Function Embedding Generation Using Program Dependency Graph Based Neural Network

Srivastava, Abhishek Kumar

2018

Peer reviewed|Thesis/dissertation
Function Embedding Generation Using Program Dependency Graph Based Neural Network.

A Thesis submitted in partial satisfaction of the requirements for the degree of

Master of Science

in

Computer Science

by

Abhishek Kumar Srivastava

June 2018

Dissertation Committee:

Dr. Heng Yin, Chairperson
Dr. Vagelis (Evangelos) Papalexakis
Dr. Silas Richelson
The Thesis of Abhishek Kumar Srivastava is approved:

________________________________________

________________________________________

________________________________________

Committee Chairperson

University of California, Riverside
Acknowledgments

There are multiple people I would like to thank, without their support this thesis could not have been completed.

First and foremost I would like to express my deepest gratitude to my advisor, Dr. Heng Yin, for his patience and helping me at each step of my thesis.

I would also like to thank all of my professors, colleagues and Computer Security Lab members who have constantly given me motivation and helped either academically or non-academically during my time at UC Riverside.
To my parents and friends for all their selfless love, encouragement and keeping me sane during tough times.
ABSTRACT OF THE DISSERTATION

Function Embedding Generation Using Program Dependency Graph Based Neural Network.

by

Abhishek Kumar Srivastava

Master of Science, Graduate Program in Computer Science
University of California, Riverside, June 2018
Dr. Heng Yin, Chairperson

Analyzing software binaries can be helpful in tackling important problems such as plagiarism, malware or vulnerability detection. Detecting similarity between two binary functions coming from different sources can be done using binary code similarity detection. Existing approaches use Control-Flow graph information of binaries in some way or another i.e. either graph matching or control-block embedding which is either slow or does not utilize all the information. In this work we propose novel way to use program dependency graph of functions to extract control and data dependency information and generate its embedding with help of Neural Network using this information. Measuring the distance between embedding of different binary functions can evaluate their similarity. Since this method does not rely on internal flow structure of the function it can be applied to more generally and is resilient to different compiler optimizations and heavy obfuscation techniques.
## Contents

**List of Figures**

**List of Tables**

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Background</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Vector Representations</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Purpose</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Outline</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Theory</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>Assembly Language</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Disassembly</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Decompilation</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Control Flow Graphs</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Program Dependency Graphs</td>
<td>13</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Control Dependency Graphs</td>
<td>14</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Data Dependency Graphs</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Neural Network</td>
<td>16</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Siamese Architecture</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>Approach</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Raw Feature Extraction</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Graph Embedding Network</td>
<td>24</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Learning Parameters Using Siamese Architecture</td>
<td>27</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Hyperparameters</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>Evaluation</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Implementation and Setup</td>
<td>30</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Dataset Generation</td>
<td>31</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Training Details</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Accuracy</td>
<td>32</td>
</tr>
</tbody>
</table>
4.2.1 CFG-Based .......................................................... 32
4.2.2 PDG-Based .......................................................... 34

5 Conclusions ............................................................... 36

Bibliography ............................................................... 37
List of Figures

1.1 Different assembly functions compiled from the same $\text{gmpz\_tdiv\_r\_2exp}$ function of Libgmp. From left to right, the assembly functions are compiled with the $\text{GCC O0}$ option and $\text{GCC O3}$ option. .............................................. 4

2.1 Siamese Architecture. ................................................................. 19

3.1 An example of a code graph on Function $\text{dtls1\_process\_heartbeat}$ (Heartbleed vulnerability). ............................................................... 24
3.2 Graph Embedding Network. ............................................................ 25

4.1 Time taken while preprocessing the binary to generate dataset using their respective approaches. ......................................................... 33
4.2 ROC curves for different approaches evaluated on the dataset created from $\text{openssl}$ binary. ......................................................... 34
List of Tables

4.1 1st Rank Accuracy for CFG based evaluation . . . . . . . . . . . . . . . 33
4.2 Embedding generation Time for CFG based evaluation . . . . . . . . . . . 34
4.3 1st Rank Accuracy for PDG based evaluation . . . . . . . . . . . . . . . 35
4.4 Embedding generation Time for PDG based evaluation . . . . . . . . . 35
Chapter 1

Introduction

Today software applications in some form or another is used in every aspect of our lives. Softwares are being constantly developed to handle more complex issues while being scalable to millions of users and due to this complexity involved in development they may suffer from serious software implications such as exploitable malwares & vulnerabilities. The study by Cui et al. showed that 80.4% of vendor-issued firmware is released with multiple known vulnerabilities, and many recently released firmware updates contain vulnerabilities in third-party libraries that have been known for over eight years [7].

Most malware’s developed today are not created from scratch but in some way they are modification of some existing malware to create new ones. This can be used to our advantage by using code reuse detection to generate feature vectors for those and use with machine learning algorithms that can recognize new or similar malware or vulnerabilities [17]. One of the other reason for analyzing binaries for finding vulnerabilities and plagiarism are often lack of source code because of third-party software [30] or not having access
of source codes we wish to perform analysis on but almost everyone have access to the binary (i.e., executable) code. Binary analysis is thus important for understanding the inner workings of malware or finding vulnerabilities in existing systems but performing these tasks manually is a very intensive task. Due of these challenges binary analysis is an very important area of research in computer science and automated tools which can do the analysis efficiently and effectively are in very high demand.

1.1 Background

Recently, there has been bloom in research to tackle the problem of binary code similarity detection. In binary code analysis detecting similar functions using binary executables remains one of the fundamental problem and is well known as “binary code similarity detection” problem. These efforts commonly involves extracting binary code into representable form either a vector or graph to represent functions of binary. Then, it is checked that whether among extracted control flow graph function representations any functions are similar or not using a graph matching algorithm.

Graph-based matching approaches have some drawbacks. One of them is their adaptability for different applications since they are evaluated by fixed graph matching algorithm. For example, even the same code compiled with different optimization level may result in producing different control flow graphs and hence it will be not be matched by fixed graph matching algorithms[31]. Other problem is the inefficiency of graph matching algorithms performance(such as bipartite graph matching) which are considerably slow thus it will degrade the overall efficiency of code similarity detection since it heavily rely on it.
1.1.1 Vector Representations

A robust vector representation of assembly code functions is very important for a practical and efficient clone searching algorithm. However, existing clone search approaches, relies on a manual feature selection for every assembly function to create its vector representation, which can fail to capture distinguishable patterns or information among those assembly functions [11]. However, it is a challenging problem to automatically pick the best representation of a function due to the fact that an equivalent assembly function from same source code can appear to be very dissimilar if different compiler optimizations and code obfuscation techniques are used to generate the binary executable. These techniques have a strong impact on the resulting assembly instructions and the linear layout of the assembly code. Figure 1.1 shows some examples of assembly functions that correspond from the same source code but compiled with different optimization level. The major challenge is how to identify these semantically equivalent but structurally different assembly functions as clones.

Typically, different approaches for assembly clone search are divided into following categories.

**Text-based Approach**: In this approach target assembly program is considered as sequence of lines or strings. These approaches mostly do not transform code or its fragments into any other representation. Work by Johnson et al [18] is one of the example of this approach and can be extended to assembly source code. It uses Robin-Karp fingerprinting algorithm to compute fingerprints of substring and then uses the hash value to identify code clones. Studies such as David et al [10] also fall into this category. It uses customized
string editing distance to identify clone of assembly functions or code fragments. These approaches heavily rely on the layout of the code which can be easily modified by using different compiler optimization options or an obfuscater hence they are not very robust.

**Token-based Approach:** These approaches parses and transforms the assembly code functions into sequence of tokens using lexical analysis. Subsequence of these tokens are then used to identify similar assembly functions or fragments. These tokens can include constants, assembly instructions such as n-grams or n-perms [21], or assembly instructions with normalized operands [11, 13, 29]. This approach also lacks robustness for the similar reasons of Text-based approaches but more robust since it does not rely on the structure of assembly functions. Using different compiler or its optimization options may result in different instruction set thus it may fail to capture semantic relationship between the tokens.
Graph-based Approach: These approaches utilize semantic information of source code by trying to identify higher abstraction of the code. Correct detection of basic block boundaries and construction of its graph representation is very essential and one to most important factor for this approach to work correctly. Studies under this category uses subgraph isomorphism algorithms [11, 12, 27] to compare assembly functions. Graph substructures properties [14, 21, 23] can also be used to help identify similar functions. These approaches are also not very robust for the same above mentioned reason. Different compiler and optimization options can change the graph structure of functions. Some properties remains intact even with overall structure changes but Obfuscator [19] destroys even subgraph sub-structure making graph-based approaches useless.

Manual feature engineering process and several assumptions are made during afore-mentioned approaches. Thus, for distinguishing different assembly function and code based on these approaches, important patterns may get missed out. Deep learning [24] in recent years has been involved in many application domains which also includes binary analysis and has concluded pretty interesting results than other approaches. Deep learning allow us to learn representations of data(vector or graphical) with multiple levels of abstraction freeing involvement of any domain knowledge which is very big advantage of using it. It is also very adaptive and can be modified to train on data of our particular interest and can be utilized in our approach.
1.2 Purpose

We propose an assembly code clone detection framework based on Xu et al.'s work [31]. Our work is developed with significant extensions on its methods of information extraction from the binary and data used to train the underlying Neural Network. It is also different from graph-based methods since we first extracts the information from the binaries and feed this information to Neural Network to generate an embedding (vector representation) of the functions. Our approach constructs a large-scale training dataset using binary functions compiled from the same source code but for different platforms and compiler optimization levels.

All previous research on assembly code clone relies on the manual feature engineering process. Our vector representation of assembly code as a way to mitigate the aforementioned issues in current handcrafted features. Combining the graph embedding networks into a Siamese network [5] naturally captures the objective that the graph embeddings of two similar functions should be close to each other and vice versa. This entire network model can then be trained end-to-end for similarity detection. Further, we design a new training and dataset creation method using a default policy to pre-train a task-independent graph embedding network. The learning process does not also require any prior knowledge about assembly code, such as compiler optimization settings or the correct mapping between assembly functions. It only needs assembly code functions. We discuss the differences between the assembly code and text data, as well as the issues we had in applying the representation learning on assembly code.
1.3 Outline

We summarize our contributions as follows:

• We propose the neural network-based approach to generating embeddings for binary functions;

• We propose a novel approach to train the embedding network using a Siamese network so that a pre-trained model can generate embedding to be used for similarity detection;

• We implement a prototype and evaluate the model on a test set constructed from OpenSS binary;

The rest of the thesis is organized as follows. Chapter 2 gives detailed introductions to several different topics that is used throughout the thesis. Chapter 3 contains describes the approach and an overview of implementation used for the binaries analysis required by this project. Chapter 4 presents results obtained during the evaluation of the system developed. Finally, Chapter 5 presents conclusions regarding binary analysis based code similarity detection of with suggestions for future work.
Chapter 2

Theory

2.1 Assembly Language

An assembly language is a programming language which is interpreted by a processor and very similar to actual machine code. An assembly language uses mnemonics, or abbreviations, for the instructions available based on the computer architecture is used instead of ones and zeroes. It is also convenient because although it is human readable but there is also 1-1 mapping between assembly instructions and their machine code representation which resolves any ambiguity while translation between them.

There are a few different dialects of the x86 assembly language, and most notable are the Intel syntax and the AT&T syntax. Important differences between these two dialects is (a) parameter order where Intel places destination before source and AT&T places source before the destination, (b) mnemonics in the AT&T syntax are suffixed with a letter to indicate the size of the operation whereas this information is derived from the register that is used in the Intel syntax and (c) general syntax for effective addresses [28].
2.1.1 Disassembly

Disassembly is the opposite assembling which is done after compilation of source code to generate binary file of code [3, 25]. In this method machine code instructions from a binary file is transformed into a set of assembly functions. There are two methods in which disassembly can be performed. One as a static method where machine code instructions from binary is disassembled one by one but never executed. Second, dynamic method where disassembly is performed while machine code instructions are executed [25] so instructions which are never executed are left out. Static disassembly gets the advantage of processing an entire set of machine code instructions at once while dynamic disassembly can only process the subset of machine code instructions that are actually executed. Dynamic disassembly on the other hand has the advantage of extracting a sequence of instructions that is guaranteed to be correct for the execution during which it was extracted.

2.1.2 Decompilation

Transformation of set of machine-code instructions or set of assembly instructions into a high-level languages are done by using Decompilation [6]. The representation of binary program with a high-level language have many benefit over its original form since they are easy readable and can be understood very easily which can be further utilized if there was any need to do some modifications to it. Decompilation is a process that is usually performed after disassembly and applies additional transformations to the assembly instructions produced by the disassembly process, thus it should not be seen as a conflicting process to disassembly.
In order to decompile a program correctly, the programs control flow paths needs to be known, or discovered, during the process. Dynamically, this information can be obtained for the subset of instructions that is actually executed while the program is being monitored. Statically the entire program will be processed, but there will be ambiguities in the result since not all information regarding the control flow can be known in a static context.

The reason for these ambiguities that arises when statically reconstructing the control flow of a binary files stems from the fact that there is more information present in the original source code. This additional information which primarily concerns control flow is removed when the program is compiled into an executable binary since the information is not needed in order for the program to execute correctly. Fully structured code, where every control-structure has a single point of entry and a single point of exit, should have considerably less ambiguities. This is because once a point of entry is found, the point of exit can be known immediately.

The modern computer programs are developed in programming languages that are a human readable form [2, 20, 4]. The source code written by software developers is compiled into a binary format. In software development, there are two classes of binaries:

- **Machine code** - is not directly understandable by software developer, but it is directly executed by the machine; it is generated by compiler depending on the hardware characteristics;

- **Intermediate code** - like machine code, is not directly understandable by software developer and is not directly executed by the machine; the executable code is obtained
after an interpreting process performed by a specialized component called virtual machine; the most known and used virtual machines are Java Virtual Machine and Common Language Runtime (CLR) [15, 8].

Due to difficulty in understanding of executable version of a programs provided in machine code it is hard to maintain them. To implement the maintainance activities, the software developer need the source code and documentation. Another way to obtain the understandable form of the machine code is to convert it into assembly language.

The practical and positive issues of the disassembly process and its results are:

- Improvement of the portability for computer programs delivered in machine code format; unlike machine code, the intermediate code is portable due to its interpreting by a virtual machine which must be mandatorily installed on the host machine;

- The software developers determine the logical flows of the disassembled software application; the algorithms and other programming entities are extracted from the software application and used in other versions or programs;

- Security issues are identified and can be patched without access to the original source code;

As negative issue, the disassembly process can be carried out by malicious software developers to discover the vulnerabilities and holes of the computer programs to hack them. Also, the discovered logical flows and algorithms can be used in other commercial computer programs without an agreement with the owners of the disassembled computer program.
The list of the available disassemblers includes tools for Windows like IDA Pro [16], PE Explorer, W32DASM, BORG Disassembler, HT Editor, diStorm64 and Linux like Bastard Disassembler, ciasdis, objdump, gdb, lida linux interactive disassembler, ldasm.

**2.2 Control Flow Graphs**

A control flow graph (CFG) is a directed, connected, graph that is used to represent all possible paths of execution that can occur within a program. Each vertex in the control flow graph represents a basic block, a linear sequence of instructions, and each edge between two vertices represents a possible control flow path [4]. A control flow graph has two artificial vertices, $v_{\text{START}}$ and $v_{\text{END}}$. The reason for these two artificial vertices is to ensure a connected graph where every actual vertex without a predecessor is is connected with $v_{\text{START}}$ and every actual vertex without a successor is connected with $v_{\text{START}}$. Because of this addition to the graph there is a path $v_{\text{START}} \ldots v \ldots v_{\text{END}}$ for every vertex $v$ within the graph.

In control flow graph, instructions can be classified either as a Transfer Instruction (TI) or as a Non Transfer Instruction (NTI). Transfer instructions are set of instructions that might transfer control flow to a part of the program different from the address of the next instruction. Unconditional jumps, where the control flow is transferred to the target jump address; conditional jumps, where control flow might be transferred to the target jump address; and subroutine calls, where control flow is transferred to the invoked subroutine, are all examples of transfer instructions. Non transfer instructions are the set of instructions that will always transfer the control flow to the next instruction in sequence [6].
A basic block is a sequence of instructions that has a single point of entry and a single point of exit. A basic block either consists of zero or more non transfer instructions and ends with a single transfer instruction or consists of one or more non transfer instructions.

In addition to the classifications made by Cristina Cifuentes in [6], call basic block that has an indirect address reference (calculated during runtime), will be treated as if they had no outgoing edges. The reason for this addition to the classifications is to preserve context sensitivity when performing dynamic analysis. Context sensitivity in binary analysis is the concept that a functions behavior, and impact on the program, is relative to the calling context, or in other words, relative to the part of the code that called the function. Since functions are only control-dependent on a specific call-instruction as long as that instruction is part of the code that is currently being executed, adding edges to a function from each call would result in erroneous control-dependencies.

2.3 Program Dependency Graphs

The program dependency graph (PDG) makes explicit use of both the data and control dependences for each operation in a program or function. Data dependence graphs provides some optimizing compilers with an explicit representation of the definition-use relationships which are implicitly present in a source program. A control flow graph has been the usual representation for the control flow relationships of a program; the control conditions on which an operation depends can be derived from such a graph. Since both kinds of dependences are present in a single form in functions, transformations like vectorization
can treat control and data dependence uniformly throughout the binary. Program transformations such as code motion, which require interaction of the two types of dependences, can also be easily handled by our single program dependency graph.

2.3.1 Control Dependency Graphs

A control dependence graph (CDG) is a partially ordered, directed, acyclic graph where the vertices of the graph represents basic blocks of functions and the edges between any two vertices represents the control conditions on which the execution of the operations depends. A control dependence between two vertices in a control flow graph exists if there is a conditional branch at one of the vertices (the source of the dependence) that determines whether or not the other vertex (the sink of the dependence) is to be executed. Control dependence graph can be represented by either statically or dynamically.

A static control dependence graph can contain fewer vertices than the initial control flow graph since there is no need to keep vertices in the static control dependence graph if they (a) depend on the same vertex as their immediate dominator and (b) doesn’t have any vertices depending on them. That being said, a static control dependence graph can still be larger than the initial control flow graph in terms of edges since a single vertex can be control dependent on hundreds or even thousands of vertices.

A dynamic control dependence graph, like its static counterpart, can contain fewer vertices than the initial control flow graph. Although, every vertex that controls whether or not another executed vertex should be executed, and every executed vertex that is control dependent on another vertex will be part of the dynamic control dependence graph. The maximum number of edges in a dynamic control dependence graph is unbounded since
almost every transition of control flow from one vertex to another will yield a new edge from the executed vertex to the vertex that it is control dependent upon.

### 2.3.2 Data Dependency Graphs

In this section we describe how DDGs are represented, starting with the logical graph representation. We then briefly outline our physical representation, and discuss how graphs are traversed.

Data dependence analysis determines what the constraints are on how a piece of code can be reorganized. Data dependences are constraints on the order in which statements may be execute. Data dependencies may be represented using a directed acyclic graph (DAG). Logical graph representation. Let $I$ and $J$ be two instructions in a program, and $i$, $j$ be execution instances of $I$ and $J$, respectively. $i$ has a data dependency on $j$ if $i$ uses data defined by $j$.

In the classical graph representation used in early works, such as, instruction instances are represented by vertices, and data or control dependencies are represented by outgoing edges from vertices. We instead use a graph representation similar to the one proposed by Zhang and Gupta in [8]. In their graphs, each static instruction is represented by a vertex, and data or control dependencies are represented by adding labeled edges between vertices. The number of vertices is thus bounded by the size of the program, while the number of edges per vertex is determined by the length of execution. The label of an edge identifies the instruction instances involved in the corresponding dependency. For example, a data dependency of $i$ on $j$ is represented by adding an edge from $I$ to $J$, labeled with instance($i$) and instance($j$) i.e. the use-instance of $I$ and the define-instance of $J$. 
2.4 Neural Network

Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. They interpret sensory data through a kind of machine perception, labeling or clustering raw input. The patterns they recognize are numerical, contained in vectors, into which all real-world data, be it images, sound, text or time series, must be translated.

Neural networks help us cluster and classify. You can think of them as a clustering and classification layer on top of the data you store and manage. They help to group unlabeled data according to similarities among the example inputs, and they classify data when they have a labeled dataset to train on.

Neural networks are typically organized in layers. Layers are made up of a number of interconnected ‘nodes’ which contain an ‘activation function’. Patterns are presented to the network via the ‘input layer’, which communicates to one or more ‘hidden layers’ where the actual processing is done via a system of weighted ‘connections’. The hidden layers then link to an ‘output layer’ where the answer is output. Most ANNs contain some form of ‘learning rule’ which modifies the weights of the connections according to the input patterns that it is presented with. In a sense, ANNs learn by example as do their biological counterparts. The delta rule is often utilized by the most common class of ANNs called ‘backpropagational neural networks’ (BPNNs). Back-propagation is an abbreviation for the backwards propagation of error.

With the delta rule, as with other types of back-propagation, ‘learning’ is a supervised process that occurs with each cycle or ‘epoch’ (i.e. each time the network is presented
with a new input pattern) through a forward activation flow of outputs, and the backwards
error propagation of weight adjustments. More simply, when a neural network is initially
presented with a pattern it makes a random 'guess' as to what it might be. It then sees
how far its answer was from the actual one and makes an appropriate adjustment to its
connection weights. Back-propagation performs a gradient descent within the solution’s
vector space towards a 'global minimum' along the steepest vector of the error surface. The
global minimum is that theoretical solution with the lowest possible error. The error surface
itself is a hyper-paraboloid but is seldom 'smooth'. Indeed, in most problems, the solution
space is quite irregular with numerous 'pits' and 'hills' which may cause the network to
settle down in a 'local minum' which is not the best overall solution. Since the nature of
the error space can not be known a priori, neural network analysis often requires a large
number of individual runs to determine the best solution. Most learning rules have built-in
mathematical terms to assist in this process which control the 'speed' (Beta-coefficient) and
the 'momentum' of the learning. The speed of learning is actually the rate of convergence
between the current solution and the global minimum. Momentum helps the network to
overcome obstacles (local minima) in the error surface and settle down at or near the global
minimumum.

Once a neural network is 'trained' to a satisfactory level it may be used as an
analytical tool on other data. To do this, the user no longer specifies any training runs
and instead allows the network to work in forward propagation mode only. New inputs are
presented to the input pattern where they filter into and are processed by the middle layers
as though training were taking place, however, at this point the output is retained and no
backpropagation occurs. The output of a forward propagation run is the predicted model for the data which can then be used for further analysis and interpretation.

It is also possible to over-train a neural network, which means that the network has been trained exactly to respond to only one type of input; which is much like rote memorization. If this should happen then learning can no longer occur and the network is referred to as having been "grandmothered" in neural network jargon. In real-world applications this situation is not very useful since one would need a separate grandmothered network for each new kind of input.

However they work very well for: capturing associations or discovering regularities within a set of patterns; where the volume, number of variables or diversity of the data is very great; the relationships between variables are vaguely understood; or, the relationships are difficult to describe adequately with conventional approaches.

There are many advantages and limitations to neural network analysis and to discuss this subject properly we would have to look at each individual type of network, which isn't necessary for this general discussion.

Backpropagational neural networks (and many other types of networks) are in a sense the ultimate 'black boxes'. Apart from defining the general architecture of a network and perhaps initially seeding it with a random numbers, the user has no other role than to feed it input and watch it train and await the output. In fact, it has been said that with backpropagation, "you almost don't know what you're doing". Some software freely available software packages (NevProp, bp, Mactivation) do allow the user to sample the networks 'progress' at regular time intervals, but the learning itself progresses on its own.
The final product of this activity is a trained network that provides no equations or coefficients defining a relationship (as in regression) beyond its own internal mathematics. The network 'IS' the final equation of the relationship.

Backpropagational networks also tend to be slower to train than other types of networks and sometimes require thousands of epochs. If run on a truly parallel computer system this issue is not really a problem, but if the BPNN is being simulated on a standard serial machine (i.e. a single SPARC, Mac or PC) training can take some time. This is because the machines CPU must compute the function of each node and connection separately, which can be problematic in very large networks with a large amount of data. However, the speed of most current machines is such that this is typically not much of an issue.
2.4.1 Siamese Architecture

Siamese networks are a special type of neural network architecture. Instead of a model learning to classify its inputs, the neural networks learns to differentiate between two inputs.

A Siamese networks is comprised of two identical neural networks, each taking one of the two input. The last layers of the two networks are then fed to a loss function, which calculates the similarity between the two input. These two sister networks, which are identical neural networks, with the exact same weights. Each input in the input pair is fed to one of these networks. The networks are optimized using a contrastive loss function. The objective of the siamese architecture is not to classify inputs, but to differentiate between them. So, a classification loss function (such as cross entropy) would not be the best fit. Instead, this architecture is better suited to use a contrastive function. Intuitively, this function just evaluates how well the network is distinguishing a given pair of images. Similarity learning with deep CNNs is typically addressed using Siamese architectures.

For this domain, we employ large siamese convolution neural networks which a) are capable of learning generic image features useful for making predictions about unknown class distributions even when very few examples from these new distributions are available; b) are easily trained using standard optimization techniques on pairs sampled from the source data; and c) provide a competitive approach that does not rely upon domain-specific knowledge by instead exploiting deep learning techniques.

Siamese nets were first introduced in the early 1990s by Bromley and LeCun to solve signature verification as an image matching problem. A siamese neural network con-
sists of twin networks which accept distinct inputs but are joined by an energy function at the top. This function computes some metric between the highest level feature representation on each side. The parameters between the twin networks are tied. Weight tying guarantees that two extremely similar images could not possibly be mapped by their respective networks to very different locations in feature space because each network computes the same function. Also, the network is symmetric, so that whenever we present two distinct images to the twin networks, the top conjoining layer will compute the same metric as if we were to present the same two images but to the opposite twins.
Chapter 3

Approach

In this section, we will discuss the major ideas which have been used to solve the binary code similarity embedding problem. From now onwards we will assume that binary code of a function $f$ will be represented by its graph form $g$. These two terms “binary code” and “graph form” will be used interchangeably to describe the function. We designed the embedding mapping $\phi$ as a neural network. Since the input to the Neural Network is a graph, we will use previous graph embedding networks models implemented by other researchers as baseline and modify to address our problem. In Dai et al.’s work [9], the graph embedding network was mainly designed to solve a classification problem, to train such model supervised data is necessary. Which for our case is not possible since our code similarity embedding problem is not a classification problem. Thus existing approaches cannot be applied directly to our problem, and we need to come up with a novel approach to modify the existing graph embedding networks which can be trained for our data to solve the similarity detection problem.
To tackle this challenge, we propose a new learning approach. The idea behind it was to modify the predictive nature of graph embedding networks $\phi$ such that it can be trained to perform well in differentiating between two given graph form of functions. In particular, we designed Siamese architecture [5] and embedded the graph embedding network based on Structure2vec [9] into it to solve our problem. A Siamese architecture takes two functions as its input, and produces the similarity score as the output. This enables our designed model to be trained end-to-end without using any heuristics to generate embeddings of function but to train only with graph form $g_1, g_2$ as input and the ground truth $\pi(f_1, f_2)$ as output. We explain more details on this overall architecture and training in Section 3.4.

As other Neural network training siamese architecture requires a very big amount of data which contains large number of similar and dissimilar function. A different policy was being used to identify between equivalent and inequivalent functions because we needed a ground truth to train our model. So, same binary functions compiled from similar binaries but with different optimization level are considered equivalent and other functions are considered as inequivalent. Using this policy we were able to generate ample data which can be used to train and test our model.

### 3.1 Raw Feature Extraction

One the first step toward training our model was to appropriately generate graph form of different binary functions. For our project we will be comparing Control Flow graph based model with Program dependency graph based model. In the following section I will
explain in detail how the features are extracted for each function to generate useful data for our graph embedding model.

3.2 Graph Embedding Network

Our graph embedding network is adapted from **Structure2vec**, by Dai et al. [9]. Denote an AG as $g = <V, E>$ where $V$ and $E$ are the sets of vertices and edges respectively; furthermore, each vertex $v$ in the graph may have additional features $x_v$ which can correspond to AG if use are control flow based vectorization or APDG if were are using

---

**Figure 3.1**: An example of a code graph on Function `dtls1_process_heartbeat` (Heartbleed vulnerability).
program dependency based vectorization. We will be treating $x_v$ as general representation and will call as Attributed Graph(AG). The graph embedding network will first compute a $p$ dimensional feature $\mu_v$ for each vertex $v \in V$, and then the embedding vector $\mu_g$ of $g$ will be computed as an aggregation of these vertex embeddings. That is $\mu_g := A_{v\in V}(\mu_v)$, where $A$ is an aggregation function, i.e., summation or average. In this work, we choose $\mu_g = \Sigma_{v\in V}(\mu_v)$ and leave the exploration of using other aggregation functions as future work.

In the following, we first explain more details about the generic graph embedding network, and then present the variants instantiated specifically for our AG embedding problem.

**Basic Structure2vec Approach:** Structure2vec is inspired by graphical model inference algorithms where vertex-specific features $x_v$ are aggregated recursively according to graph topology $g$. After a few steps of recursion, the network will produce a new feature

![Graph Embedding Network Overview](image1)

![One Layer (Iteration) of the Graph Embedding Network](image2)

Figure 3.2: Graph Embedding Network.
representation (or embedding) for each vertex which takes into account both graph characteristics and long-range interaction between vertex features. More specifically, we denote $N(v)$ as the set of neighbors of vertex $v$ in graph $g$. Then one variant of the Structure2vec network will initialize the embedding $\mu_v^{(0)}$ at each vertex as 0, and update the embeddings at each iteration as

$$\mu_v(t + 1) = F(x_v, \sum_{u \in N(v)} \mu_u^{(t)}), \forall v \in V$$

In this fixed-point update formula, $F$ is a generic nonlinear mapping which we will specify our choice later. Based on the update formula, one can see that the embedding update process is carried out based on the graph topology, and in a synchronous fashion. A new round of embedding sweeping across the vertices will start only after the embedding update for all vertices from the previous round has finished. It is easy to see that the update also defines a process where the vertex features $x_v$ are propagated to the other vertices via the nonlinear propagation function $F$. Furthermore, the more iterations one carries out the update, the farther away a vertex feature will propagate to distant vertices and get aggregated nonlinearly at distant vertices. In the end, if one terminates the update process after $T$ iterations, each vertex embedding $\mu_v^{(T)}$ will contain information about its $T$-hop neighborhood determined by both graph topology and the involved vertex features.

Instead of manually specifying the parameters in the nonlinear mapping $F$, we learn these parameters. To train a Structure2vec model which is originally designed for a classification problem, previous work requires a ground truth label for every input graph $g$ to indicate which class it belongs to. Then the model is linked with a Softmax-layer, so that the entire model can be trained end-to-end by minimizing the cross-entropy loss.
As discussed above, this approach is not applicable to our case since our problem is not a classification problem.

**Our Parameterization for $F$** : We now discuss our parameterization of $F$ using a neural network. Figure 3 visualizes our network architecture. In particular, we design $F$ to have the following form

$$F(x_v, \sum_{u \in N(v)} \mu_u) = \tanh(W_1 x_v + \sigma(\sum_{u \in N(u)} \mu_u))$$

where $x_v$ is a $d$-dimensional vector for graph node (or basic-block) level features, $W_1$ is a $d \times p$ matrix, and $p$ is the embedding size as explained above. To make the nonlinear transformation $\sigma(.)$ more powerful, we will define $\sigma$ itself as an $n$ layer fully-connected neural network:

$$\sigma(l) = P_1 \times \text{ReLU}(P_2 \times \ldots \text{ReLU}(P_n l))$$

where $P_i (i = 1, \ldots, n)$ is a $p \times p$ matrix. We refer to $n$ as the embedding depth. Here, ReLU is the rectified linear unit, i.e., ReLU($x$) = max {0, $x$}

### 3.2.1 Learning Parameters Using Siamese Architecture

In this section, we explain our design of the overall network architecture to train a graph embedding for similarity detection. In particular, we use the Siamese architecture combined with the graph embedding Structure2vec network. The Siamese architecture uses two identical graph embedding networks, i.e., Structure2vec, which join at the top. Each graph embedding network will take one AG $g_1$ ($i = 1, 2$) as its input and outputs the embedding $\phi(g_i)$. The final outputs of the Siamese architecture is the cosine distance of the two embeddings. Further, the two embedding networks share the same set of parameters; thus
during training the two networks remain identical. The overall architecture is illustrated in Figure 2.1.

Given a set of $K$ pairs of AGs $< g_i, g'_i >$, with ground truth pairing information $y_i \in \{+1, -1\}$, where $y_i = +1$ indicates that $g_i$ and $g'_i$ are similar, i.e. $\pi(g_i, g'_i) = 1$, or $y_i = 1$ otherwise. We define the Siamese network output for each pair as

$$\text{Sim}(g, g') = \cos(\phi(g), \phi(g')) = \frac{< \phi(g), \phi(g') >}{\|\phi(g)\| \cdot \|\phi(g')\|}$$

where $\phi(g)$ is produced by graph embedding network. Then to train the model parameters $W_1, P_1, ..., P_n$, and $W_2$, we will optimize the following objective function

$$\min \sum_{i=1}^{K} (\text{Sim}(g_i, g'_i) - y_i)^2$$

We can optimize the above objective with stochastic gradient descent for parameter $W_1, P_1...P_n$ and $W_2$. The gradients of the parameters are calculated recursively according to the graph topology. In the end, once the Siamese network can achieve a good performance (e.g., using AUC as the measure), the training process terminates, and the trained graph embedding network can convert an input graph to an effective embedding suitable for similarity detection.

### 3.2.2 Hyperparameters

In this section, we evaluate the effectiveness of hyperparameters in the Gemini model. In particular, we examine the impact of the number of training epochs, embedding depth, embedding size, AG attributes, and number of iterations. The examination of impact of the number of training epochs is using the similarity validation set. We examine
other hyperparameters using the similarity testing set. On the entire similarity testing set, however, the AUC values are almost identical. Since we are more interested in the performance of our model on large graphs, the examinations of other hyper-parameters are using the large-graph subset of the similarity test set.

- **Number of epochs**: It is related to how many times the model is trained.

- **Embedding size**: Embedding size was actually size of the final output from the network embedding.

- **AG attributes**: It is the initial size of the vector.

- **Number of iterations**: We observe that the model achieves the best performance when the number of iterations $T$ is 5 or larger. This is reasonable, since in this dataset all graphs have a size larger than 10. It needs 5-hops to propagate local information on one vertex to most part of the graph.
Chapter 4

Evaluation

In this section, we evaluate our model search accuracy and its computation efficiency for finding same function. We use our dataset for which we already have ground truth to evaluate the accuracy of our trained model.

4.1 Implementation and Setup

Our whole model mainly involves graph information extraction and neural network model which uses that information for generating function embeddings. Graph information extraction method is also extension of work done by Xu et al. [31]. Since for both cases raw feature extraction which is preprocessing step is executed in similar way, hence the extracted information can be treated consistent with either case. Which relevant information is extracted and how it is extracted is already described in previous chapter. Both graph information extraction method and neural network implementation is done in python. Neural network is created using TensorFlow [1].
Measurement Baseline: There have been several previous works addressing the bug-search problem: discovRE [14], Multi-HM and Multi-k-HM [22], a centroid-based search [8], Genius [15] and Gemini. Xu et al. [31] have demonstrated that the Gemini approach is both more accurate and efficient than the other approaches. Therefore, in our evaluation, we consider that model as baseline approaches and their evaluation is also based on embedding function using control flow information and Neural network but have few shortcomings therefore it will be used for our comparison.

4.1.1 Dataset Generation

Sample is generated after the preprocessing of the binary to generate graph information of each function depending on which graph based approach was used. After preprocessing data was read and transformed into matrix form because it is to be send as input for the Neural Network.

The dataset used for neural network training and baseline comparison was OpenSSL. We compiled OpenSSL (version 1.0.1f and 1.0.1u) using GCC v5.4. The compiler optimization levels O0-O3 is used to generate x86 binary. It consists of binaries compiled from source code, so that we have ground truth. That is, we consider two AGs compiled from the same source code function to be similar, and those from different functions to be dissimilar. The statistics are presented in Table 2. During the split, we guarantee that no two binary functions compiled from the same source function are separated into two different sets among training and testing sets. In doing so, we can examine whether the pre-trained model can generalize to unseen functions.
4.1.2 Training Details

Our neural network model is first pre-trained using the dataset generated as mentioned above. Adam optimization algorithm [22] was used as base optimizer and learning rate of 0.0001 was used during whole training. Siamese model for trained 1000 epochs which was our default number of epochs. To train the neural network training data was created at each epoch. For each epoch, 3 AG $<g, g_1, g_2>$ were passed to the siamese neural network which contains 1 similar pair $<g, g_1>$ and 1 dissimilar pair $<g_1, g_2>$. First base source function $g$ was chosen at random. After that from the training set AG $<g_1>$ similar to base source function was searched and ground truth +1 is assigned to the pair. That’s the reason we are making sure that similar pair appears in same dataset after separation into training and testing. Similarly a dissimilar AG $<g_2>$ was chosen randomly since there is no conditions on choosing dissimilar function and -1 is assigned as their ground truth.

Since we are randomly choosing AG pair’s the training dataset varies every epoch which is very good thing to remove any bias form the training data. After every 100 epoch, we measure the loss and AUC on the currently trained set. During these 1000 training epochs, we save the model that achieved the least loss and will be further used for testing.

4.2 Accuracy

4.2.1 CFG-Based

This part covers the implemented model based Xu et al. [31] works. After preprocessing around 23000 ACFG’s were generated from 4 binaries of OpenSSL compiled for
Figure 4.1: Time taken while preprocessing the binary to generate dataset using their respective approaches.

Table 4.1: 1st Rank Accuracy for CFG based evaluation

<table>
<thead>
<tr>
<th>epochs</th>
<th>p=16</th>
<th>p=32</th>
<th>p=64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.6160</td>
<td>0.6207</td>
<td>0.6418</td>
</tr>
<tr>
<td>2000</td>
<td>0.6184</td>
<td>0.625</td>
<td>0.6346</td>
</tr>
<tr>
<td>3000</td>
<td>0.5846</td>
<td>0.6201</td>
<td>0.6489</td>
</tr>
</tbody>
</table>

different optimizations. After then whole Dataset was split disjointly into training, and testing sample space respectively. Results are shown in below tables. Table 4.1 Shows the accuracy to predict the similar function as 1st rank evaluated for different epochs and embedding size. Figure 4.1 shows the ROC curve for the whole data which is a better metric for measure the accuracy since the results are not always correct while measuring just 1st rank accuracy.
Table 4.2: Embedding generation Time for CFG based evaluation

<table>
<thead>
<tr>
<th>epochs</th>
<th>p=16</th>
<th>p=32</th>
<th>p=64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>836.42</td>
<td>917.59</td>
<td>921.92</td>
</tr>
<tr>
<td>2000</td>
<td>810.74</td>
<td>832.63</td>
<td>852.41</td>
</tr>
<tr>
<td>3000</td>
<td>798.46</td>
<td>816.20</td>
<td>843.68</td>
</tr>
</tbody>
</table>

Figure 4.2: ROC curves for different approaches evaluated on the dataset created from openssl binary.

4.2.2 PDG-Based

This is evaluation of work proposed by us. After preprocessing to generate PDG we were able to get around 21500 PDG’s. One of the reason for getting lesser number of PDG’s is because we ignored PDG’s only with 1 vertex for better dataset generation speed and also not providing substantial input information for the neural network. Table 4.3 and 4.4 shows respective evaluation with CFG based model.

Few things are pretty clear that this model does not perform much better than CFG based model. The reason being only relying on data-dependency and control-dependency
information other part of embedding is generated using Word2Vec [26] approach which does not capture the information perfectly. But advantage of this model is that it can be applied to obfuscated binary for finding similar functions where CFG-based evaluation will not be able perform at all.

Table 4.3: 1st Rank Accuracy for PDG based evaluation

<table>
<thead>
<tr>
<th>epochs</th>
<th>p=16</th>
<th>p=32</th>
<th>p=64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.5115</td>
<td>0.5382</td>
<td>0.5441</td>
</tr>
<tr>
<td>2000</td>
<td>0.5496</td>
<td>0.5623</td>
<td>0.5863</td>
</tr>
<tr>
<td>3000</td>
<td>0.5904</td>
<td>0.5929</td>
<td>0.5975</td>
</tr>
</tbody>
</table>

Table 4.4: Embedding generation Time for PDG based evaluation

<table>
<thead>
<tr>
<th>epochs</th>
<th>p=16</th>
<th>p=32</th>
<th>p=64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>895.38</td>
<td>996.19</td>
<td>897.86</td>
</tr>
<tr>
<td>2000</td>
<td>896.64</td>
<td>897.27</td>
<td>898.31</td>
</tr>
<tr>
<td>3000</td>
<td>898.04</td>
<td>898.91</td>
<td>899.28</td>
</tr>
</tbody>
</table>
Chapter 5

Conclusions

In this paper, we present an approach to generate binary embeddings for function using a neural network approach which can then further utilized in solving binary code similarity problem. Our evaluation shows although it is not able to perform upto par of Gemini [14] based model it can be used to solve the problem where other model are not able to perform at all.

There work has a lot of future extendability. One of the major work which can be done is to generate better instruction embedding currently is uses a very basic method which might not be able to capture the information completely. Other observation was made that this model losses lot of control dependency information with respect to CFG based approach, so a better method can be to include the missing control information with our approach which can definitely improve the ROC curve.
Bibliography


