Investigating Graph Embedding Methods for Cross-Platform Binary Code Similarity Detection

Victor Cochard
Cyber-Defence Campus
armasuisse S+T
Lausanne, Switzerland

Damian Pfammatter
Cyber-Defence Campus
armasuisse S+T
Zurich, Switzerland

Chi Thang Duong
DISL
EPFL
Lausanne, Switzerland

Mathias Humbert
DESI
University of Lausanne
Lausanne, Switzerland

***Abstract*—IoT devices are increasingly present, both in the industry and in consumer markets, but their security remains weak, which leads to an unprecedented number of attacks against them. In order to reduce the attack surface, one approach is to analyze the binary code of these devices to early detect whether they contain potential security vulnerabilities. More specifically, knowing some vulnerable function, we can determine whether the firmware of an IoT device contains some security flaw by searching for this function. However, searching for similar vulnerable functions is in general challenging due to the fact that the source code is often not openly available and that it can be compiled for different architectures, using different compilers and compilation settings. In order to handle these varying settings, we can compare the similarity between the graph embeddings derived from the binary functions. In this paper, inspired by the recent advances in deep learning, we propose a new method – GESS (graph embeddings for similarity search) – to derive graph embeddings, and we compare it with various state-of-the-art methods. Our empirical evaluation shows that GESS reaches an AUC of 0.979, thereby outperforming the best known approach. Furthermore, for a fixed low false positive rate, GESS provides a true positive rate (or recall) about 36% higher than the best previous approach. Finally, for a large search space, GESS provides a recall between 50% and 60% higher than the best previous approach.**

1. Introduction

Devices connected to the Internet, known as Internet of Things (IoT), are increasingly present in our daily lives. Cameras, refrigerators, smart speakers, connected sensors, lightning fixtures, and wearable devices are some examples thereof. IoT devices might have access to confidential information (e.g., health devices or security cameras) or play a crucial role for safety (e.g., door controllers). However, security is often not the main priority when designing such devices, and the use of untrusted third-party code, frequently even in outdated versions, is a common practice (known as *code reuse*). Moreover, since these devices often rely upon cheap components with limited memory and computing resources, classical protection techniques (such as antivirus software) are typically too heavy-weight, and in consequence not being used [1]. As a consequence, hackers increasingly target IoT devices, thereby creating serious damages and putting human lives at risk [2], [3]. In order to reduce the attack surface and prevent potential cyberattacks, it becomes crucial to early identify vulnerabilities in IoT devices' firmwares.

Unfortunately, identifying vulnerabilities is a difficult task, especially in the situation where firmwares are closed-source. Even finding known security vulnerabilities across devices poses a major challenge. This is due to the fact that the same piece of (vulnerable) source code can be compiled for various architectures, using different compilers and compilation settings (reflecting varying objectives such as runtime efficiency or binary code size), thus leading to strongly differing code representations in its binary form. Hence, having a mechanism to compare binary functions, and assigning a measure of similarity in terms of source code equivalence, is of high added value. With such a similarity detection technique, known vulnerabilities (e.g., in a popular third-party library) can be searched across various *closed-source firmware* images.

Previous studies have shown that search techniques based upon control flow graphs (CFGs) can be effective and accurate to find security vulnerabilities across different architectures, and generally outperform approaches that directly work on binary instructions [4]. However, searches based upon raw CFGs hardly scale, due to the computational cost of graph matching. One approach to improve scalability is to map attributed control flow graphs (ACFGs) into high-dimensional numeric vectors referred to as *graph embeddings* [5], [6]. Graph embeddings have not only shown to provide better search-time capabilities than graph matching, but have also led to promising results in terms of robustness to code variations across different architectures and compiler settings.

In order to better understand how graph embeddings can help detect security vulnerabilities, consider the workflow depicted in Figure 1. Initially, two databases (firmware and vulnerability) are built in an offline preparation phase. This is done by taking a set of available firmware images, respectively vulnerabilities, extracting the corresponding ACFGs and storing the resulting graph embeddings in the corresponding databases. Once these databases are built, two approaches become possible:

- Vulnerability search: In case a new vulnerability becomes known, the corresponding embedding can be searched in the firmware database, returning candidate devices that might be affected by the requested vulnerability.
- Firmware search: In case a new device catches interest (e.g., during procurement), its embedding

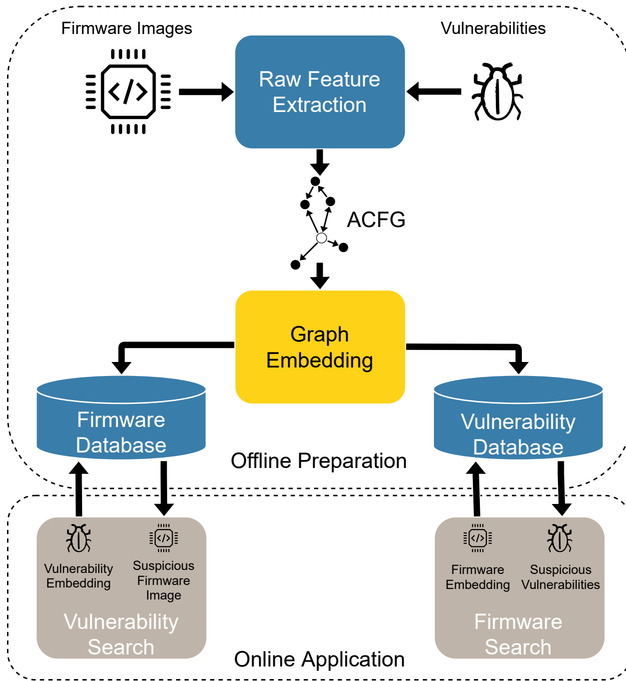


Figure 1: Graph embeddings similarity search to identify security vulnerabilities in firmware images. Figure adapted from [6].

can be computed and compared against the vulnerability database, to get a list of potential vulnerabilities the device might contain.

Existing approaches require to trade-off recall and precision when trying to detect similar binary functions in real-world settings where the search space is large and the number of dissimilar pairs of functions largely outweigh the number of similar pairs. Therefore, one key challenge that remains when searching for vulnerable functions in large firmware databases is to achieve both a high recall and a high precision in order to quickly identify vulnerable functions and limit the need of expert knowledge.

Contributions. In this work, we propose *GESS*, a new approach for graph embedding based on neural networks. It works in two steps. The first step transforms an arbitrary ACFG into a specific fixed-size numeric vector. It does so by selecting a fixed number of representative nodes from the ACFG and, for each of these nodes, computing their fixed-size neighborhood. The numeric vector then consists of the attributes of the nodes' neighbors. The second step of GESS abstracts away compilation settings to transform this numeric vector into a representation that is robust to cross-platform differences. To this end, it uses a specifically-crafted convolutional neural network trained with hundreds of thousands of ACFGs from different libraries. We compare GESS to the best existing approach, Gemini [6], and to a new backward message passing method. We also adapt popular graph neural networks (GNNs), specializing them to the considered problem. These include GraphSAGE [7], graph convolutional networks [8], and graph isomorphism networks [9], nowadays widely used for graph-related problems [10]–[13].

We evaluate the performance of GESS and alternative approaches by using two open-source software components widely used in IoT devices: *OpenSSL* and *binutils*. GESS provides an area under the ROC curve (AUC) of 0:979, whereas the best alternative approach, Gemini reaches an AUC of 0:949 on the same dataset. Furthermore, at a false positive rate of 0.01, GESS provides a true positive rate (TPR) of 0.76 against a true positive rate of 0.56 for Gemini, representing a relative TPR increase of 35.7%. Moreover, GESS generalizes better to completely unalike functions, such as functions from distinct libraries. For example, when training the models on binary functions coming from the OpenSSL dataset and testing on binutils functions, GESS provides an AUC of 0:931 whereas Gemini provides an AUC of 0:898.

The improvement provided by GESS is especially striking in the most realistic setting where there is an imbalance between the number of similar and dissimilar functions and a large search space. When retrieving the 20 nearest neighbors of a binary function, Gemini provides a recall of 37% whereas GESS reaches a recall of 60%, representing an increase of more than 60%. Besides, in order to reach a recall of 50%, GESS requires only the 10 nearest binary functions while Gemini requires 116 nearest neighbors. This in turn dramatically affects the precision of Gemini, which drops to less than 3% (against 32% for GESS). This implies that, in order to identify actual vulnerable functions, a security analyst would have to analyze ten times more binary functions with Gemini than with GESS due to the false positive cases.

Finally, GESS shows comparable or better results than Gemini in terms of time performance. For the training phase, it can reach the same AUC as Gemini in 40 minutes instead of 5 hours for Gemini. Besides, although the full training process takes about 14 hours for GESS, this method already reaches an AUC close to its maximum value in 5 hours (AUC = 0.977). Moreover, GESS is more than four times faster than Gemini for the testing phase.

We summarize our contributions as follows:

- We create a dataset of attributed control flow graphs corresponding to functions widely used in IoT devices, compiled for different architectures with various compilers settings. This dataset can be used to train and compare different binary code similarity detection techniques.
- We propose a new approach based on convolutional neural networks to generate graph embeddings for binary functions. Our evaluation shows that GESS can achieve better AUC and recall than the state-of-the-art approach.
- Our experiments show that GESS can be trained to match Gemini's performance on our dataset 8 times faster.
- We further evaluate several other graph embeddings methods such as graph neural networks and compare their performance with GESS.
- For reproducibility purposes, both our dataset and implementation are made publicly available.[1]

We introduce the concepts that serve as a basis for this work in Section 2. From there, we describe our two new methods in Section 3. We present our experimental results and comparison between the different methods in

1. Available at: https://github.com/GESS-code/GESS

```c
int iterative_fibonacci(int index) {
    int i, current=1, pred=1;
    if(index <=0) {
        return 0;
    }
    if(index <=2) {
        return 1;
    }
    for(i=2; i<index; i+=1) {
        current=current+pred;
        pred=current - pred;
    }
    return current;
}
```

Figure 2: C function computing a specific value of the Fibonacci sequence.

Section 4. We discuss the existing approaches for binary code similarity detection and other related work in Section 5, before concluding in Section 6.

2. Background

In this section, we provide some background on fundamental concepts the remainder of this paper relies on.

2.1. Control Flow Graphs

Control flow graphs (*CFGs*) are a well-known concept used as the foundation of many (static and dynamic) binary analysis techniques. CFGs model the different flows the execution of a program or function can take. CFGs are represented by directed graphs, where vertices correspond to basic code blocks, and edges to possible control transfers between them. A *basic code block* is a straight-line and branch-free sequence of code. Control therefore always enters at the first instruction of a basic block and exits at its last one.

Branches in binary code can either be conditional or always taken (i.e., unconditional). In the case of an unconditional branch, the corresponding basic block has a single successor, whereas for a conditional one, two successors are possible. As a consequences, vertices in CFGs have an out-degree of at most two. CFGs, within the scope of this work, are statically extracted, i.e., without actually running the corresponding binary files. Further, the CFGs are obtained on a per-function basis (*function-local CFGs*). Therefore, they model possible execution flows within a single function.

2.2. Attributed Control Flow Graphs

The *attributed control flow graph* (*ACFG*) of a function is its control flow graph with some attributes added to the vertices. These attributes describe relevant statistics of the basic code block they belong to (e.g., the total number of instructions contained within the corresponding basic block).

As an example, consider the source code depicted in Figure 2, which corresponds to a function computing a value of the well-known Fibonacci sequence. When extracted from a binary representation, the same source-level function might lead to different ACFGs, depending on how the binary was built. Even for simple functions, both the graph structure and node attributes may vary. This is visualized in Figure 3, where ACFGs for four different combinations of *architecture*, *compiler* and compiler *optimization* level are shown (all belonging to the Fibonacci function in Figure 2). This poses a major challenge for binary code similarity detection, where the goal is to determine that all four ACFGs originate from the same source-level function.

The three node attributes listed in Figure 3, however, only represent a subset of the actual attributes being used within this work. In our implementation, we use the reverse engineering framework *Radare2*, version 5.1.0 [14], for ACFG extraction (although any other tool with the capability to extract CFGs might be considered), and employ the following nine attributes for each basic block:

1) Number of *arithmetic* instructions
2) Number of *logic* instructions
3) Number of *shift* instructions
4) Number of *data manipulation* instructions
5) Number of *call* instructions
6) Number of *jump* instructions
7) Number of *other* selected instructions
8) Number of *references* to data or code
9) Number of *total* instructions

The attribute vector of a basic block containing two shift and one jump instruction (thus three instructions in total) would therefore be:

$$0 \quad 0 \quad 2 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 3$$

The attributes used in our work are both influenced by what was proposed in previous work (such as in [15], [5] and [6]) and by the capabilities provided by Radare2 [14].

2.3. Graph Neural Networks

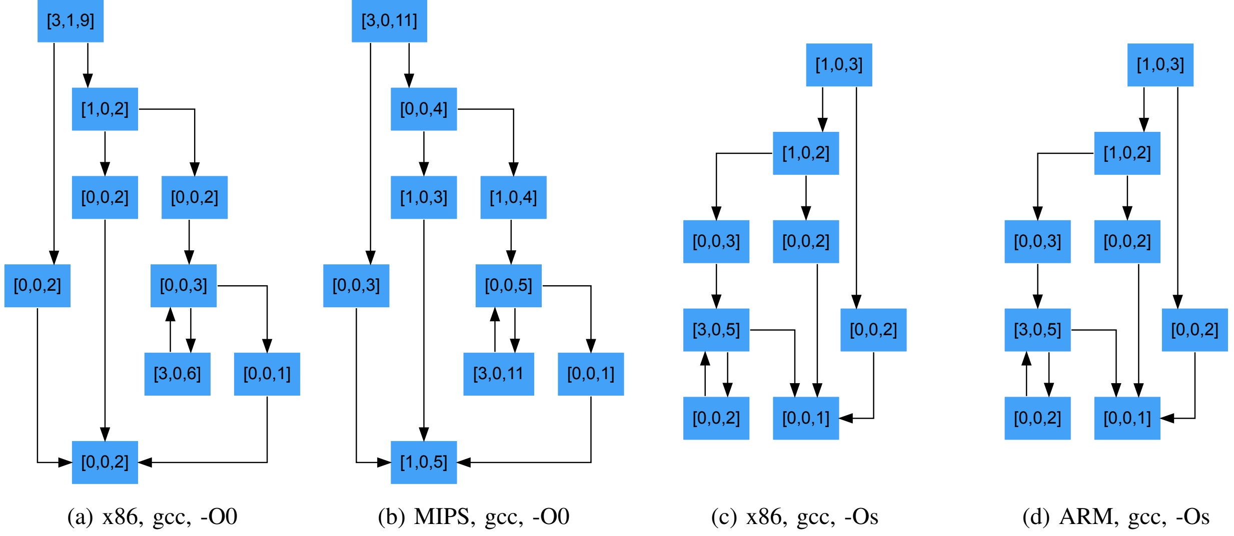
One method for obtaining graph embeddings is to rely on graph neural networks (GNNs). A GNN essentially computes node embeddings by passing messages through the network edges. Then, graph embeddings can be computed by pooling the node embeddings together (see Section 2.4 for details).

General Framework. At the beginning, a node's embedding is merely its attributes. After this initialization phase, the message passing can take place. First, nodes use their embeddings to compute a message that they will pass to their immediate neighbors. Then, nodes take the messages from their neighbors and aggregate them. Finally, a node updates its embedding based on this aggregation and its current embedding. This message passing phase can be repeated for several iterations. After n iterations, we consider that a node's embedding contains information about its neighbors that are at most n hops away. Figure 5 shows how the embedding of node A from Figure 4 is updated from its neighbors' embeddings in one iteration of the algorithm.

Formally, a node u constructs a message in an i-th iteration using a parameterized function f_m:

$$m_{u \to v}^{(i)} = f_m(z_u^{(i)}; c_v)$$

where $z_u^{(i)}$ is the embedding of node u and c_v is the information from node v, which could be empty.



(a) x86, gcc, -O0 (b) MIPS, gcc, -O0 (c) x86, gcc, -Os (d) ARM, gcc, -Os

Figure 3: Four ACFGs representing the function depicted in Figure 2. Each node contains three attributes representing the number of *arithmetic*, *call* and *total* instructions in the corresponding basic code block. Each subfigure represents a binary function (or ACFG) compiled with a different combination of *architecture*, *compiler* and compiler *optimization* level.

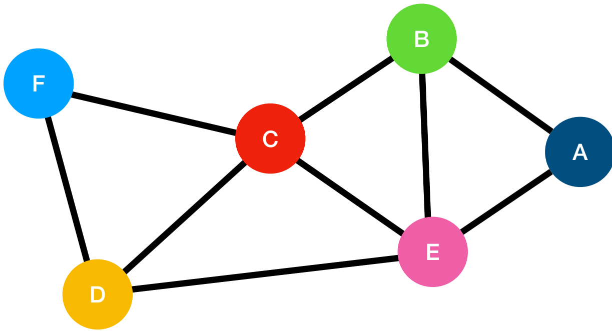


Figure 4: A sample undirected graph, serving as basis for Figure 5 and Figure 6.

Once a node v receives messages from all of its neighbors, it aggregates them using an aggregation function f_a:

$$z^{(i)}_{N(v)} = f_a(\{m^{(i)}_{u\to v}, \forall u \in N(v)\})$$

A node v updates its embedding with the update function f_u as follows:

$$z^{(i+1)}_v = f_u(z^{(i)}_{N(v)}, z^{(i)}_v)$$

There exist various GNNs that essentially differ in the sent messages and aggregation functions. In this paper, we will rely on and compare three popular GNNs.

Graph Convolutional Network. In a GCN [8], a node's message sent to its neighbors is the embedding of the sending node divided by the square root of the degrees (i.e., the number of neighbors) of sending and receiving nodes multiplied together. Formally, the message function f_m from node a to node b is $f_m(a, b) = \frac{z_a}{\sqrt{d_a \cdot d_b}}$, where d_n is the degree of node n. The aggregation function f_a is a sum.

GraphSAGE. In GraphSAGE [7], the message function f_m returns a node's embedding, and the aggregation

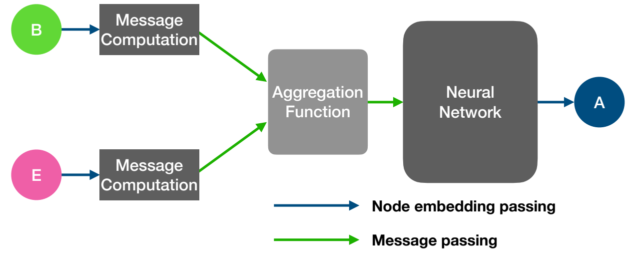


Figure 5: Message passing to update the embedding of node A from Figure 4. The neural network is a multi-layer perceptron (MLP) and the message computation and aggregation function depend on the type of GNN used.

function f_a is either a sum or a mean of the neighbor embeddings, while the update function f_u is a concatenation of the neighbors' embeddings with the node's own embedding. For example, if sum is chosen as the concatenation operation, node n updates its embedding by feeding $z_n \| \sum_{u \in N(n)} z_u$ to the neural network (as shown in Figure 5), where $\|$ is the concatenation operator.

Graph Isomorphism Network. A GIN [9] is a GNN that is used to compute graph-global embeddings that discriminate non-isomorphic graphs. Nodes directly send their embedding as message, and the aggregation function is a sum. After each iteration, the embeddings of each node in the graph are summed, and this sum is stored. This sum is used as the graph embedding.

2.4. From Node to Graph Embeddings

Global Pooling. Apart from GINs which are designed to embed graphs, the other GNN techniques only compute node embeddings. We can however use these node embeddings and create a graph embedding from them by pooling *all* of them together using a pooling function. Two traditional pooling functions are global sum pooling and global mean pooling where the graph embedding is computed by

summing, respectively averaging, the node embeddings. For example, given the set of node embeddings $\{z_v\}$ of a graph G, the graph embedding constructed using global mean pooling is:

$$z_G = \frac{1}{|G|} \sum_{v \in G} z_v$$

where $|G|$ is the number of nodes.

Differential Pooling. Global pooling in GNNs leads to a loss of information about node embeddings, as it is impossible to reconstruct the individual node embeddings from the result of a global pooling function that is not injective. Intuitively, the information loss gets more important as more nodes are pooled together.

In order to get graph embeddings for GCN and GraphSAGE without pooling all nodes together, we can use differential pooling [16]. Instead of pooling all the nodes in a graph, differential pooling divides it into subgraphs, using another GNN that computes the partitions in parallel, and performs the pooling by subgraph. The pooling result is considered as the embedding of the subgraph. Since the subgraphs are smaller, less information is lost than if global pooling had to be performed. Then, a new smaller graph is output, each of its nodes representing a subgraph of the original graph.

In differential pooling, the whole process (message passing + subgraph pooling) is repeated until only one node is left. The graph embedding is then the embedding of the remaining node. Differential pooling uses weighted edges to consider the fact that some subgraphs are more interconnected than others (depending on the number of interconnected nodes). The assignments are computed using soft clustering. Hence, a node's clustering assignment is the probability for the node to be in each cluster. A node can therefore partly be in multiple clusters at the same time (with the sum of the clustering probabilities being equal to 1).

3. Methodology

In this section, we describe the technical details of the two new approaches we propose.

3.1. Graph Embeddings for Similarity Search (GESS)

We first explain the details of GESS, our first novel graph embedding method. GESS starts by labeling the nodes of the ACFGs based on their attributes (as defined in Section 2.2). We define the node label as the linear combination of its attributes. Formally, a node with 9 attributes $a_{i;1\leq i\leq 9}$ takes as label:

$$\sum_{i=1}^{9} a_i p_i$$

where $p_{i;1\leq i\leq 9}$ are the weights of the attributes.

Then, by building upon Patchy-San [17], we transform the labeled graphs into receptive fields in order to later feed them to a convolutional neural network (CNN).

A CNN is a neural network that uses convolutional layers in addition to dense layers [18]. A convolutional layer divides its inputs into small – possibly overlapping – chunks. A chunk can be a slice from a larger image or a 1-second part of an audio sample for example. These chunks are then fed to convolutional filters, which extract higher-level features. The same filters are used for each chunk. A CNN often stacks convolutional layers. The idea is to first compute local features on small slices and progressively use them to get relevant features on larger slices. A CNN often ends with one or more dense layers which output global features on the whole data.

Generally, arbitrary graphs cannot be easily divided into relevant chunks because of their lack of structure (unlike an audio sample or a 2D image for example). In order to embed a graph, we select a fixed number of nodes and return a fixed-size neighborhood of those nodes. These neighborhoods are called the graph's receptive fields and are then fed to the convolutional neural network (as graph "chunks"), that returns the embeddings. The different steps of the construction of a neighborhood are shown in Figure 6.

We use the node labels to sort them and select those that will generate the receptive fields: The first receptive field will be the neighborhood of the node with the highest label, and so on. The neighbors are returned sorted: First comes the source node, i.e., the node used to create the neighborhood, and then the neighbors are sorted by distance to source. If some neighbors have the same distance to the source, those with the highest label will come first. For example, in Figure 6, node A is selected first, as it is the source node. Then come nodes B and E – sorted by label (in this example, alphabetical order) – since they are one-hop away from A. Finally come C and D since they are two-hop away from A. Only C is eventually included in the receptive field since the receptive field is of size 4 and can thus include only 4 nodes.

Once the receptive fields have been generated, we feed them into a CNN with the same structure as the one depicted in Figure 7. The first convolutional layer takes the receptive fields as chunks and extracts, for each of them, local features. Then, the second layer takes each of these features as chunks and, for each of them, extracts graph-global features. For example, in Figure 7, it uses the second local feature in order to extract three different graph-global features represented in red, green and blue. A rectified linear unit (ReLU) activation function is applied to these features. It replaces negative values by 0 and leaves positive values unchanged. This allows the network to be non-linear and makes it more expressive. Finally, a dense layer with ReLU activation returns the graph embedding.

In summary, GESS computes graph embeddings by:

1) Computing labels for each node from their attributes using a linear filter;
2) Selecting highest-labeled nodes;
3) Creating receptive fields around these nodes;
4) Ordering the nodes by distance and then by label in each receptive field;
5) Feeding the receptive fields to a CNN.

3.2. Backward Message Passing (BMP)

We now introduce a new technique that takes advantage of the following two properties of ACFGs:

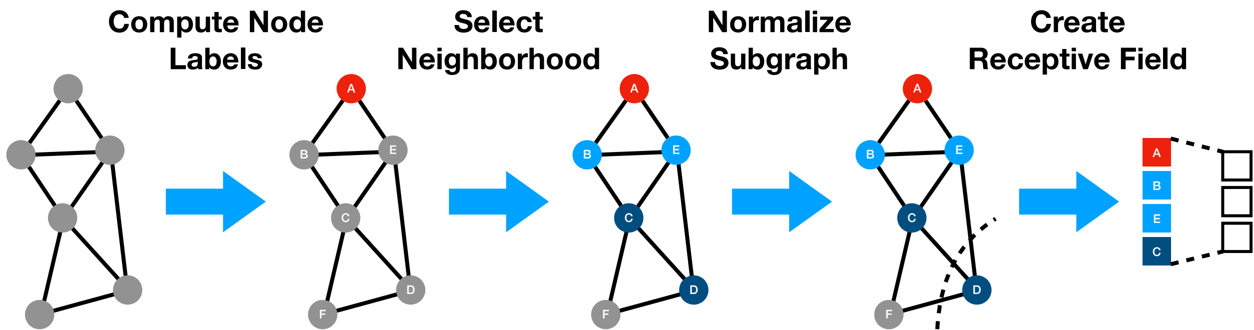


Figure 6: Overview of GESS (first part): Labeling the nodes and creating the receptive field(s) that will be fed into the convolutional neural network (see Figure 7).

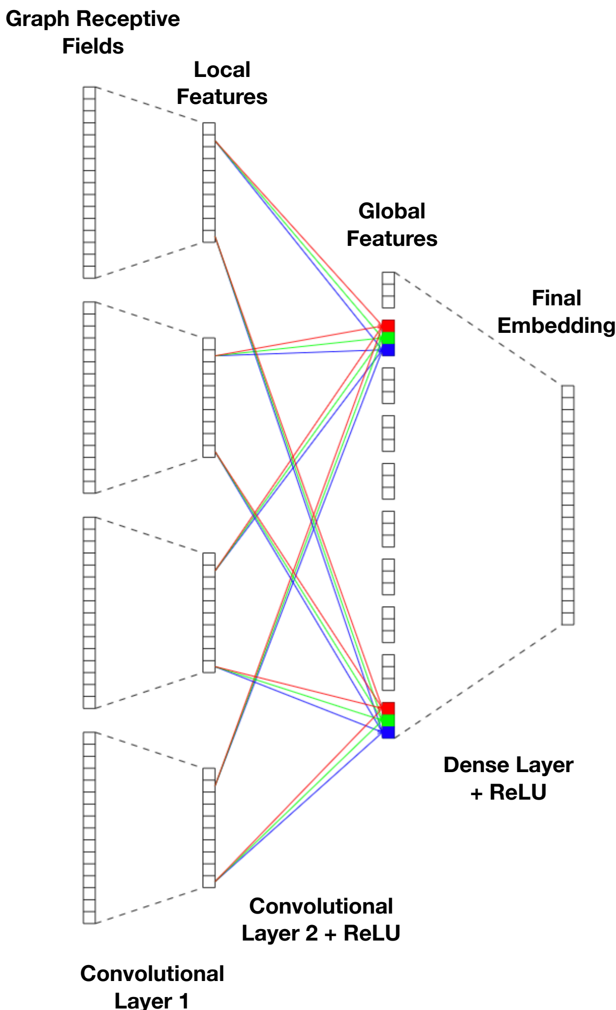


Figure 7: Overview of GESS (second part): Convolutional neural network generating GESS embeddings. This toy example takes 4 receptive fields of size 16 and has convolutional and dense layers with output dimensions 10, 3, and 20, respectively.

- Nodes have at most two successors (un-/conditional branches);
- Graphs have a unique root node (function entry point).

As with graph neural networks, nodes are given an embedding that is initialized with their node attributes. Then, the message passing phase takes place: nodes send their embedding to their predecessor without any message computation or aggregation (contrary to GNNs). Each node receives the embeddings of its successors, thus receiving at most two embeddings. All nodes take their current embedding and the two received embeddings and pass them through a neural network in parallel. For the nodes that receive strictly less than two messages, empty spots are replaced by zeros. Information therefore flows from function exit points to the function entry point, following the paths the function would take during execution in reverse order. Since information flows towards the function entry point, after a sufficient number of iterations, the entry point's embedding should contain information about the whole graph. Therefore, the embedding of the entry point is returned as the graph embedding.

This approach heavily relies on the fact that every relevant basic block is reachable from the function entry point. However, functions might contain indirect branch instructions, i.e., branches to a target represented by a register or memory value. The effective target of such a branch might depend on runtime values (e.g., user-controlled) or be the result of a complicated computation, which in turn cannot be tracked by a static disassembler. In consequence, statically extracted ACFGs can be incomplete. This is problematic for this approach because, if no edge connects two blocks, no message is passed between them. Taking the extreme case of disconnected components for example, no information can flow from the blocks that are not connected to the function entry point, thus having no influence on the function embedding computed at the entry point. This prevents the final embedding from properly representing the whole function.

In Figure 8, we notice that, for ACFG (a), there is a path following red arrows from every node to the function entry point (red node, here labelled A). Since arrows represent message passing, we conclude that, with a sufficient number of iterations, information from every node can influence the embedding of the entry point. However, for ACFG (b), nodes C to F have no red-arrow path to the entry point. Therefore, the embedding of the entry point

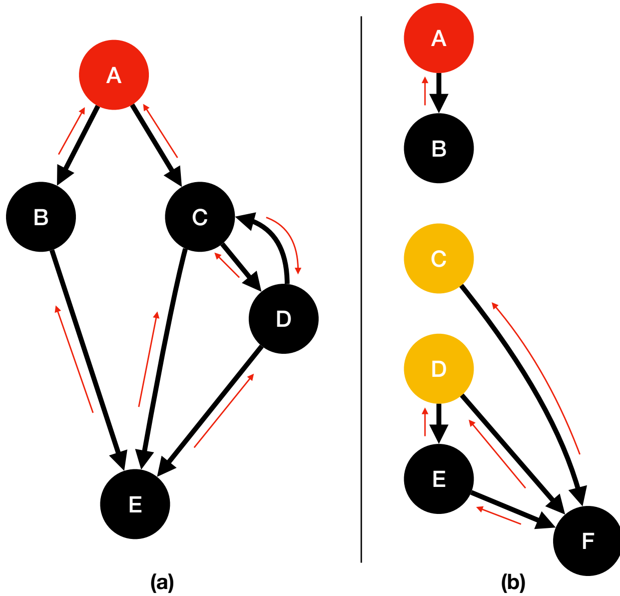


Figure 8: Two different ACFGs: (a) connected, and (b) disconnected. The red nodes are function entry points while the yellow nodes are nodes with no predecessor. The thin red arrows represent the backward message passing (BMP) described in Section 3.2

cannot appropriately represent the whole function.

In order to deal with this issue, we pool all basic blocks with no predecessor (except the entry point). Their embeddings are summed, passed through a neural network, and finally added to the graph embedding. While this does not resolve ACFG incompleteness, it still allows the function embedding to contain information about the code that is only reachable via indirect jumps. Therefore, in Figure 8 (b), the approach would sum the embeddings of the yellow nodes (C and D) at the end of the iteration process and pass them through a neural network before adding this result to the embedding of the entry point.

One problem of pooling the yellow nodes is that there is no way to determine whether C and D are possible targets of function-local indirect jumps, or just some code that is unreachable from within the function. Since these nodes are taken into account when computing the embedding, this embedding can now depend on unreachable code (that might even belong to a completely different function). This is undesirable, since unreachable code does not change the way the function works, and therefore a perfect embedding function should not depend on it. This approach has nevertheless been chosen, based on the assumption that such unreachable code blocks are uncommon. It is also worth noting that all other approaches evaluated in this paper, including previous work, suffer from the same problem in the presence of unreachable code.

4. Experimental Results

All of the following experiments have been carried out on an 8-core Intel Xeon E5-2620 CPU (2.1GHz) with 128 GB of RAM. The tensor operations were dispatched to an Nvidia Titan V GPU with 12 GB of memory using CUDA 10.1.

4.1. Dataset

Our experimental dataset consists of two open-source software components, commonly used in IoT devices: *OpenSSL* (v3.0.0 alpha13) and *binutils* (v2.36.1). All functions were compiled for three different architectures (i386, armhf, mips; all 32-bits) using two different compilers (gcc v8.3.0.6 [19] and clang v7.0.1.8 [20]).[2] For each combination of compiler and architecture, we used five different optimization levels (-O0, -O1, -O2, -O3 and -Os). In order to be able to recover function names after compilation and thus maintain a ground truth, debugging information (-g flag) was included in all binaries.

ACFGs have then been extracted from the resulting binaries using Radare2 [14]. Only graphs with strictly more than five nodes have been kept in our dataset (in accordance to previous work [15]). Finally, we divided our dataset into training (70%), validation (15%), and testing (15%) sets in such a way that:

- All ACFGs originating from the same source function are in the same set. This way, validation and testing are performed on functions unknown during training.
- The training/validation/testing ratio is the same for all used software components (i.e., OpenSSL and binutils).

This provides us with a final dataset of $607{,}940$ attributed control flows graphs: $468{,}538$ from binutils and $139{,}402$ from OpenSSL. The graphs from binutils have an average, median, and maximum size (i.e., number of nodes) of 30, 15, and 874, respectively. The graphs from OpenSSL have an average, median, and maximum size of 19, 12, and 2445, respectively. Note that, for all graphs, the minimum size is 6 by design.

4.2. Experimental Protocol

Training Protocol. We generate a graph embedding for each ACFG. The graph embeddings serve as ACFG "signatures" where we can compare two ACFGs by comparing their signatures. We want to train our models such that similar ACFGs should have similar signatures, i.e., similar graph embeddings. Here we assume that two ACFGs originating from the same source-level function are similar, and thus need to be classified as such by the approaches. To this end, as shown in Figure 9, we leverage a Siamese architecture [22] to train our model, similarly to what was proposed in Gemini [6].

Let f be our graph embedding model. f takes an ACFG G as input and returns a graph embedding z_G: $z_G = f(G)$. Let $\langle (G, G'), y \rangle$ be a training sample where (G, G') is a pair of ACFGs, and $y = 1$ if G and G' are similar while $y = -1$ if they are dissimilar. Given a set of n training samples $\langle (G_i, G_i'), y_i \rangle$, we want to train

2. A known bug prevented compilation for mips using clang. Therefore, we only used gcc to compile for the mips architecture [21].

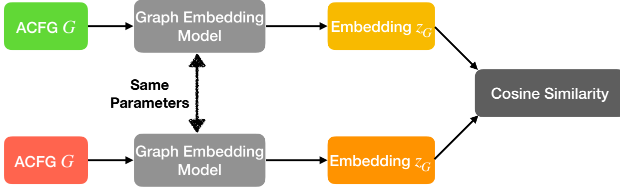


Figure 9: Siamese network architecture. It takes two ACFGs G and G' as input, transform them into two graph embeddings z_G and $z_{G'}$, and then computes the cosine similarity between z_G and $z_{G'}$.

the graph embedding model f such that it minimizes the following loss function:

$$\sum_i^n (sim(f(G_i), f(G_i')) - y_i)^2$$

where sim is a similarity function that measures the similarity between two embeddings. In our paper, we rely on cosine similarity.

Training data generation. To train our models, we need to generate training samples, i.e., pairs of ACFGs. Computing all pairs of ACFGs using a Cartesian product and training on them would however be too computationally expensive. Therefore, we resort to sample pairs as follows.

For positive samples, for each ACFG a, we sample m similar ACFGs (originating from the same source function) at random and pair them with a. These pairs of similar ACFGs are given a training label of 1 corresponding to the cosine similarity of two identical embeddings.

For negative samples, we also sample m different ACFGs (originating from different source functions) at random and pair them with a. These pairs are given a label of -1 corresponding to the cosine similarity of two opposite embeddings. This gives a balanced dataset containing as many similar pairs as dissimilar pairs. During the training and validation phases, for each ACFG is created one pair with a similar ACFG, as well as one pair with a different ACFG. We compute different pairings for each epoch. This allows us to train the model on a higher number of pairs and to reduce the risks of overfitting. For the testing phase, we generate 10 similar and 10 different pairs for each ACFG. We select a higher number of pairs in order to get more robust results across different testing conditions with random pairings.

Training process. After each training epoch, we validate the model on our validation set and, at the end of the training phase, we test it on the testing set. Validation and testing follow the same protocol:

1) Pairs of ACGFs are generated from the data.
2) Each pair is fed to the graph embedding model in order to compute two embeddings, one for each ACFG in the pair.
3) The cosine similarity between the embeddings of each pair is computed.
4) The pair labels and their cosine similarity are used in order to compute a receiver operating characteristic (ROC) curve. The ROC curve shows the true positive rate (TPR) on the y-axis against the false positive rate (FPR) on the x-axis.
5) The area under the ROC curve (AUC) is computed.

K	W	Validation Loss
10	8	0.65629
8	8	0.65894
8	9	0.66034
6	10	0.66037
10	6	0.66125
5	9	0.66324
8	7	0.66519
2	9	0.66542
4	10	0.66598
9	10	0.66618

TABLE 1: Receptive field hyperparameters. Validation loss with respect to the number and size of the receptive fields (sorted by increasing validation loss).

Note that an AUC of 0.5 corresponds to random guessing (i.e., false positive rate = true positive rate) while an AUC of 1 corresponds to a perfect prediction or classification.

Early stopping. We continue training our model until the validation ROC AUC starts decreasing. Each model has a patience parameter h, and, if the validation ROC AUC does not increase for h epochs, training stops. h is chosen empirically depending on the variability of the validation ROC AUC (i.e., h needs to be high enough for the model to not stop training if it can still improve while h should not be too low for the training to end prematurely). After each iteration, if the highest validation ROC AUC is reached, the model is stored. For testing, we load the best model according to the validation ROC AUC.

Hyperparameter Selection. All experiments on GESS were performed using TensorFlow 2.3.0. To tune the hyperparameters, we vary one of them while fixing the others. For each hyperparameter, we select the best value according to the validation AUC. For each configuration of the hyperparameters, the model is trained for a fixed number S of epochs. In order to speed-up the process, the same pairs are used for all S epochs and recomputed only between iterations. Moreover, the validation loss is only computed after the S epochs. This loss is the value that we want to minimize.

GESS has the following hyperparameters:

- The weights $p_{i,1\leq i\leq 9}$ of the labeling function;
- The number of receptive fields (K) and the size (or width W) of these fields;
- The number of convolutional and dense layers of the CNN, and the size of these layers.

Labeling function. To tune the weights of the labeling function, we rely on simulated annealing [23], a metaheuristic used to find the (approximate) global optimum of a given function. The weights p_i are all initialized by 0, and at each iteration, a neighbor of the weight vector is computed by adding a value uniformly distributed in $[-1, 1] \setminus \{0\}$ to one of the weights. For the other hyperparameters, we fix $K = 4$, $W = 5$, the CNN is fixed to have two convolutional layers of sizes 16 and 3, and a dense layer of size 10. For our tuning process, we chose $S = 7$ and simulated annealing parameters $\alpha = 0.95$, $T_0 = 0.7$ and $T_{\text{end}} = 0.002$.

Receptive fields. The number of receptive fields K and their size W have been optimized using grid search. More

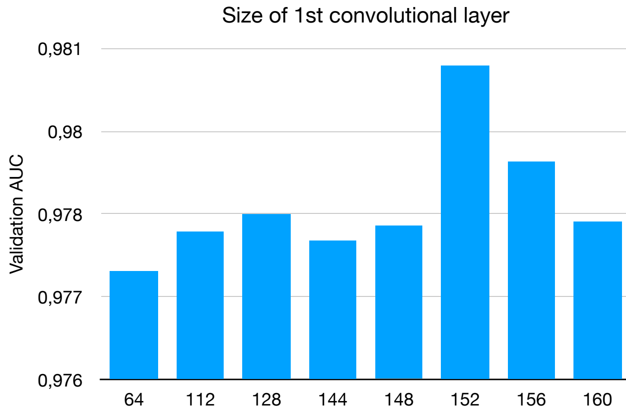


Figure 10: Validation ROC AUC with respect to the output size of the first convolutional layer, for second convolutional layer and dense layer of sizes 30 and 25, respectively.

precisely, we tried receptive field numbers and sizes in the range of 1 to 10. For each combination of K and W, a dataset of pairs of ACFGs is computed and used for all S training epochs as discussed above. The experiments showed that, for both parameters, no significant improvement resulted from selecting values over 8 for both hyperparameters, as shown in Table 1. Therefore, in order to obtain a small CNN and eventually get a more efficient implementation, both hyperparameters have been set to 8.

CNN layers. For the CNN, we tried with different network depth and observed that a network depth of three (two convolutional layers and one dense layer) gave the best results. In order to select the size of the convolutional and dense layers, we first tried with some random values before focusing on the most promising ones. For example, regarding the output size of the first convolutional layer, we first tried with the values shown in Figure 10. We observe that the best output size of the first convolutional layer is 152. To be sure, we tried the values close to the approximate best size (in our example, it would be 144, 148, 156, and 160). We repeat this process for other hyperparameters of the CNN. Finally, we selected convolutional layers of sizes 152 and 30, and a dense layer of size 25 as the maximum validation AUC was reached with these values.

Backward Message Passing. We also implemented this approach with TensorFlow 2.3.0. The optimal hyperparameters for this method are three iterations of message passing, and two dense layers: (i) one with input size of 27 (3 node embeddings of size 9) and output size of 30, and (ii) one with input size of 30 and output size 9. Furthermore, we set the output size of the dense layer to 9 in order to match the size of the node embeddings used in the next iteration of BMP. Indeed, with the BMP method, the node embedding is initialized with the node attributes, and is thus of size 9.

GNNs. We implemented the three GNNs introduced in Section 2.3 by relying on PyTorch 1.7.1 and PyTorch Geometric 1.7.0 [24].

GCN. We implemented GCN with global mean pooling (note that global sum pooling would give collinear embeddings, and so would give the same cosine similarity,

Approach	AUC
GraphSAGE (DP)	0.813
GCN	0.864
GraphSAGE (GMP)	0.917
GIN	0.920
BMP	0.930
Gemini	0.949
GESS	**0.979**

TABLE 2: Testing area under the ROC curve (AUC) of the different graph embedding methods under study. GESS outperforms all other methods, and GNN methods provide the worst AUC. Note that GraphSAGE has been tested with two pooling methods: global mean pooling (GMP) and differential pooling (DP).

thus the same loss). We experimentally tested this GNN with 1, 2, 3, 5, and 10 iterations, and found an optimum at 3 iterations. We believe that the fact that performance is poor with a higher number of iterations is due to the phenomenon of over-smoothing: as the number of iterations increases, node embeddings get too general and do not represent any part of the graph anymore.

GraphSAGE. We implemented GraphSAGE with global mean pooling and with differential pooling. However, the GNN that learns clustering assignments in differential pooling needs a fixed number of outputs. Therefore, the number of clusters is fixed. The problem is that the input ACFGs can have from 6 to more than $2,000$ nodes. Therefore, it is hard to find a fixed number of clusters that suits all graphs. The experiments were performed with 30 clusters at the first iteration, 15 at the second one, 8 at the third, and so on until 1. We further performed our experiments with 1, 2, 3, 5, and 10 iterations of GraphSAGE for each iteration of differential pooling. The optimum was reached with 2 iterations of GraphSAGE.

GIN. The graph embedding returned by a GIN is the concatenation for all iterations of the sum of the node embeddings. Since the results of the first iteration are also contained in the embedding, we can run it for a larger number of iterations without over-smoothing. Thus, we tried with 8, 10, and 12 iterations. After experimenting GIN with different neural networks of different depths and sizes, an optimum was found experimentally using a neural network with 4 hidden layers of 70 nodes. There are therefore 5 layers in total, including the output layer (of size 9 to pass the output to the embeddings of the next iteration like in BMP). The optimal ROC AUC was reached using GIN with this neural network for 12 iterations.

Baseline. We compared our newly proposed methods with the best existing approach, Gemini [6], as a baseline. We re-implemented Gemini by optimizing its performance using the most recent version of PyTorch.

4.3. General Results

Table 2 summarizes the results of all the graph embedding methods under study. We first observe that GESS, with an AUC of 0.979, outperforms Gemini (AUC = 0.949). Our other approach, backward message passing (BMP), reaches an AUC of 0.930, lower than Gemini

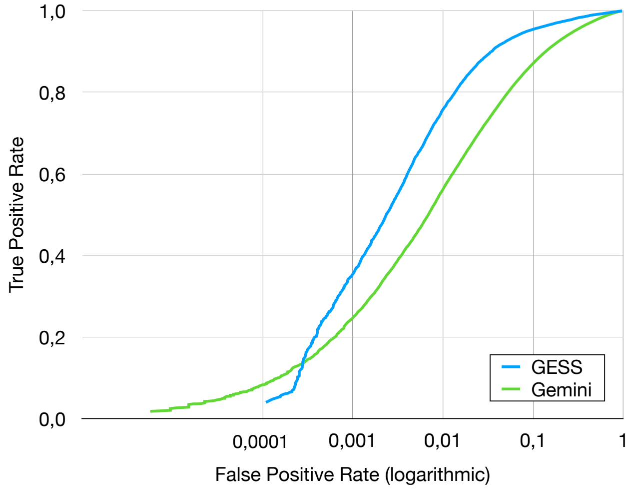


Figure 11: ROC curves of GESS and Gemini computed on the testing set (with logarithmic *X*-axis).

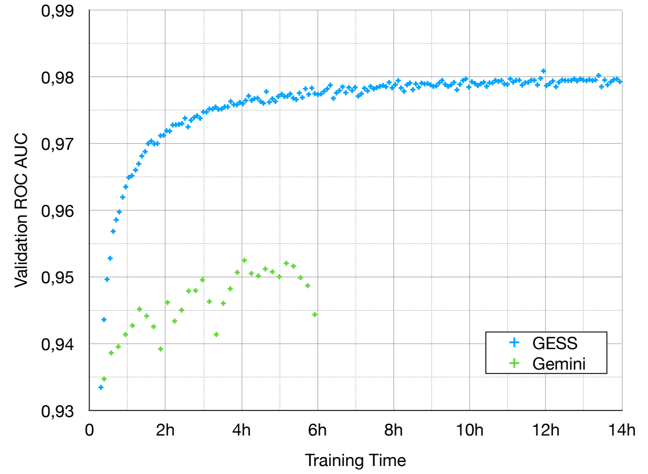


Figure 12: Validation AUC of GESS and Gemini with respect to the training time. We observe that GESS can reach the same AUC as Gemini in 40 minutes instead of more than 4 hours (6 faster).

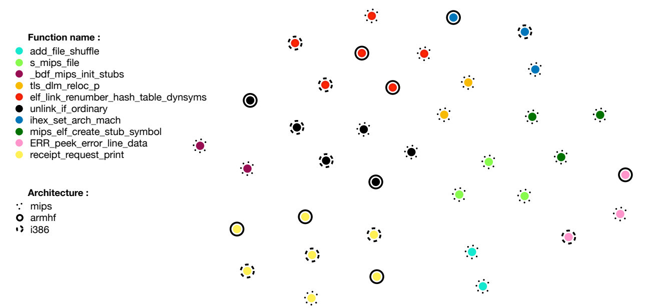


Figure 13: Two-dimensional map of the ACFGs compiled from 10 different source functions (represented by different colors) for three architectures (represented by different circle line types), plotted using the t-SNE dimensionality reduction technique [25].

and GESS, but it nevertheless represents an interesting avenue for future research. Indeed, this method remains relatively simple and could be further optimized. On the contrary, all approaches based on graph neural networks (GNNs) show relatively low AUC. Note that we have implemented GraphSAGE both with global mean pooling and differential pooling, and we see that global mean pooling provides better AUC (of more than 0.1 point). Given this result, we have implemented only global mean pooling for the GCN.

Let us now examine the ROC curves of Gemini and GESS in Figure 11. We observe that, for a fixed false positive rate (FPR) of 0:01, GESS achieves a true positive rate (TPR) of 0:76 against 0:56 for Gemini, which represents a relative TPR increase of 35:7%. Therefore, if we want to keep a low false positive rate while maximizing the true positive rate, GESS is significantly better than the best existing method.

Figure 12 shows the validation AUC with respect to the training time for both GESS and Gemini. First, we observe that GESS can reach the same AUC as Gemini in 40 minutes instead of more than 4 hours (6 faster). On the other hand, if we favor the accuracy (AUC) over the time performance, we can pay the price of a longer training time (of about 14 hours) with GESS against about 5 hours for Gemini. However, we can observe that, for a 5-hour training time, GESS achieves an AUC already close to its maximum validation AUC and, above all, much higher than Gemini. We further measure the overall testing time for Gemini and GESS. The average testing time is of 138 seconds – 0.235 ms per pair sample – for Gemini and 29 seconds – 0.049 ms per pair sample – for GESS, which implies that GESS is more than four times faster than Gemini at determining whether two binary functions are similar or not.

In order to better visualize the graph embeddings returned by GESS, we plot the embeddings of 10 source functions in a two-dimensional map by relying on the t-SNE dimensionality reduction technique [25]. Figure 13 depicts the 10 source functions compiled for three different architectures (mips, armhf, i386) while Figure 14 depicts the same source functions compiled with two different compilers (gcc and clang). In both figures, we observe that the embeddings representing the 10 different functions (ACFGs) are clustered together (i.e., by color) and not by architecture or compiler type. This shows how our graph embedding method can efficiently detect similar source functions even when these are compiled with different architectures or compilers.

4.4. Cross-Dataset Training

Next, we would like to investigate whether the graph embedding models can generalize to completely unknown data. Indeed, while the ACFGs on which validation and testing is performed are extracted from different functions than the training ACFGs, so far all our dataset functions contain both binutils and OpenSSL, and thus might learn characteristics from both binutils and OpenSSL (e.g., coding style, functionality, etc.). In order to determine how robust Gemini and GESS are when using cross-dataset training, we split up ACFGs from OpenSSL and binutils, and we then train and validate on the OpenSSL dataset and test on the binutils dataset or, vice versa, train and validate on the binutils dataset and test on the OpenSSL dataset.

The corresponding results are shown in Table 3. First, we observe that the area under the ROC curve is higher

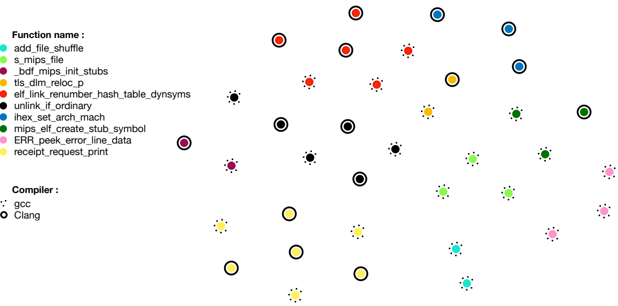


Figure 14: Two-dimensional map of the ACFGs compiled from 10 different source functions (represented by different colors) with two compilers (represented by different circle line types), plotted using the t-SNE dimensionality reduction technique [25].

Train Data	OpenSSL		binutils	
Test Data	OpenSSL	binutils	OpenSSL	binutils
Gemini	0.968	0.898	0.942	0.947
GESS	**0.978**	**0.931**	**0.954**	**0.980**

TABLE 3: Testing AUC after cross-dataset training, i.e., training and testing with different software components (OpenSSL or binutils). For each configuration, the best AUC is emphasized in bold.

when the model has been trained on functions from the same software component than on functions from another software component. We also notice that, for both models, the AUC is higher when training on binutils and testing on OpenSSL than the opposite. One explanation for this might be that binutils contains more functions than OpenSSL and that binutils is more diverse in the sense that it encompasses a collection of various binary tools.[3] Thus, a model trained on all these tools might generalize better than a model trained only on OpenSSL. This shows the importance of having a diverse training dataset for accurately detecting similar binary functions.

Last but not least, we observe that GESS performs better than Gemini in all scenarios and, in particular, when testing with the binutils dataset where the AUC is always higher for GESS by at least 0.003. This demonstrates the ability of GESS to handle better the diversity of the binutils source functions than Gemini.

4.5. Identifying Unknown Similar Functions

So far we have constructed our training, validation, and test datasets such that they contain as many similar pairs as dissimilar pairs. Moreover, in our experimental setting, we have tried to guess whether a given pair was similar or not, but never tried to identify all possible functions similar to a given vulnerable function. In a practical scenario where we want to detect a vulnerability, we have to consider the whole set of binary functions and compare them with the vulnerable function.

In a practical scenario where we want to detect some vulnerabilities in a given target function, we have to consider the whole set of binary functions and compare them with the target function.

3. The list of the binary tools is available here: https://www.gnu.org/software/binutils

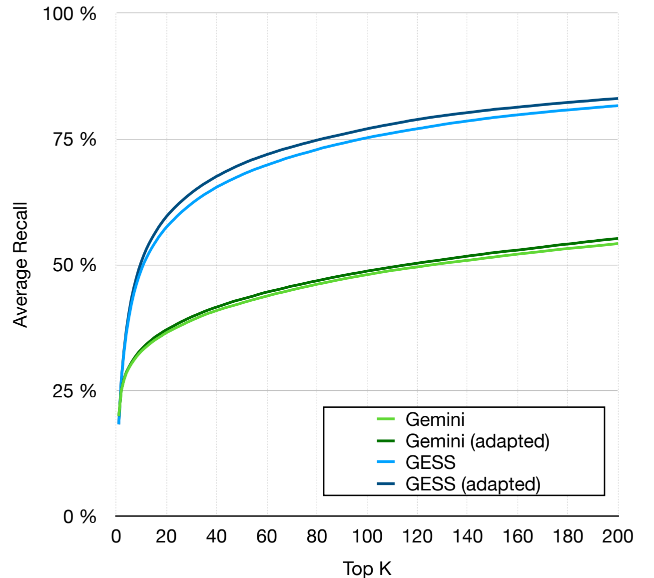


Figure 15: Average recall on the testing dataset when retrieving the top-k nearest functions.

Therefore, in this new setting, we cannot rely on the ROC curve and AUC but we rather should compute the recall and precision results averaged over all ACFGs. In order to identify all possible similar binary functions from the whole set of possible functions, we select the functions that are the k-nearest neighbors (in terms of cosine similarity) of the target function. In order to get the nearest neighbors, we compute the distance between graph embeddings of the ACFGs by relying on FAISS [26] which provides efficient nearest-neighbor search by optimizing the memory-speed-accuracy trade-off. It is in practice several orders of magnitude faster than any other k-nearest neighbor (k-NN) implementation. Moreover, in terms of accuracy, the difference between FAISS and traditional k-NN approaches is negligible in our setting.

We fine-tune our model for this new experimental setting by adapting the manner model validation is performed. Instead of relying on the validation AUC as before, we now aim to maximize the recall averaged over all possible values between 1 and k for every ACFG in the validation set. Changing the validation process has a direct impact on the model since the validation metric is what determines the number of training epochs. In the following results, we refer to these new models as *adapted*. The training process remains the same as the one described in Section 4.2. Finally, for the testing phase, for every ACFG in the testing set, we get the k nearest neighbors among all other ACFGs in the testing set.

Similarly to Feng et al. [5], we perform our experiments for k ranging from 1 to 200. We measure, for every k, the average recall and average precision over all ACFGs in the testing set. Figure 15 depicts the average recall while Figure 16 depicts the average precision.

First, as expected, the adapted models perform slightly better than the original ones on this problem, and thus we focus on these results in the following. We observe that, when retrieving the 20 nearest neighbors, GESS (adapted) provides an average recall of 59.6% and Gemini (adapted) an average recall of 37.0%, which represents a relative increase of more than 60%. For $k = 200$, the average

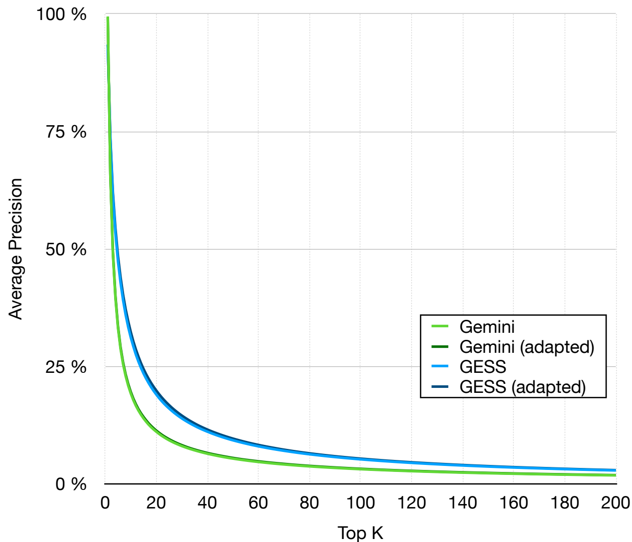


Figure 16: Average precision on the testing dataset when retrieving the top-k nearest functions.

recall of GESS (adapted) is of 83.1% while the recall of Gemini (adapted) is of 55.2%, which represents an increase of more than 50%. In fact, the average recall of GESS is at least 50% greater than the average recall of Gemini for any $k > 8$.

We notice that GESS achieves a recall of 50% when retrieving the top-10 nearest neighbors, while Gemini needs to retrieve 116 nearest neighbors for a comparable performance, which then dramatically affects the precision. As we can observe in Figure 16, the average precision quickly drops for increasing values of k, but GESS still outperforms Gemini. If we take $k = 10$, we observe that GESS still provides a decent average precision of 32.3%. If we retrieve the 116 nearest neighbors to reach the same recall with Gemini, the average precision drops to less than 3%. This implies that a security analyst would only identify one actual vulnerable function out of the 30 functions (detected by Gemini) that they would inspect.

5. Related Work

We summarize here the main previous approaches tackling the problem of binary code similarity detection, and in particular the cross-architecture methods that rely on control flow graphs.

Pewny et al. proposed the first effective cross-architecture similarity detection technique based on graph matching [4]. Their approach works by tracing the input/output operations in a binary function and by using them as node features in the control flow graph. It then compares graphs individually using graph matching. In order to provide a more efficient approach, Eschweiler et al. propose to rely on features that are simple – and therefore faster to extract – such as the total number of instructions and the number of arithmetic instructions [15]. However, their approach still relies on slow graph matching algorithms.

Feng et al. borrow techniques from image processing and adapt them to graphs for bug detection [5]. Their approach, called Genius, relies on unsupervised techniques to extract an embedding from a graph and uses locality-sensitive hashing (LSH) in order to perform an embedding search in scalable time. The main drawback of this approach is that it relies on bipartite graph matching to learn its embedding function. This is still highly inefficient and requires an offline phase of training before being able to compute the embeddings. This prevents the model from being updated frequently, for instance to take newly discovered vulnerabilities into account.

Phan et al. [27] propose to use graph-based CNN in addition to max pooling to compute the graph embedding. Their approach uses a CNN with fixed receptive fields in the CFGs. In their approach, nodes within a receptive field are treated differently whether they are outgoing, incoming or current nodes as they are assigned a different filter. This increases the number of parameters and leads to potential overfitting. Our approach which uses backward message passing can alleviate this problem as it does not require treating nodes differently. In addition, the problem they want to solve is different than ours. They want to classify whether a program can run without errors or it cannot compile or it has syntactical problems. In our setting, we want to check whether two functions are similar based on their ACFGs. This is considered to be harder as we need to train the model in an unsupervised manner.

Similar to the above approach, Massarelli et al. [28] propose a graph embedding model based on message passing neural network. Their approach differs from ours and [27] as they learn the node features instead of constructing them manually. This is done based on word2vec where each instruction is considered as a word and the whole function is considered as a document. SimInspector [29] is another approach that aims to learn the node features of the CFGs. Their approach involves 3 steps. First, instruction embeddings are constructed similar to [28]. In the second step, the features of nodes of the CFGs are built using BiLSTM where the inputs are the instruction embeddings. Finally, the authors use structure2vec [30] to construct the graph embedding from the node features obtained from the BiLSTM. The idea of learned node features is interesting, and we will consider it in future work.

Ding et al. [31] aim to construct "paragraph" embeddings where a paragraph embedding captures the semantic and structure of an assembly function and its CFG. First, from each CFG, they identify possible execution paths, e.g., by using random walks on the CFG. The execution paths of a CFG are considered as sentences and they together form a paragraph. The paragraph embeddings can then be constructed using an improved version of word2vec that can construct embeddings for paragraphs. As their approach does not use a graph embedding model, it cannot capture the relations between nodes in CFGs directly. While the execution paths can somewhat capture the relations, they are only approximations.

Xu et al. use supervised learning techniques and deep neural networks to get an embedding that learns on the data some notion of similarity [6]. Their method, referred to as Gemini, relies on neural networks and computes graph embeddings using structure2vec [30]. The model is trained end-to-end using Siamese networks [32]. FuncNet [33] is a similar method to Gemini as it uses structure2vec [30] as the underlying graph embedding model.

However, FuncNet uses triplet loss to learn the model in an end-to-end manner. Compared to Genius, Gemini does not rely on expensive graph matching, which makes it train faster (in the order of hours). Moreover, the embedding function is a neural network and can be retrained quickly in order to account for new data or for a difference in the definition of similar graphs. It also gives more accurate results on the same data and computes the graph embeddings more than 380 times faster than Genius. Given these results and the similarity between FuncNet and Gemini, we re-implemented Gemini and used it as our comparison baseline.

Li et al. propose to extend graph neural network (GNN) models for similarity learning [34]. Their new graph matching networks derive a similarity score through a cross-graph attention mechanism to associate nodes across graphs and identify differences. It provides very good accuracy on a dataset generated from a popular open-source video processing software. However, it only matches graphs by pair, i.e., takes a pair of graphs as input and returns a similarity score as output, which significantly reduces the scalability of the approach. Indeed, this approach would be prohibitive in the case where we want to detect some vulnerabilities in a given target function which requires to compare it with a large set of binary functions (as described in Section 4.5).

Song et al. [35] tackle a different problem than ours. They want to classify kernel objects from the memory snapshot of a running system. Recognizing kernel objects is important in identifying if an attack is happening. They construct a memory graph which captures the point-to relations of memory segments and their adjacency. They design a GNN architecture specifically for the memory graph which captures the node features, node relations, and temporal information. While their approach also uses GNN to construct graph embedding, the underlying memory graphs are different from our CFGs which makes their GNN not applicable to our problem.

Besides the aforementioned related academic research, a blog post from Google's Project Zero team demonstrates the practical value that binary code similarity detection can have for supporting bug hunters [36]. In their work, Google's team developed an Apache-licensed C++ toolkit to find statically-linked copies of (vulnerable) third-party library functions in binary executables. The proposed approach uses a linear function (based on SimHashing) to hash disassembled functions, preserving some notion of function similarity. As a potential way for improving their tool in the future, the academic literature reviewed in this section is mentioned.

Finally, note that earlier approaches have been proposed for cross-platform bug search [37]–[40], but these are too costly to be applied for large-scale firmware bug search and will thus not be discussed in details here.

6. Conclusion and Future Work

In this paper, we have investigated several graph embedding methods for performing cross-platform binary code similarity detection, including graph neural networks (GNNs) and the best existing approach, Gemini. Building upon the recent advances in deep learning, we have proposed and implemented a new method, GESS, that outperforms all other graph embedding methods. More specifically, GESS enables us to reach, for a fixed false positive rate of 0.01, a true positive rate 36% higher than the best existing approach. The improvement provided by GESS is even more striking when considering a more realistic setting of a large function search space. In this scenario, we show that, in order to reach the same recall rate (detection capability) of about 50%, GESS reduces the number of binary functions to inspect for a security analyst by ten compared to Gemini.

In future work, our newly proposed method could be tested on datasets that contain functions with known vulnerabilities, and it could be applied to functions extracted from widely used device firmwares. Computing function similarities on such datasets would allow to measure GESS' performance on a more concrete scenario, with the goal to discover security vulnerabilities in these devices.

Acknowledgment

The authors would like to thank Eric Jollès and Patrick Schaller for providing feedback on the manuscript and our shepherd for his guidance and feedback on the revised manuscript.

References

[1] A. Cui, M. Costello, and S. J. Stolfo, "When Firmware Modifications Attack: A Case Study of Embedded Exploitation," p. 13, 2013.

[2] A. Greenberg, "Hackers remotely kill a jeep on the highway-with me in it," Jul 2015. [Online]. Available: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[3] J. Berger, "A dam, small and unsung, is caught up in an iranian hacking case," Mar 2016. [Online]. Available: https://www.nytimes.com/2016/03/26/nyregion/rye-brook-dam-caught-in-computer-hacking-case.html

[4] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-Architecture Bug Search in Binary Executables," in *2015 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2015, pp. 709–724. [Online]. Available: https://ieeexplore.ieee.org/document/7163056/

[5] 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 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pp. 480–491. [Online]. Available: https://dl.acm.org/doi/10.1145/2976749.2978370

[6] 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 2017 ACM SIGSAC Conference on Computer and Communications Security*. Dallas Texas USA: ACM, Oct. 2017, pp. 363–376. [Online]. Available: https://dl.acm.org/doi/10.1145/3133956.3134018

[7] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," *arXiv:1706.02216 [cs, stat]*, Sep. 2018, arXiv: 1706.02216. [Online]. Available: http://arxiv.org/abs/1706.02216

[8] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," *arXiv:1609.02907 [cs, stat]*, Feb. 2017, arXiv: 1609.02907. [Online]. Available: http://arxiv.org/abs/1609.02907

[9] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How Powerful are Graph Neural Networks?" *arXiv:1810.00826 [cs, stat]*, Feb. 2019, arXiv: 1810.00826. [Online]. Available: http://arxiv.org/abs/1810.00826

[10] Z. Diao, X. Wang, D. Zhang, Y. Liu, K. Xie, and S. He, "Dynamic spatial-temporal graph convolutional neural networks for traffic forecasting," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 890–897.

[11] W. Torng and R. B. Altman, "Graph convolutional neural networks for predicting drug-target interactions," *Journal of chemical information and modeling*, vol. 59, no. 10, pp. 4131–4149, 2019.

[12] S. He, F. Bastani, S. Jagwani, E. Park, S. Abbar, M. Alizadeh, H. Balakrishnan, S. Chawla, S. Madden, and M. A. Sadeghi, "Roadtagger: Robust road attribute inference with graph neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, 2020, pp. 10 965–10 972.

[13] A. Pal, C. Eksombatchai, Y. Zhou, B. Zhao, C. Rosenberg, and J. Leskovec, "Pinnersage: multi-modal user embedding framework for recommendations at pinterest," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 2311–2320.

[14] "radare2," 2021. [Online]. Available: https://rada.re/n/radare2.html

[15] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2016. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/discovre-efficient-cross-architecture-identification-bugs-binary-code.pdf

[16] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, "Hierarchical Graph Representation Learning with Differentiable Pooling," *arXiv:1806.08804 [cs, stat]*, Feb. 2019, arXiv: 1806.08804. [Online]. Available: http://arxiv.org/abs/1806.08804

[17] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning Convolutional Neural Networks for Graphs," p. 10, 2016.

[18] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *ArXiv e-prints*, 11 2015.

[19] "Gcc, the gnu compiler collection," 2021. [Online]. Available: https://gcc.gnu.org/

[20] "Clang: a c language family frontend for llvm," 2021. [Online]. Available: https://clang.llvm.org/

[21] "Bug 48623," Dec. 2020. [Online]. Available: https://bugs.llvm.org/show_bug.cgi?id=48623

[22] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," *Advances in Neural Information Processing Systems*, vol. 6, pp. 737–744, 1993.

[23] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[24] "Pytorch geometric documentation¶," 2021. [Online]. Available: https://pytorch-geometric.readthedocs.io/en/latest/

[25] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[26] "Faiss," 2021. [Online]. Available: https://ai.facebook.com/tools/faiss

[27] A. V. Phan, M. Le Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2017, pp. 45–52.

[28] 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 *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, 2019.

[29] X. Zhu, L. Jiang, and Z. Chen, "Cross-platform binary code similarity detection based on nmt and graph embedding," *Mathematical Biosciences and Engineering*, vol. 18, no. 4, pp. 4528–4551, 2021.

[30] H. Dai, B. Dai, and L. Song, "Discriminative Embeddings of Latent Variable Models for Structured Data," p. 10, 2016.

[31] 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 *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.

[32] J. Bromley, J. W. Bentz, L. Bottou, I. Guyon, Y. Lecun, C. Moore, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 07, no. 04, p. 669–688, 1993.

[33] M. Luo, C. Yang, X. Gong, and L. Yu, "Funcnet: A euclidean embedding approach for lightweight cross-platform binary recognition," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2019, pp. 319–337.

[34] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *International conference on machine learning*. PMLR, 2019, pp. 3835–3845.

[35] W. Song, H. Yin, C. Liu, and D. Song, "Deepmem: Learning graph neural network models for fast and robust memory forensic analysis," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 606–618.

[36] T. Dullien, "Searching statically-linked vulnerable library functions in executable code," https://googleprojectzero.blogspot.com/2018/12/searching-statically-linked-vulnerable.html, accessed: 2021-09-13.

[37] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," *Sstic*, vol. 5, no. 1, p. 3, 2005.

[38] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *International Conference on Information and Communications Security*. Springer, 2008, pp. 238–255.

[39] J. Ming, M. Pan, and D. Gao, "ibinhunt: Binary hunting with interprocedural control flow," in *International Conference on Information Security and Cryptology*. Springer, 2012, pp. 92–109.

[40] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013, pp. 1–10.