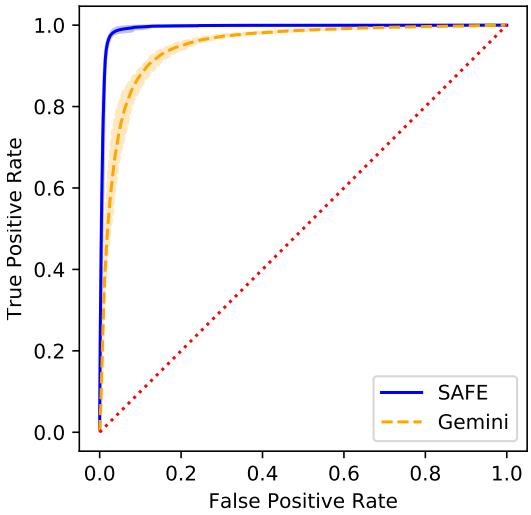
Measures of Performances As in [29], we measure the performance using the *Receiver Operating Characteristic* (ROC) curve [17]. Following the best practices of the field we measure the *area under the ROC curve*, or AUC (Area Under Curve). Loosely speaking, higher the AUC value, better the predictive performance of the algorithm.

6.1.2 Experimental Results

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. Even if the improvement is 6.8, it is worth to notice that SAFE provides performance that are close to the perfect case (0.99 AUC).



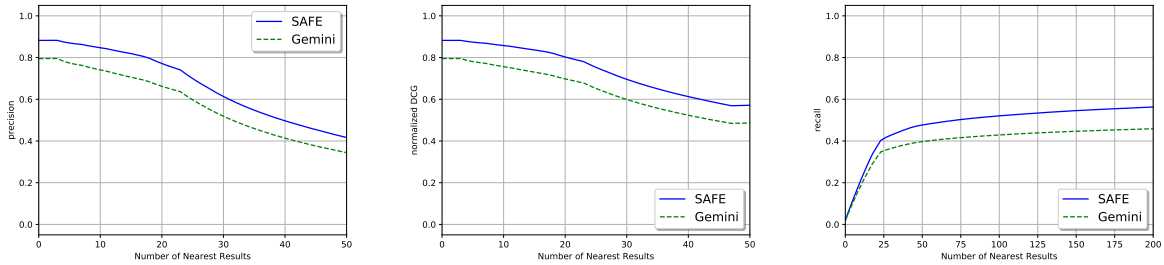
(a) Test on AMD64multipleCompilers Dataset. ROC curves for the comparison between SAFE and Gemini on the test set. The dashed line is the ROC for Gemini, the continuous line the ROC for SAFE. The AUC of our solution is $0,990$ the AUC of Gemini is $0,932$



(b) Test on AMD64ARMOpenSSL Dataset. ROC curves for the comparison between SAFE and Gemini 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 Gemini, the continuous line the average for our solutions. For both we color the area between the ROC curves with minimum AUC and the maximum AUC. The average AUC of our solution is $0,992$ the average AUC of Gemini is $0,948$

Figure 4: ROC curves on AMD64multipleCompilers Dataset and AMD64ARMOpenSSL Dataset. Task 1-Validation and Test of Single Platform and Cross Platform Models

AMD64ARMOpenSSL Dataset We compare ourselves with Gemini in the cross-platform case. The results are in Figure 4b and they shows the average ROC curves on the five runs of the 5-fold cross validation. The Gemini results are reported with an 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 $0,006$ over the five runs, 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%.



(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 $k \leq 200$.

Figure 5: Results for Task 3 -Function Search, on AMD64PostgreSQL Dataset (581K functions) average on 160K queries.

6.2 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 AMD64PostgreSQL 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 AMD64PostgreSQL Dataset.

6.2.1 Dataset

We built AMD64PostgreSQL Dataset by compiling postgreSQL 9.6.0 for AMD64 using 12 compilers: 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. 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 AMD64PostgreSQL Dataset contains an average number of 33 similars. We do not reach an average of 48[9] similars because some functions are lost due to disassembler errors.

6.2.2 Measures of Performances

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)* [2]:

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

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.

6.2.3 Experimental Results

Our results on precision, nDCG and recall are reported in Figure 5.

[9] 48= 12 compilers × 4 optimizations level

The performances were calculated by averaging the results of 160K queries. The queries are obtained by sampling, in AMD64PostgreSQL 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}.

Let us recall that, on average, for each query we have 33 similar functions (e.g., functions compiled from the same source code) in the dataset.

- Precision: The results are reported in Figure 5a. The precision is above 80% for $k \in [0, 15]$, and it is above 60% for $k \in [0, 30]$. 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 $\{84\%, 77\%, 61\%, 49\%, 41\%\}$ for SAFE and $\{74\%, 66\%, 51\%, 41\%, 34\%\}$ for Gemini.
- nDCG: The tests are reported in Figure 5b. Our solution has a performance above 80% for $k \in [0, 18]$. 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 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 $\{85\%, 80\%, 69\%, 61\%, 57\%, 59\%, 62\%\}$ for SAFE and $\{75\%, 69\%, 59\%, 52\%, 48\%, 50\%, 52\%\}$ for Gemini.
- Recall: The tests are reported in Figure 5c. We have a recall at $k = 50$ of 47% (vs. 39% 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 $\{21\%, 36\%, 42\%, 45\%, 47\%, 52\%, 56\%\}$ for SAFE and $\{18\%, 31\%, 36\%, 38\%, 39\%, 42\%, 45\%\}$ for Gemini.

6.3 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.

6.3.1 Dataset and methodology

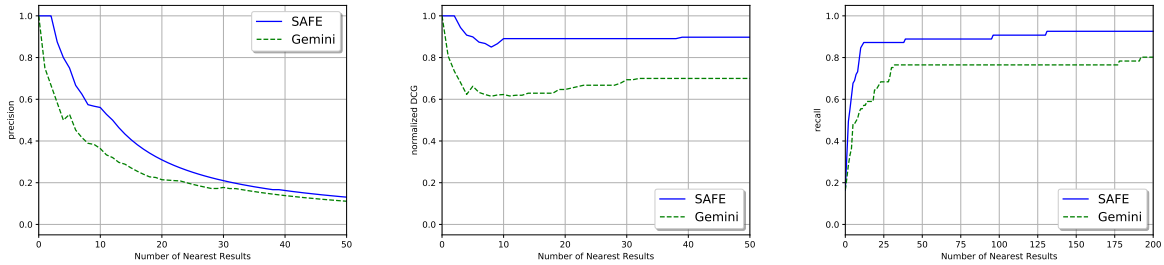
The dataset used is the vulnerability dataset of [9]. The dataset contains several vulnerable binaries compiled with 11 compilers in the families of clang, gcc and icc. The total number of different vulnerabilities is 8[10]. We disassembled the dataset with ANGR, obtaining 3160 binary functions. The 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[11]. We performed a lookup for each of the 8 vulnerabilities, computing the precision, nDCG, and recall on each result. Finally, we averaged these performances over the 8 queries.

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

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

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



(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 $k \leq 200$.

Figure 6: Results for Task 3 - Vulnerability Search.

Table 2: 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

6.4 Task 4 - Semantic Classification

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.

6.4.1 Dataset

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 AMD64PostgreSQL Dataset, we took the object files and after disassembling them with ANGR we obtained a total of 15158 binary functions, see details in Table 2. 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.

6.4.2 Measures of Performances

As performance measures we use precision, recall and F-1 score.

6.4.3 Experimental Results

The results of our semantic classification tests are reported in Table 3. 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

Table 3: 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	Embedding Model	Precision	Recall	F1-Score
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

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%.

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 represent 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-dimensional projection of the 100-dimensional vector space where binary functions embeddings lie, obtained using the *t-SNE* [12] visualisation technique [21]. 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 Malwares We decided to test the semantic classifcation 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 on malwares. We analyzed two samples of window malwares found in famous malware repositories: the *TeslaCrypt* and *Vipasana* ransomwares. We disassembled the malwares 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* [13]: 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.

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

[13] The sample is available at url `https://github.com/ytisf/theZoo/tree/master/malwares/Binaries/Ransomware.TeslaCrypt`. The variant analyzed is the one with hash *3372c1eda...*

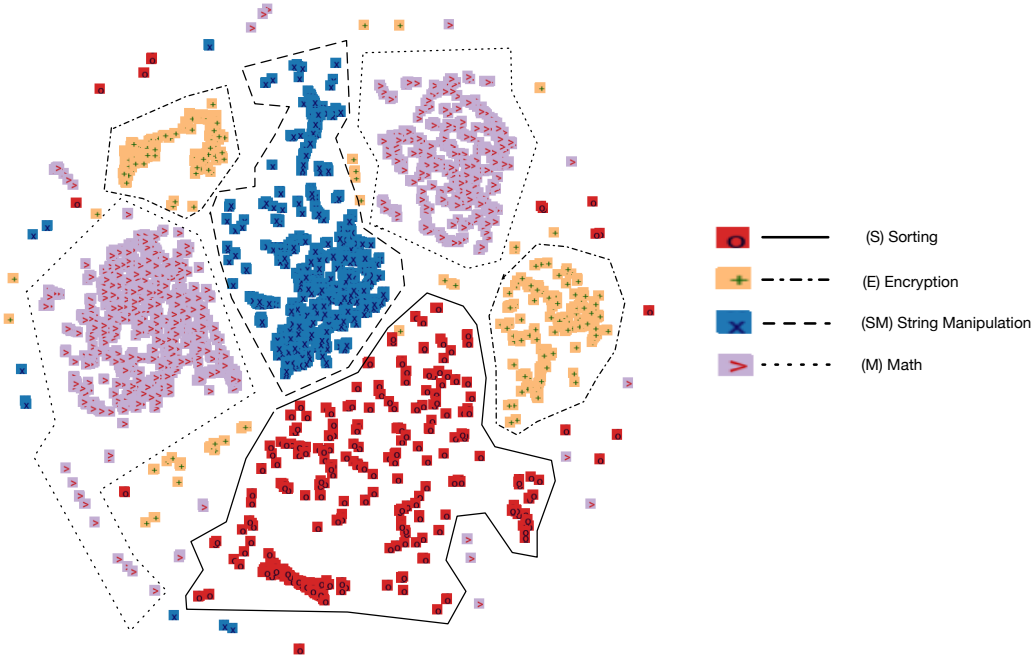


Figure 7: 2-dimensional visualization of the embedding vectors for all binary functions in Semantic Dataset. The four different categories of algorithms (Encryption, Sorting, Math and String Manipulation) are represented with different symbols and colors.

- *Vipasana* [14]: 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 malwares 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.

7 Speed considerations.

As reported in the introduction, one of the advantages of SAFE that it ditches the use of CFGs. From our tests on radare2 disassembling a function is 10 times faster than computing 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. During the disassembling we assume to know the starting address of a function, see [26] for a paper using neural networks to find functions in a binary.

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.

Gemini is up to 10 times slower, it needs 43 minutes for postgres and 26 minutes for openssl.

[14] The sample is available at url `https://github.com/ytisf/theZoo/tree/master/malwares/Binaries/Ransomware.Vipasana`. The variant analyzed is the one with hash *0442cfabb...4b6ab*

8 Conclusions and future works

In this paper we introduced SAFE an architecture for computing embeddings of functions in the cross-platform case that does not use debug symbols. Our architecture does not need the CFG, and this leads to a considerable speed advantage. SAFE creates thousand embeddings per second on a mid-CTOS GPU. Even when we factor the disassembling time our end-to-end system (from binary file to function embedding), processes more than 100 functions per second. This considerable speed comes with a significant increase of predictive performances with respect to the state of the art. Summing up, SAFE is both faster and more precise than previous solutions.

Finally, we think that our experiments on semantic detection are really interesting, and they pave the way to more complex and refined analysis, with the final purpose of building binary classifiers that rival with the classifiers today available for image recognition.

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

The use of libc symbols would enable a more fine grained semantic classification: as example we could be able to distinguish a function that is sending encrypted content on a socket from a function that is writing encrypted content on a file. Summarizing, the field of applied machine learning for binary analysis is still in its infancy and there are several opportunities for future works.

Acknowledgments. The authors would like to thank Google for providing free access to its cloud computing platform through the Education Program. Moreover, the authors would like to thank NVIDIA for partially supporting this work through the donation of a GPGPU card used during prototype development. Finally, the authors would like to thank Davide Italiano for the insightful discussions. This work is supported by a grant of the Italian Presidency of the Council of Ministers and by the CINI (Consorzio Interuniversitario Nazionale Informatica) National Laboratory of Cyber Security.

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] 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.

[3] 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.

[4] R. Baldoni, G. A. Di Luna, L. Massarelli, F. Petroni, and L. Querzoni, "Unsupervised features extraction for binary similarity using graph embedding neural networks," in *arXiv preprint arXiv:1810.09683*, 2018.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[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] 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.

[24] 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.

[25] Radare2 Team, "radare2 disassembler repository," https://github.com/radare/radare2, 2017.

[26] 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, (USENIX Security)*, 2015, pp. 611–626.

[27] 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.

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

[29] 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.

[30] 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.