A Dynamic Graph-Based Malware Classifier

by

Hossein Hadian Jazi

Bachelor of Software Engineering, Isfahan University of Technology, 2008

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor: Ali Ghorbani, PhD, Computer Science

Examining Board: Suprio Ray, PhD, Computer Science, Chair
John DeDourek, PhD, Computer Science
Brent Petersen, PhD, Electrical and Computer Engineering

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

March, 2016

©Hossein Hadian Jazi, 2016
Abstract

The anti-virus industry receives a sheer amount of new malware samples on a daily basis. The prevalence of new sophisticated instances, for most of which no signature is available, coupled with the significant growth of potentially harmful programs have made the adoption of an effective automated classifier almost inevitable.

Due to the vast majority of obfuscation techniques employed by the malware authors, extraction of a high-level representation of malware structure is an efficient way in this regard. High-level graph representations such as Function Call Graphs or Control Flow Graphs are able to represent the main functionality of a given sample in more abstract way. The graph-based approaches have mostly revolved around static analysis of the binary and share the common drawbacks of any static-based approaches. For example, generating a graph from a packed executable does not reflect the real structure of the code at all.

In addition to the type of analysis, the scalability of these approaches is also affected by the employed graph comparison algorithm. Full graph comparison
is by itself an NP-hard problem. Approximated graph comparison algorithms such as \textit{Graph Edit Distance} have been commonly studied in the field of graph classification.

To address the two major weaknesses involved with the current graph-based approaches, we propose a dynamic and scalable graph-based malware classifier. At the time of this proposal, this is the first attempt to generate and classify dynamic graphs. In spite of providing more accurate graphs, dynamic analysis leads to the generating larger graphs, and aggravating the problem of comparison measurement. To address this problem we modify an existing algorithm called \textit{Simulated Annealing} to reduce computational complexity.

To have a reasonable estimation of the effectiveness, our proposed system is compared against Classy, which is the state-of-the-art graph-based system. Our results show that proposed classifier is able to outperform Classy by an average classification accuracy of 94\%, 4\% false positive rate, and leaving only 2\% of samples unlabeled.
Dedication

This thesis work is dedicated to my wife, Elaheh Samani, who has been a constant source of support and encouragement during the challenges of graduate school and life. I am truly thankful for having you in my life. This work is also dedicated to my parents who have always loved me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve.
Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Ali Ghorbani for the continuous support of my master study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.
# Table of Contents

Abstract  
Dedication  
Acknowledgments  
Table of Contents  
List of Tables  
List of Figures  
Abbreviations  

1 Introduction  
  1.1 Introduction  
  1.2 Summary of Contributions  
  1.3 Thesis Organization  

2 Literature Review  
  2.1 Graph Representations  

vi
2.1.1 Function Call Graphs (FCGs) .................................. 9
2.1.2 Control Flow Graphs ........................................... 12
2.1.3 Hybrid Graphs .................................................. 13
2.2 Binary Unpacking ................................................... 13
2.3 Graph Matching .................................................... 15
  2.3.1 Exact Matching ............................................... 16
  2.3.2 Inexact Matching ............................................. 22
    2.3.2.1 Graph Edit Distance ................................. 23
    2.3.2.2 Other Inexact Graph Matching Techniques .. 26
2.4 Analyzers .......................................................... 28
  2.4.1 Static Analyzers ............................................. 29
  2.4.2 Dynamic Analyzers ......................................... 30
2.5 Detection .......................................................... 35
2.6 Concluding Remarks .............................................. 36

3 Proposed System ................................................... 39
  3.1 Overview ......................................................... 40
  3.2 Graph Generation ............................................... 40
    3.2.1 Dynamic Analyzer ...................................... 41
  3.2.2 Preprocessor ................................................ 42
  3.2.3 Graph Generator ........................................... 43
  3.3 Graph Matching ................................................ 47
    3.3.1 GED Calculator ............................................ 47
List of Tables

2.1 Overview of graph-based malware classification methods . . . 37

5.1 Distribution of Malware Types in the Dataset . . . . . . . . 72
5.2 Distribution of Packers Types in the Dataset . . . . . . . . . 72
5.3 Size of the output of each phase comparison . . . . . . . . . 78
5.4 Extracted Graphs . . . . . . . . . . . . . . . . . . . . . . . . 79

List of Figures

2.1 An Example of Function Call Graph . . . . . . . . . . . . . 11
2.2 Control Flow Graph Example . . . . . . . . . . . . . . . . . 12
2.3 graph isomorphism . . . . . . . . . . . . . . . . . . . . . . . 18
2.4 Maximum common sub-graph . . . . . . . . . . . . . . . . . 21
2.5 A possible edit path between two graphs . . . . . . . . . . . 24
3.1 General overview of the proposed malware detection system . 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Dynamic Function Call Graph of Fareit Malware</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>Dynamic CFG of Fareit Malware in Assembly Instructions</td>
<td>48</td>
</tr>
<tr>
<td>3.4</td>
<td>Dynamic CFG of Fareit Malware in Intermediate Representation</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>Proposed System Component Diagram</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Proposed System Class Diagram</td>
<td>69</td>
</tr>
<tr>
<td>4.3</td>
<td>Proposed System Sequence Diagram</td>
<td>70</td>
</tr>
<tr>
<td>5.1</td>
<td>Accuracy considering different parameters value</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Rejected samples considering different parameters value</td>
<td>75</td>
</tr>
<tr>
<td>5.3</td>
<td>Graph extraction comparison</td>
<td>77</td>
</tr>
<tr>
<td>5.4</td>
<td>Preprocessing comparison</td>
<td>78</td>
</tr>
<tr>
<td>5.5</td>
<td>Time requirement comparison original SA</td>
<td>80</td>
</tr>
<tr>
<td>5.6</td>
<td>Time requirement comparison modified SA</td>
<td>80</td>
</tr>
<tr>
<td>5.7</td>
<td>Distribution of number of edges</td>
<td>82</td>
</tr>
<tr>
<td>5.8</td>
<td>Distribution of number of nodes</td>
<td>82</td>
</tr>
<tr>
<td>5.9</td>
<td>Classification Performance</td>
<td>84</td>
</tr>
<tr>
<td>5.10</td>
<td>Comparison Study</td>
<td>86</td>
</tr>
</tbody>
</table>
List of Symbols, Nomenclature or Abbreviations

CFG    Control Flow Graph
FCG    Function Call Graph
GED    Graph Edit Distance
MCS    Maximum Common Sub-graph
PE     Portable Executable
BAP    Binary Analysis Platform
DECAF  Dynamic Executable Code Analysis Framework
AIF    Assembly Instruction File
FCLF   Function Call Log File
Chapter 1

Introduction

1.1 Introduction

Malicious software (Malware) is referred to any software that is used to disrupt computer operation to gather sensitive information, or gain access to private computer systems. The damage caused by malware can range from a minor increase in the outgoing traffic to a complete network breakdown and loss of critical data.

The rapid increase in the number and diversity of malware, make their analysis and identification process difficult. According to McAfee Labs Threats Report [55], the total number of unique malicious codes exceed 433 million in the second quarter of 2015 among which 47 million were new. Therefore, given such a large volume of malware, it is of paramount importance to quickly and accurately analyze malicious executables in an automated way.
The signature-based method is the most common way used by anti-virus products to determine if a sample is indeed malicious. It performs well when facing known malicious executables. However, the process of generating signature is currently done through manual analysis, which is expensive, time-consuming and error-prone. Moreover, since this method ignores program’s functionality, it can be easily crippled by obfuscation techniques such as code re-ordering, routine re-ordering, self-mutation, and code-obfuscation techniques [19].

Due to the vast majority of obfuscation techniques employed by malware authors, the extraction of a high-level representation of malware structure is required. Control flow graphs (CFGs) and function call graphs (FCGs) are the most common abstract representations of an executable. These graphs provide distinctive characteristics of a binary that is identifiable over strains of malware variants [16]. These graphs represent the execution paths a program may take. The FCG represents the inter-procedural control flow of a program while the CFG represents intra-procedural control flow.

Both CFGs and FCGs have been widely used as basic components of most malware detection approaches [11, 16, 42]. However, these methods suffer from the following limitations:

1. They resort to static analysis to generate CFG and FCG graphs mostly by employing PE-Explorer [79] or IDA Pro disassemblers [66]. The static analysis can easily be bypassed in obfuscated cases such as using packers. To alleviate this drawback some of the approaches [52, 49] have
used unpacker tools to remove the obfuscated layer of the executable before disassembling it. However, most of them are limited to a fixed set of known packers or are restricted by the fidelity of the emulation environment.

2. None of the approaches have used recent binary analysis platforms [10, 75] to avoid the limitation of disassemblers, which cannot make all side effects of assembly instructions explicit. In assembly language, a function or expression is said to have a side effect if it modifies some state or has an observable interaction with calling functions or the outside world. In the presence of side effects, a program’s behaviour may depend on its execution history. To make all side effects explicit, the assembly instructions need to be converted to Intermediate language (IL). IL enables all subsequent analyses to be written in a syntax-directed fashion.

3. Scalability of the approaches is affected by the employed graph comparison algorithm. The techniques used to compare graphs such as exact graph matching methods are only suitable for comparing graphs that have small number of nodes. Unfortunately, most of the graph comparison problems, including full-graph comparison and computing the largest common sub-graph, are computationally hard, which makes the scalability of a system questionable. As a result it is essential to use an approximation algorithm to compute the similarity between graphs
to make the method scalable.

4. The static approaches [28, 21, 15, 71, 56, 88, 49] cannot handle on-line streams of malicious executables due to their off-line classification or clustering algorithms.

In this work, motivated by the trends of large-scale threats, we propose a solution to automatically detect malware variants using the dynamic graphs. To the best of our knowledge, our approach is the first attempt that employs a dynamic graph-based approach to circumvent the shortcomings of static analysis.

Graph comparison is done by computing the graph edit distance (GED) of pairs of graphs. We modified the simulated annealing algorithm [52] that can improve the classification accuracy while maintaining low computational complexity. Moreover, we adopted the stream clustering algorithm proposed in [67] to cluster a stream of graphs.

To demonstrate the applicability of the system, we compare our classification method with most recent conventional approach [52] using a reference dataset consisting of more than 50,000 unique executables from different malware families.

### 1.2 Summary of Contributions

The main purpose of this thesis is to develop a dynamic graph-based malware classification system. The contributions of this thesis can be summarized as
follows:

- Employing dynamic analysis to generate FCGs and CFGs.
- Devising a new algorithm to generate dynamic FCGs.
- Improving the simulated annealing algorithm to compute similarity between graphs by employing stochastic beam search concept [64]. It improves the accuracy of the classification while maintaining low computational complexity.
- Developing an on-line stream clustering algorithm for clustering streams of FCGs.
- Performing experimental evaluation on a large set consisting of diverse malware samples. Our comparative analysis shows that our dynamic graph-based method performs better in detecting malware samples compared to static graph-based methods even when they use unpacker tools. This also proves that even though dynamic analysis only traces one execution path of binary, it is sufficient enough to distinguish malicious programs.

The experimental results demonstrate the effectiveness of our classifier by an average classification accuracy of 94%, 4% false positive, and leaving only 2% samples unlabeled. This result highly outperform the existing static graph-based classification methods. The results prove that such a system
is extremely valuable and under circumstances reliable enough to provide automatic classifications.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 reviews graph based malware classification approaches, which employ CFG or FCG and discuss their drawbacks. Since all of the methods use static analysis we also provide an overview of different techniques that can be used to generate dynamic call graphs. Moreover, we provide a summary of different graph matching techniques and consider their advantages and disadvantages. Finally we provide an overview of different detection methods which are suitable for graph clustering.

Chapter 3 provides a general overview of the system, followed by a description of each module. It explains how an input binary sample is passed through the system to be classified as an existing or a new malware family. First, the process of graph extraction based on dynamic analysis is introduced. Then, the employed graph matching algorithms are discussed. Finally, the algorithm for incremental classification of received samples is explained.

Design and implementation criteria of the proposed framework are described in Chapter 4. This Chapter introduces the main components and classes implemented in our framework and lists their main functions. Chapter 5 reports the evaluation dataset, metrics, and comparative experimental results of the
proposed framework with existing malware classification approach. Finally, Chapter 6 concludes the thesis by discussing its contributions, limitations and possible improvements to the work done in this thesis.
Chapter 2

Literature Review

2.1 Graph Representations

In general, Graph representation of an object is able to represent properties of the object and binary relationships at the same time. In other words, graph offers great flexibility in terms of object representation by describing parts of an object with nodes and binary relations among the parts with edges.

The graph representation symbolizes the main functionality of an executable, which is distinguishable among other executables. Due to the discriminative power of graph representations, they have become popular in malware classification. In the following subsections we will consider FCGs and CFGs as the most common graph representations.
2.1.1 Function Call Graphs (FCGs)

A FCG is a directed graph that represents calling relationships between functions in a program. Vertices, representing the functions a program is composed of, are interconnected through directed edges that symbolize function calls. A vertex can represent either one of the following two types of functions:

- Local functions, implemented by the program author.
- External functions: system and library function calls.

Local functions: the most frequently occurring functions in any program, are written by the program author. External functions, such as system and library calls, are stored in a library as part of an operating system. Contrary to local functions, external functions never invoke local functions. Analogous to Hu et al. work [43], FCGs are formally defined as follows:

**FCG:** A FCG is a directed graph $G$ with vertex set $V=V(G)$, representing the functions, and edge set $E=E(G)$, where $E(G) \subseteq V(G) \times V(G)$, in correspondence with the function calls. For a vertex $v \in V$, two functions are defined $V_n(v)$ and $V_f(v)$, which provide respectively the function name and function type of the function represented by $v$. The function type $t \in \{0,1\}$ can either be a local function (0), or an external function (1).

Figure ?? shows an example of FCG which was generated from IDA pro [66]. The external and internal functions are represented by the nodes with pink
and black colours, respectively. The node with the green colour represents the starting point of FCG.

FCGs can be extracted by dynamic or static analysis. A dynamic FCG is the record of an execution of a program, e.g., as output by a profiler. Thus, a dynamic FCG can be exact, but only describes one run of the program. A static FCG is a call graph intended to represent every possible run of the program. The exact static FCG is an undecidable problem, so static graph algorithms are generally based on approximations. That is, every call relationship that occurs is represented in the graph, and possibly also some call relationships that would never occur in actual runs of the program.

Since FCG provides an abstraction of program, which are able to represent its main functionally, it has been widely used to identify malicious programs [28, 21, 15, 71, 56, 88, 43, 49, 52]. However, these approaches have mostly revolved around static analysis of the binary and share the common drawbacks of any static-based approaches. The main weakness of static analysis is that the code analyzed may not necessarily be the code that is actually executed, e.g. when files are packed or a third party code is being downloaded and executed. Also, malware can employ a wide range of obfuscation mechanisms that make static analysis ineffective. Most recent researches have widely demonstrated the inefficiency of the static-based methods in the analysis of sophisticated malware. In general, when the malware author employs obfuscation techniques, the extracted graph from static analysis does not reflect the real behaviour of an executable.
Figure 2.1: An Example of Function Call Graph
2.1.2 Control Flow Graphs

CFG represents the paths that an application code might use during its execution. CFGs have been used in the analysis of software and have been studied for many years [46, 59, 77]. CFG can be considered as a directed graph, where each node represents a statement of the program and each edge represents control flow between the various statements. Examples of these statements include copy statements, assignments and branches. Figure 2.2 shows an example of a CFG.

CFG is used in detecting metamorphic malware [6, 11, 4, 92, 76, 81, 1] to alleviate byte sequence-based methods limitations since their syntactic signatures ignore the program semantics. CFG can capture the nature of an executable and its functionality, and therefore is distinguishable across
variants of samples. Nevertheless, the CFG-based approaches suffer from the same limitation of FCGs since they resort to static analysis to extract graphs.

2.1.3 Hybrid Graphs

CFG or FCG itself do not contain enough information about malicious samples [24, 25]. As a result, there is a need to improve the graphs by either merging them together or adding more information to them. Some of the approaches merge CFG, FCG and, register flow graph of a given binary [2, 9] while others tried to enrich the generated graphs by employing statistical information of dependency assembly instructions and API calls [24, 25].

Since these approaches are revolved around static analysis, they have the same limitation of CFGs and FCGs.

2.2 Binary Unpacking

Historically, malware have used self-modifying code to disguise its malicious intents and make static analysis more cumbersome. While such modifications were first performed by incorporating the self-modifying parts in the malware itself, more recent developments have led to packer tools. A packer program automatically transforms an executable into a syntactically different, but semantically equivalent, representation.
The packer creates the semantically equivalent representation by obfuscating or encrypting the original binary and stores the result as data in a new executable. An unpacker routine is prepended to the data, whose responsibility upon invocation lies in restoring (i.e., deobfuscating or decrypting) the data to the original representation. This reconstruction takes place solely in memory, which prevents leaking any unpacked versions of the binary to the disk. After unpacking, the control is handed over to the, now unpacked, original binary that performs the intended tasks. Polymorphic variants of a given binary can be automatically created by choosing random keys for the encryption. However, their unpacking routines are, apart from the decryption keys, largely identical. Therefore, while signatures cannot assess the threat of the packed binary, signature matching can be used to detect the fact that a packer program was used to create the binary. Metamorphic variants, in contrast to polymorphic binaries, can also mutate the unpacking routine, and may encumber detection even more.

According to Wei et al. [89], a large percentage of malicious software today comes in packed form. Moreover, malware instances that apply multiple recursive layers of packers are becoming more prevalent. Due to this increase, most of the graph-based approaches are ineffective [1, 2, 4, 6, 11, 15, 21, 28, 71, 81] since they rely on static analysis for graph extraction process. To combat this issue, some of the works employ unpacker tools [9, 16, 42, 49, 52, 56]. In their methods, before a call-graph is extracted, the binary is first examined to determine whether it is packed or protected. To detect whether
a file is packed, they mostly use pattern-matching tools such as PEiD [47], which contains signature databases for a series of known packers. Once a packer has been identified, one needs to apply the appropriate unpacker. This approach is fast and works well for the vast majority of known packers, but the primary limitation of pattern-matching unpackers is that they are ineffective when facing unknown/new packer.

An alternative solution is to use heuristics. Most of them can be considered as run-time unpacking tools and may require an isolated environment. In general, unpacking heuristics are of questionable reliability and can be evaded. For example, they may fail under the presence of anti-virtualization and anti-emulator techniques. Finally, more than one packing technique may be applied simultaneously.

2.3 Graph Matching

Detecting malware through the use of FCGs requires means to compare FCGs mutually, and ultimately, means to distinguish FCGs representing benign programs from call graphs based on malware samples. This process can be done by employing graph matching.

The process of evaluating the similarity of two graphs is commonly referred to as graph matching. The overall aim of graph matching is to find a correspondence between the nodes and edges of two graphs that satisfies some, more or less, stringent constraints. That is, by means of the graph matching
process similar substructures in one graph are mapped to similar substructures in the other graph. Based on this matching, a dissimilarity or similarity score can eventually be computed indicating the proximity of two graphs.

Graph matching has been the topic of numerous studies in computer science over the last decades. Roughly speaking, graph matching techniques can be classified into two categories, namely, exact matching and inexact matching. In the former case, for a matching to be successful, it is required that a strict correspondence is found between the two graphs being matched, or at least among their sub-parts. In the latter approach this requirement is substantially relaxed, since also matchings between completely non-identical graphs are possible. That is, inexact matching algorithms are endowed with a certain tolerance to errors and noise, enabling them to detect similarities in a more general way than the exact matching approach. Therefore, inexact graph matching is also referred to as error-tolerant graph matching. In the following subsections we will consider these techniques in details.

2.3.1 Exact Matching

Exact matching algorithms try to determine whether two graphs, or at least part of them, are identical in terms of structure and labels. Adjacency matrix is the most common way that has been used to describe the structure of a graph. In general, there is no unique canonical order for the vertices and also the edges of a graph. As a result, there exist \( n! \) different adjacency matrix for a graph with size \( n \) \( (n \) is the number of vertices) because there are \( n! \)
possibilities to order the graph nodes. Consequently, for checking two graphs for structural identity, we cannot simply compare their adjacency matrices. The identity of two graphs is commonly established by defining a function, called graph isomorphism mapping one graph to another one.

**Graph Isomorphism:** Let us consider two graphs denoted by \( G_1 = (V_1, E_1, \mu_1) \) and \( G_2 = (V_2, E_2, \mu_2) \) where \( V \) is vertex set, \( E \) is edge set and \( \mu \) is vertices labels set. A graph isomorphism is a bijective function \( \phi : V_1 \rightarrow V_2 \) satisfying:

- \( \mu_1 (u) = \mu_2 (\phi(u)) \) for all nodes \( u \in V_1 \)
- for each edge \( e_1 = (u, v) \in E_1 \), there exists an edge: \( e_2 = (\phi(u), \phi(v)) \in E_2 \)
- for each edge \( e_2 = (u, v) \in E_2 \), there exists an edge: \( e_1 = (\phi^{-1}(u), \phi^{-1}(v)) \in E_1 \)

Two graphs are called isomorphic if there exists an isomorphism between them. Obviously, isomorphic graphs are identical in both structure and labels. That is, a one-to-one correspondence between each node of the first graph and each node of the second graph has to be found such that the edge structure is preserved and node labels are consistent. In Figure 2.3 two isomorphic graphs are shown in which graph (b) is isomorphic to (a), and graph (c) is isomorphic to a sub-graph of (a). Node labels are indicated by different colours.
Unfortunately, no polynomial runtime algorithm is known to exist for dealing with the problem of graph isomorphism \cite{34, 33}. That is, in the worst case, the computational complexity of any of the available algorithms for graph isomorphism is exponential in the number of nodes of the two graphs.

Closely related to graph isomorphism is sub-graph isomorphism, which can be seen as a concept describing sub-graph equality. A sub-graph isomorphism is a weaker form of matching in terms of requiring only that an isomorphism holds between a graph $G_1$ and a sub-graph of $G_2$. Intuitively, sub-graph isomorphism is the problem to detect if a smaller graph is identically present in a larger graph.

**Sub-graph Isomorphism**: Let $G_1 = (V_1, E_1, \mu_1)$ and $G_2 = (V_2, E_2, \mu_2)$. An injective function $f : V_1 \rightarrow V_2$ from $G_1$ to $G_2$ is a sub-graph isomorphism if there exists a sub-graph $G \subseteq G_2$ such that $f$ is a graph isomorphism between $G_1$ and $G$.

The process of graph matching primarily aims at identifying corresponding substructures in the two graphs under consideration. Through the graph
matching procedure an associated similarity or dissimilarity score can be easily inferred. In view of this, graph isomorphism as well as sub-graph isomorphism provide us with a basic similarity measure, which is 1 (maximum similarity) for (sub)graph isomorphic, and 0 (minimum similarity) for non-isomorphic graphs. Hence, two graphs must be completely identical, or the smaller graph must be identically contained in the other graph, to be deemed similar. Consequently, the applicability of this graph similarity measure is rather limited. Consider a case where most, but not all, nodes and edges in two graphs are identical. The rigid concept of (sub)graph isomorphism fails in such a situation in the sense of considering the two graphs to be totally dissimilar. Due to this observation, the formal concept of the largest common part of two graphs is established.

**Maximum common sub-graph:** Let $G_1 = (V_1, E_1, \mu_1)$ and $G_2 = (V_2, E_2, \mu_2)$ be graphs. A common sub-graph of $G_1$ and $G_2$, $cs(G_1, G_2)$, is a graph $G = (V, E, \mu)$ such that there exist sub-graph isomorphisms from $G$ to $G_1$ and from $G$ to $G_2$. We call $G$ a maximum common sub-graph (MCS) of $G_1$ and $G_2$, $MCS(G_1, G_2)$, if there exists no other common sub-graph of $G_1$ and $G_2$ that has more nodes than $G$.

A maximum common sub-graph of two graphs represents the maximal part of both graphs that is identical in terms of structure and labels. Note that, in general, the maximum common sub-graph is not uniquely defined, that is, there may be more than one common sub-graph with a maximal number of nodes. A standard approach to computing maximum common sub-graphs is
based on solving the maximum clique problem in an association graph \([57, 60]\). The association graph of two graphs represents the whole set of possible node-to-node mappings that preserve the edge structure and labels of both graphs. Finding a maximum clique in the association graph, that is, a fully connected maximal sub-graph, is equivalent to finding a maximum common sub-graph.

Graph dissimilarity measures can be derived from the maximum common sub-graph of two graphs. Intuitively speaking, the larger a maximum common sub-graph of two graphs is, the more similar are the two graphs. For instance, in [13] such a distance measure is introduced, defined by:

\[
d_{MCS}(G_1, G_2) = 1 - \frac{|MCS(G_1, G_2)|}{\max\{|G_1|, |G_2|\}}
\]

Note that, in contrast to the maximum common subgraph of two graphs which is not uniquely defined, the \(d_{MCS}\) distance is defined uniquely. If two graphs are isomorphic, their \(d_{MCS}\) distance is 0; on the other hand, if two graphs have no part in common, their \(d_{MCS}\) distance is 1. It has been shown that \(d_{MCS}\) is a metric and produces a value in \([0, 1]\). In Figure 2.4, graph (e) is maximum common sub-graph of graph (a) and (b).

The remainder of this section serves as a brief literature review of different graph isomorphism and MCS approaches. Funabiki and Kitamichi design a two-stage discrete optimization approach for MCS. In the first stage, a greedy
search is performed to find an arbitrary common subgraph, after which the second stage executes a local search for a limited number of iterations to improve upon the graph discovered in stage one. Similarly to the Funabiki and Kitamichi approach [30], the authors of [86] also rely on a two-stage optimization procedure, however contrary to [30], their algorithm tolerates errors in the $MCS$ matching. A genetic algorithm approach to $MCS$ is given by Wagener and Gasteiger [83]. Finally, Bradde et al. [8] propose a distributed technique for $MCS$ based on message passing.

Several approaches have used graph isomorphism and MCS to measure similarity between malicious executables. BinHunt [32] provide a more thorough test of flowgraph similarity by soundly identifying the maximum common subgraph, but at reduced levels of performance and without application to malware classification. Identifying common subgraphs of fixed sizes can also indicate similarity and has better performance [54]. Park et al. [63] utilize a version of $MCS$ denoted as weighted $MCS$ to drive common malware behaviour. A weighted maximum common sub-graph (WMCS) of a set of
graphs $G$ is a sub-graph of maximal size for which there is an isomorphic sub-graph in all of the graphs.

### 2.3.2 Inexact Matching

Due to the intrinsic variability of the patterns under consideration and the noise resulting from the graph extraction process, it cannot be expected that two graphs representing the same class of objects are completely, or at least to a large part, identical in their structure. Moreover, if the node label alphabet $L$ is used to describe non-discrete properties of the underlying patterns, e.g. $L \subseteq \mathbb{R}^n$, it is most probable that the actual graphs differ somewhat from their ideal model. Obviously, such noise crucially hampers the applicability of exact graph matching techniques, and consequently exact graph matching is rarely used in real-world applications.

In order to overcome this drawback, it is advisable to endow the graph matching framework with a certain tolerance to errors. That is, the matching process must be able to accommodate the differences of the graphs by relaxing to some extent the underlying constraints. In the first part of this section the concept of graph edit distance is introduced to illustrate the paradigm of inexact graph matching. In the second part, several other approaches to inexact graph matching are briefly discussed.
2.3.2.1 Graph Edit Distance

Graph Edit Distance (GED) [12, 70] offers an intuitive way to integrate error-tolerance into the graph matching process and is applicable to virtually all types of graphs. Originally, edit distance has been developed for string matching [84] and a considerable amount of variants and extensions to the edit distance have been proposed for strings and graphs. The key idea is to model structural variation by edit operations reflecting modifications in structure and labeling. A standard set of edit operations is given by insertions, deletions, and substitutions of both nodes and edges. Note that other edit operations, such as merging and splitting of nodes [3], can be useful in certain applications.

GED, calculates the minimum number of edit operations required to transform graph $G_1$ into graph $G_2$. Given two graphs, the source graph $G_1$ and the target graph $G_2$, the idea of graph edit distance is to delete some nodes and edges from $G_1$, relabel some of the remaining nodes, and insert some nodes and edges in $G_2$, such that $G_1$ is finally transformed into $G_2$. A sequence of edit operations $e_1, \ldots, e_k$ that transform $G_1$ into $G_2$ is called an edit path between $G_1$ and $G_2$. In Figure 2.5 an example of an edit path between two graphs $G_1$ and $G_2$ is given. This edit path consists of three edge deletions, one node deletion, one node insertion, two edge insertions, and two node substitutions.

Since exact solutions for GED are computationally expensive, a large amount of research has been devoted to developing fast and accurate approximation
algorithms for these problems, mainly in the field of image processing and for bio-chemical applications [86]. In the following sections we will consider these approaches.

A survey of three different approaches to perform GED calculations is conducted by Riesen et. al. in [62, 68, 69]. They first give an exact GED algorithm using A* search, but this algorithm is only suitable for small graphs [62]. Next, A*-Beam search, a variant of A* search, which prunes the search tree more rapidly, is used. As is to be expected, the latter algorithm provides fast but suboptimal results. The last algorithm they survey uses Munkres bipartite graph matching algorithm as an underlying scheme. Benchmarks show that this approach, compared to the A*-search variations, handles large graphs well, without affecting the accuracy too much.

Justice and Hero [48] formulate the GED problem as a Binary Linear Program, but the authors conclude that their approach is not suitable for large graphs. Nevertheless, they derive algorithms to calculate the lower and upper bounds of the GED in polynomial time, which can be deployed for large graph instances as estimators of the exact GED. Inspired by Justice and Hero [48] approach, Zeng et al. [91] provide a new polynomial algorithm which finds
tighter upper and lower bounds for the GED problem. In the area of malware classification there exists a few works that have used GED to compare FCGs or CFGs. SMIT [43] is the first approach which employs GED in malware analysis. It identifies variants using minimum cost bipartite graph matching and the Hungarian algorithm, which finds an exact one-to-one vertex assignment with the goal of minimizing the total mapping cost, improving upon the greedy approach to graph matching. SIGMA [2] also used the minimum cost bipartite graph matching to calculate the similarity between their own graph representations. Kostakis et al [53] propose an adapted version of Simulated Annealing to compute GED. It is a local search algorithm which searches for a vertex mapping that minimizes the GED. This algorithm turns out to be both faster and more accurate than, for example, the algorithms based on Munkres\' bipartite graph matching algorithm as applied in Hu et al. approach [43]. Two steps can be distinguished in the Simulated Annealing algorithm for call graph matching. In the first step, the algorithm determines which external functions a pair of call graphs have in common. These functions are mapped one-to-one. Next, the remaining functions are mapped based on the outcome of the Simulated Annealing algorithm, which attempts to map the remaining functions in such a way that the GED for the call graphs under consideration is minimized. Simulated Annealing has also been used in several works [52, 49] to compute the GED. Elhadi et al. [23] employ a modified greedy approach that supports GED
metric to find the set of best paths from the data graph that match the set of query graph edges and construct the best sub-graph with high degree of similarity.

2.3.2.2 Other Inexact Graph Matching Techniques

Several other important classes of error-tolerant graph matching algorithms have been proposed. Among others, algorithms based on artificial neural networks, Relaxation Labeling, Spectral Decompositions, and Graph Kernels have been reported.

One class of error-tolerant graph matching methods employs artificial neural networks to classify directed acyclic graphs. For example Micheli [61] used the idea to represent the nodes of a graph in an encoding network. In this encoding network local transition functions and local output functions are employed, expressing the dependency of a node on its neighbourhood and describing how the output is produced, respectively. As both functions are implemented by feedforward neural networks, the encoding network can be interpreted as a recurrent neural network.

Another class of error-tolerant graph matching method employs relaxation labeling techniques. The basic idea of this particular approach is to formulate the graph matching problem as a labeling problem. Each node of one graph is to be assigned to one label out of a discrete set of possible labels, specifying a matching node of the other graph. During the matching process, Gaussian probability distributions are used to model compatibility coefficients measur-
ing how suitable each candidate label is. The initial labeling, which is based on the node attributes, node connectivity, and other information available, is then refined in an iterative procedure until a sufficiently accurate labeling, i.e. a matching of two graphs, is found. Wilson and Hancock [87] employed Bayesian consistency measure to derive a graph edit distance.

The general idea of spectral methods is that the eigenvalues and the eigenvectors of the adjacency or Laplacian matrix of a graph are invariant with respect to node permutation. Hence, if two graphs are isomorphic, their structural matrices will have the same eigendecomposition. The converse, i.e. deducing from the equality of eigendecompositions to graph isomorphism, is not true in general. However, by representing the underlying graphs by means of the eigendecomposition of their structural matrix, the matching process of the graphs can be conducted on some features derived from their eigendecomposition. The main problem of spectral methods is that they are rather sensitive structural errors, such as missing or spurious nodes. Moreover, most of these methods are purely structural, in the sense that they are only applicable to unlabeled graphs, or they allow only severely constrained label alphabets.

Kernel methods were originally developed for vectorial representations, but the kernel framework can be extended to graphs in a very natural way. A number of graph kernels have been designed for graph matching [31]. A seminal contribution is the work on convolution kernels, which provides a general framework for dealing with complex objects that consist of simpler parts [38]. Convolution kernels infer the similarity of complex objects from
the similarity of their parts.

A second class of graph kernels is based on the analysis of random walks in graphs. These kernels measure the similarity of two graphs by the number of random walks in both graphs that have all or some labels in common [7, 35]. Gartner et al. [35] show that the number of matching walks in two graphs can be computed by means of the product graph of two graphs, without the need to explicitly enumerate the walks. In order to handle continuous labels the random walk kernel has been extended by Borgwardt et al. [7]. This extension allows one to also take non-identically labeled walks into account.

A third class of graph kernels is given by diffusion kernels. The kernels of this class are defined with respect to a base similarity measure which is used to construct a valid kernel matrix [74]. This base similarity measure only needs to satisfy the condition of symmetry and can be defined for any kind of objects.

2.4 Analyzers

Malware analysis can be performed in two traditional ways: static or dynamic. Static analysis is the process of examining a sample without running it to find basic information, e.g. strings, imported/exported libraries or functions. While straightforward and fast, malware can thwart static analysis by employing obfuscation techniques such as code packing, dead-code insertion, and code integration. In dynamic analysis, a sample is executed in a pro-
ected environment, e.g. sandbox, and its actual behavior is captured in the form of API/system calls, or in an instruction dump. Dynamic analysis of malware is immune to most obfuscation techniques and has shown to be more effective in differentiating malware families. In this section we will consider the static analyzers and also dynamic analyzers that can be used to extract FCGs.

2.4.1 Static Analyzers

IDA Pro and PE-Explorer are the most popular disassembler tools that have been widely used in many research works to extract assembly instructions or generate graphs from a binary.

**IDA Pro**  The Interactive Disassembler Professional (IDA Pro) is a powerful disassembler used in many malware analysis and detection, reverse engineering and vulnerability analysis works. It supports several file formats, such as Portable Executable (PE), Common Object File Format (COFF), Executable and Linking Format (ELF). IDA Pro can disassemble an entire program and perform tasks such as function discovery, stack analysis, local variable identification, and much more. The primary limitation of this tool is that, it only inspects binaries statically, and therefore malware authors can easily thwart it by employing obfuscation techniques. Most of the graph-based approaches employ IDA Pro to extract graphs from binary [49, 52, 21, 28, 15, 9].
**PE-Explorer** PE-Explorer as an another disassembler tool decomposes Portable Executable (PE) and DLL files. It is less powerful than IDA pro, however [24] and [25] use this tool to extract the instructions of Windows malware binaries. PE-Explorer is a static analysis tool and suffers from the same limitation as that of IDA pro.

### 2.4.2 Dynamic Analyzers

All of the existing FCG based malware classification approaches generate FCGs by employing static analysis. However, there are several approaches that can be used as a base framework to generate dynamic call graphs or can generate dynamic call graphs directly. These kinds of dynamic graphs have never been used for malware analysis. Some of these employ source code instrumentation, while others work on executables. In this section, we focus specifically on approaches that work on executables.

**OS Integrated Tools** In this approach, instrumentation frameworks are built into operating system (OS) kernels. *DTrace*, an advanced dynamic tracing framework designed to improve the observability of software systems [14], is an example of this approach. Both Solaris and Mac OS have incorporated *DTrace* as a core component of their development and administration tools. *DTrace* enables users to observe the system by exporting various runtime probes, implemented and managed by providers. The fbt (Function Boundary Tracing) and pid providers support function tracing in the kernel and
user-space. These providers allow tracing of any function entries and exits by attaching a trap immediately before each call instruction. DTrace is notified when this trap hits and automatically executes the user-defined actions. Because DTrace can instrument programs with low overhead, it is suitable for production environments.

Although such approaches are powerful and high-performance, they are tightly integrated with kernels and therefore, can only work in the kernels that support such features. Consequently, such tools do not work in a large class of embedded devices because they rarely use operating systems with such support.

**OS Interface Tools** In this approach, tools are built to exploit OS and runtime interfaces to capture dynamic FCGs. As an example, ltrace or library trace is a debugging utility in Linux [37] that works with fork and clone system calls to perform function tracing. Currently, ltrace only intercepts the first function call to dynamically linked libraries. It traces neither the function calls between shared libraries nor statically linked function calls in programs. Moreover, ltrace only works in Linux.

To address some of these limitations, latrace extends ltrace to support tracing of dynamic function calls between shared libraries at runtime [29]. It is implemented on top of LD_AUDIT, which is the GNU dynamic linker audit feature. However, no dynamic library call can be traced if one of the shared libraries does not include a relocation Procedure Linkage Table (.REL.PLT)
in the ELF binary. Both ltrace and latrace can operate with low overhead.

**Dynamic Binary Instrumentation** Binary instrumentation can generate dynamic FCGs by inserting code snippets at the beginning of functions. However, in doing so, additional code is generated, which can result in some differences in runtime states when compared to native code with no instrumentation. As an example, *Pin* is an open-source binary instrumentation framework that has been widely used in debugging, profiling, and evaluating performance [58]. *Pin* provides several APIs so that developers can customize their own *Pin* tools to perform tasks such as counting executed instructions and collecting function call information [58]. Currently, *Pin* can instrument Linux, Mac OS X, and Windows executables for several architectures. Jalan and Kejariwal [45] proposed a framework enables extraction of dynamic FCGs by employing *Pin*.

Recent work by Hazelwood and Klauser [39] shows that the overhead of *Pin* ranges from 1.5 to 8 times slower than native execution. Moreover, since *Pin* runs in the user space it cannot capture the instructions running in kernel space, and therefore the generated FCGs would be imprecise.

Another example is *Valgrind*, an instrumentation framework that can be used to build dynamic analysis tools. It currently works in Linux and Mac OS X. One of the tools in *Valgrind* that can be used to generate dynamic FCGs is *Callgrind*. It is an extension of *Cachegrind*, a cache profiler. *Callgrind* augments *Cachegrind* with FCG information so that it can generate FCGs
for both statically and dynamically linked libraries [85]. The overhead of Callgrind ranges from 20 to 100 times slower than native execution. Also, similar to Pin it has the limitation of capturing kernel space instructions.

**Full System Simulators** Unlike typical instruction set simulators, which do not simulate I/O components, full-system simulators can be modeled to simulate complete computer systems with I/O components, bus interconnects, processors, and memory subsystems. Therefore, they provide virtual platforms that can run complex software systems (e.g., applications and OS kernels) without any modifications.

QEMU and Simics [5, 82] are the most popular full system emulators that have been used widely in many research works. Chen [18] uses Simics emulator to generate dynamic FCGs. Simics provides infrastructure for developers to model and use hardware devices in their simulations. The modeling process is fast, so engineers can have a new virtual platform up and running several months before the completion of the hardware prototype. As a commercial product, it also supports many advanced features and interfaces that developers can use to create their own instrumentation and dynamic analysis tools. Its execution overhead ranges from 3 times (for processor intensive applications) to 30 times (for I/O intensive applications) slower than native execution.

QEMU, on the other hand, is faster than Simics in terms of performance. It also has Trace Generation, which is a component that works in conjunction
with *DineroIV*, a memory reference tracing simulator, to generate execution traces and perform analysis [22]. However, *QEMU* lacks the capability to allow developers to model a full range of hardware devices.

In summary, full-system simulators provide an attractive platform to carry out dynamic FCG extractions for two reasons.

- **Non-intrusive Instrumentation of executables:** Instrumentation occurs at binary-level and without disturbing execution or affecting the virtualized state of a system. Therefore, it can simulate and profile systems accurately in the presence of instrumentation. Furthermore, these simulators can collect the exact profile data instead of relying on sampling or probability. Thus, the profiled information is more complete. For the problem we try to address, this is an important consideration.

- **Support more types of executables:** Full-system simulators support executables with or without operating systems. This is different from other approaches, which are operating system dependent (e.g., *Pin* can only work on Linux or Mac OS X binaries). Therefore, they can work in diverse applications and systems ranging from executables running in stand-alone embedded devices with no operating systems to executables running in large computing clusters.

We choose *QEMU* as our base emulator to generate dynamic FCGs. The reason is that there are several powerful binary analysis platforms that were
developed specifically for malware analysis such as TEMU and DECAF [90, 41]. The benefit of these systems is that they employ several transparent techniques that try to alleviate anti-virtualization techniques which has been used by program authors. Therefore, our system can extract complete and precise dynamic FCGs which include both user and kernel space behaviours.

2.5 Detection

Graph-based approaches employ different techniques to identify malware families. Some of them [6, 1, 17] convert the extracted graph into signature in order to be suitable for pattern matching techniques, while others [21, 28, 4, 81, 71] employ maximum common sub-graph to measure similarity which is sufficient enough to detect metamorphic malware. The primary limitation of these two groups is that their detection method is only applicable to small graph sizes.

Machine learning algorithms have been widely used to cluster or classify malware graphs. A group of works [92, 24, 25] extract features from the graphs to feed them into machine learning classification algorithms such as Random Forest, Naive Bayes, and Decision Tree, while other group [52, 49] employ k-medoids and DBSCAN clustering methods to identify groups of malware with strong structural similarities. Due to the nature of these machine learning algorithms, they are unable to cope with the stream of the graphs, and as a result they are only applicable for off-line detection.
Classy [52] is the only system that devises a new online clustering algorithm to cluster streams of FCGs. However, they had an assumption in the clustering algorithm which makes it ineffective. At very first step, to determine the candidate clusters for the incoming sample, they only consider those graphs that have the same number of node as the incoming sample has. Therefore, the algorithm ignores a lot of possible samples that have a different number of nodes but can have similar behaviour.

2.6 Concluding Remarks

In Table 2.1, we summarize the recent graph-based malware detection approaches and compare them in terms of using unpacker and graph matching techniques. The primary shortcoming of the approaches is the use of static analysis to generate graphs. Static analysis can be ineffective if malware undergoes the obfuscation techniques to hide its malicious behaviour, and as a result the generated graph does not reflect the real functionality of the malware.

Moreover, even though some of the techniques try to circumvent this issue by employing unpacker tools, the experiments show that most of the static and dynamic unpackers are ineffective when faced with complex malware families. The static unpackers are either unable to handle novel samples, or vulnerable to various evasion techniques. On the other hand, dynamic unpackers are susceptible to a variety of anti-monitoring defenses, as well as time bombs or
logic bombs, and can be slow and tedious to identify and disable.

For the purpose of identifying, quantifying, and expressing similarities between malware samples, most of the works used MCS or graph isomorphism that are proven to be an NP-Hard problem. Therefore, applying these methods to large number of graphs, which have huge number of nodes, are ineffective. Moreover, the time complexity of the system is an important parameter that also needs to be considered. SMIT and Classy are the only works that consider the effectiveness of their system in the large-scale scenarios.

To address the mentioned shortcomings, dynamic analysis has been used to generate graphs. Moreover, graph edit distance, which is known to be the most suitable method to compare graphs, has been improved extensively.

Table 2.1: Overview of graph-based malware classification methods

<table>
<thead>
<tr>
<th>Analysis Type</th>
<th>Graph Representation</th>
<th>Unpacker Type</th>
<th>Graph comparison Method</th>
<th>ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>FCG</td>
<td>in-house</td>
<td>Approximate graph matching</td>
<td>[9]</td>
</tr>
<tr>
<td>Static</td>
<td>FCG</td>
<td>-</td>
<td>Approximate graph matching</td>
<td>[15]</td>
</tr>
<tr>
<td>Static</td>
<td>FCG</td>
<td>-</td>
<td>Partial graph isomorphism</td>
<td>[21]</td>
</tr>
<tr>
<td>Static</td>
<td>FCG</td>
<td>-</td>
<td>Graph isomorphism</td>
<td>[28]</td>
</tr>
<tr>
<td>Static</td>
<td>FCG</td>
<td>Sympack</td>
<td>GED- bipartite matching</td>
<td>[43]</td>
</tr>
<tr>
<td>Static</td>
<td>FCG</td>
<td>in-house</td>
<td>GED- adapted version of simulated annealing</td>
<td>[49]</td>
</tr>
<tr>
<td>Static</td>
<td>FCG</td>
<td>in-house</td>
<td>GED- adapted version of simulated annealing</td>
<td>[52]</td>
</tr>
<tr>
<td>Static</td>
<td>FCG</td>
<td>in-house</td>
<td>MCS</td>
<td>[56]</td>
</tr>
<tr>
<td>Static</td>
<td>FCG</td>
<td>-</td>
<td>MCS</td>
<td>[71]</td>
</tr>
<tr>
<td>Static</td>
<td>FCG</td>
<td>RL!depacker and UPX</td>
<td>graph maximum common vertexes or edges</td>
<td>[88]</td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>---------------------</td>
<td>----------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Static</td>
<td>CFG</td>
<td>-</td>
<td>MCS</td>
<td>[1]</td>
</tr>
<tr>
<td>Static</td>
<td>CFG</td>
<td>-</td>
<td>Graph isomorphism</td>
<td>[4]</td>
</tr>
<tr>
<td>Static</td>
<td>CFG</td>
<td>-</td>
<td>Graph isomorphism</td>
<td>[6]</td>
</tr>
<tr>
<td>Static</td>
<td>CFG</td>
<td>-</td>
<td>Graph isomorphism</td>
<td>[11]</td>
</tr>
<tr>
<td>Static</td>
<td>CFG</td>
<td>-</td>
<td>MCS</td>
<td>[81]</td>
</tr>
<tr>
<td>Static and Dynamic (Used Dynamic unpacker)</td>
<td>CFG</td>
<td>in-house</td>
<td>Approximate graph matching</td>
<td>[17]</td>
</tr>
<tr>
<td>Static</td>
<td>Hybrid</td>
<td>-</td>
<td>modified version of GED</td>
<td>[2]</td>
</tr>
</tbody>
</table>
Chapter 3

Proposed System

In this chapter a new scalable dynamic graph-based malware classifier is proposed to address the limitation of static graph-based methods. The proposed system can be used as an automated malware classification service to classify an incoming sample. It also facilitates the whole stack of the Security Response Unit of an AV company; from the analysts dissecting malware and writing detection to the data scientists that try to comprehend every aspect of malware to create new prototype solutions.

For example, providing real time clustering information assists the analysts writing pattern-based signatures since they become aware of as many relevant samples as possible. Therefore, their end result generates better and more accurate detection coverage, and the fact that information is provided in real time, allows to save critical time for protecting the end-users. Alternatively, by examining a group of related malware samples, the analyst might decide
that a different approach must be taken to better detect them in the future. Finally, the additional knowledge gained from the clustering results may also be used for prioritizing samples in the queues of other automation systems or manual inspections.

3.1 Overview

The general framework of the proposed system is outlined in Figure 3.1. As it can be seen each incoming sample undergoes three phases:

- Graph Generation: Suspicious binaries are sent to dynamic analyzer, preprocessor, and graph generator. The outputs of this phase are the corresponding dynamic function call graphs and control flow graphs.

- Graph Matching: A number of graph edit distance techniques are applied to graphs to compute their similarity.

- Classification: The distances calculated in the previous phase are used to classify the incoming samples using machine learning algorithms.

3.2 Graph Generation

In the graph generation phase, an input binary is passed to the following three steps to generate its output graphs: dynamic analyzer, preprocessor and graph generator. Dynamic analyzer records instruction traces of a given
sample. The output traces are transformed into assembly instructions by the preprocessor. Also, the function call names are added to assembly instructions to prepare the assembly file for graph generator. Finally, graph generator generates dynamic graphs from the incoming assembly instruction files.

### 3.2.1 Dynamic Analyzer

We take a trace-based approach to construct graphs via dynamic analysis. The analyzer records every single instruction and its related states during
the execution of an incoming sample. It also records statistics about loaded executables and libraries, tracks the entry of tainted data to the process space, and produces a log of function calls, including arguments and return values that we later use to generate function call graphs.

The output of the dynamic analyzer are the generated trace file in hexadecimal format and the function calls log for each running sample.

### 3.2.2 Preprocessor

Instruction traces generated by dynamic analyzer are in hex format and not yet suitable for graph generator. Therefore, the hex instruction traces need to be converted into assembly instructions. To convert traces, we take sequences of 1-15 bytes along with machine mode information from a trace file and produce an assembly instruction structure describing the op-code, their operands, and the corresponding flags.

The generated assembly files still are not suitable for generating function call graphs. Additional information such as system and library calls need to be added to the assembly instructions to prepare them for graph generation.

The algorithm 1 shows the procedure of the adding function calls to assembly instructions by using the log of function calls.

The process starts looking for every `call` or `ret` instruction to extract the address following these instructions. It then checks the extracted address to see if there is an equivalent for it in the function call log. If there exists a match, the function call name is extracted from the log file and will be added
to the end of the current instruction line. These function calls would be the
external function calls. If there does not exist any match for the requested
address in the log file, it means the function call is internal. Therefore, the
algorithm generates a internal function call name using “sub_” following the
address, (e.g. sub_0x7009453). Then the generated function call will be
added to the end of the instruction line.

Algorithm 1 Add Function Calls To Assembly Instruction

1: function ADD_FUNCTION_CALLS_TO_ASSEMBLY_INSTRUCTION(Assembly Instructions file(AIF), Function Calls Log File(FCLF))
2:     Read the first line of AIF
3:     while not end of assembly instruction file do
4:         Get the instruction and it’s corresponding address
5:         if the instruction is Call or Ret then
6:             if the FCLF contains the address then
7:                 Get the function call name equivalent to the address from the FCLF
8:                 Add the function call name to the end of the current line
9:             else
10:                Generate a internal function call by using the “sub_” + the address
11:                Add the generated internal function call to the end of the current line
12:             end if
13:         end if
14:     Read next line
15:     end while
16: end function

3.2.3 Graph Generator

The graph generator takes the generated assembly file, which includes the
function calls names, and generates the corresponding dynamic function call
graph and dynamic control flow graph.

Dynamic function call graph generator
The static function call graph generator algorithms are not applicable to generate dynamic graphs because they extract assembly instructions of a given sample without running it and as a result, there does not exist any ret instruction for every call instruction. While in dynamic analysis, there exist a ret instruction for each call instruction. Therefore, edge generation methods in static analysis are completely different than those of the dynamic analysis.

Due to the mentioned problem, we propose a new algorithm to generate dynamic function call graphs. The proposed algorithm in pseudo-code is given in Algorithm 2. In the proposed algorithm the function object is created by using the function call name at the end of each instruction line. The algorithm works as follow:

- For the first instruction, create a function object at the address of that instruction.
- For each call statement or push + ret, create a function object and add an edge from the current function to this new function object.
- For each new call statement, create or reuse a function object and add an edge from the current function to the new or already known function object.
- After a ret instruction, change the current function to the previous one.

Figure 3.2 shows dynamic function call graph of Fareit malware, which was generated by our system.
Algorithm 2 Dynamic Call Graph Generation

1: function CallGraphGenerator(Assembly instruction files included function calls(AIF))
2: Read the first line of AIF
3: For the first instruction, create a Function object at its address
4: CurrentFunction ← Created function object
5: Push the address to the stack
6: while it is not the end of file do
7: Read the next line of the AIF
8: Get the address and instruction
9: if instruction = Call then
10: Create a node with the address or function name
11: Create an edge from the currentFunction to this node
12: Push the address or function name to the stack
13: CurrentFunction ← Node address or node function name
14: Read the next line of the AIF
15: end if
16: if Instruction = push then
17: Read the next line of the AIF
18: Get the address and instruction of the next line
19: if Instruction = Ret then
20: Create a node with the address or function name
21: Create an edge from the currentFunction to this node
22: Push the address or function name to the stack
23: Read the next line of the AIF
24: CurrentFunction ← Node address or node function name
25: end if
26: if Instruction = Call then
27: Create a node with the address or function name
28: Create an edge from the currentFunction to this node
29: Push the address or function name to the stack
30: CurrentFunction ← Node address or node function name
31: Read the next line of the AIF
32: end if
33: Push the address or function name to the stack
34: CurrentFunction ← Node address or node function name
35: end if
36: if Instruction = Ret then
37: Pop from stack
38: CurrentFunction ← Top of stack
39: end if
40: end while
41: end function
Figure 3.2: Dynamic Function Call Graph of Fareit Malware
**Dynamic control flow graph generator** To generate dynamic control flow graphs, we employ the method introduced by Kinder et al. [50]. The algorithm implements multiple rounds of assembly instruction analysis interleaved with dataflow analysis. In each round, the assembly instructions are translated to an intermediate representation, from which the platform builds a more accurate control-flow graph.

Figure 3.4 shows the small part of dynamic CFG of Fareit malware in assembly format and figure 3.4 represents its equivalent in intermediate language.

### 3.3 Graph Matching

The graph comparison is the most essential part of our proposed system. Its accuracy and time complexity are directly related to the performance of the system from the perspective of quality and throughput.

#### 3.3.1 GED Calculator

We employ GED to compare graphs. As mentioned before the GED is NP-hard problem and an approximation algorithm is needed to deal with this problem. The *Simulated Annealing* algorithm is used to approximate GED. To introduce the GED approximation algorithm, we provide a short overview of the terminology and notation used in the algorithm, along with the appropriate background.
Figure 3.3: Dynamic CFG of Fareit Malware in Assembly Instructions
Figure 3.4: Dynamic CFG of Fareit Malware in Intermediate Representation
3.3.1.1 Background

A graph \( G = (V,E) \) is composed of a set of vertices \( V \) and a set of edges \( E \subseteq V \times V \). The order of a graph \( G \) is the number of vertices \( |V(G)| \) in \( G \). In this thesis, we only deal with directed graphs; an edge (or arc) is denoted by its endpoints as an ordered pair of vertices. Vertex \( v \) is said to be a direct predecessor of \( u \) if \( (v, u) \in E \). In this case, \( u \) is called a direct successor of \( v \). The out-degree \( d^+(v) \) of vertex \( v \) is the number of direct successors of \( v \). Similarly, the in-degree \( d^-(v) \) is the number of direct predecessors of \( v \). Finally, the degree \( d(v) \) of vertex \( v \) is \( d^+(v)+d^-(v) \). The out-neighbourhood (direct successor set) \( N^+(v) \) of vertex \( v \) consists of the vertices \( \{w \mid (v,w) \in E\} \), and the in-neighbourhood (direct predecessor set) \( N^-(v) \) is the set \( \{w \mid (w,v) \in E\} \).

Definition (Graph matching): For two graphs, \( G \) and \( H \), of equal order, the graph matching problem consists of finding a bijective mapping \( \phi : V(G) \rightarrow V(H) \) of optimal value with respect to a cost function.

Definition (Graph Edit Distance): The GED of two graphs \( G,H \) is the minimum cost induced by elementary edit operations required to transform a graph \( G \) into graph \( H \). A cost is defined for each elementary edit operation. In our case, the elementary edit operations considered are: vertex insertion/deletion, edge insertion/deletion and vertex relabelling. We assign unit cost to all operations.

To find a bijection that maps the vertex set \( V(G) \) to \( V(H) \) the graphs \( G \) and \( H \) have to be of the same order. However, the latter is rarely the case.
when comparing call graphs. To circumvent this problem, the smaller of the vertex sets $V(G)$ and $V(H)$ can be supplemented with disconnected (dummy) vertices $\epsilon$ such that the resulting sets $V'(G)$ and $V'(H)$ are of equal size. A mapping of a vertex $v$ in graph $G$ to a dummy vertex $\epsilon$ is then interpreted as deleting vertex $v$ from graph $G$, whereas the opposite mapping implies a vertex insertion into graph $H$.

Now, for a given graph matching, we can define three cost functions: VertexCost, EdgeCost and RelabelCost.

- **VertexCost** The number of deleted/inserted vertices:
  \[
  |\{v : v \in [V'(G) \cup V'(H)] \land [\phi(v) = \epsilon \lor \phi(\epsilon) = v]\}|
  \]

- **EdgeCost** The number of unpreserved edges:
  \[
  |E(G)| + |E(H)| - 2 \times |\{(i,j) : (i,j) \in E(G) \land (\phi(i), \phi(j)) \in E(H)\}|
  \]

- **RelabelCost** The number of mismatched functions. A function is mismatched if it is either a local function and is matched to an external function, or if it is an external function matched to a local function or an external function with a different name.

The sum of these cost functions results in the graph edit distance $\lambda_\phi(G,H)$:

\[
\lambda_\phi(G,H) = \text{VertexCost} + \text{EdgeCost} + \text{RelabelCost}
\]

51
Definition (Graph dissimilarity): The dissimilarity $\delta(G,H)$ between two graphs $G$ and $H$ is a real value on the interval $[0,1]$, where 0 indicates that graphs $G$ and $H$ are identical whereas a value near 1 implies that the pair is highly dissimilar. In addition, the following constraints hold: $\delta(G,H) = \delta(H,G)$ (symmetry), $\delta(G,G) = 0$, and $\delta(G,K_0) = 1$ where $K_0$ is the null graph, $G \neq K_0$. Finally, the dissimilarity $\delta(G,H)$ of two graphs is obtained from the graph edit distance $\lambda_\phi(G,H)$:

$$\delta(G,H) = \frac{\lambda_\phi(G,H)}{|V(G)| + |V(H)| + |E(G)| + |E(H)|}$$

As mentioned before, finding the minimum GED, i.e. $\min_\phi(\lambda_\phi(G,H))$, is a NP-hard problem but can be approximated.

### 3.3.1.2 Simulated Annealing

Simulated annealing (SA) is a generic probabilistic method that was first proposed in 1983 by Kirkpatrick et al. [51] to solve hard combinatorial optimization problems. No specific knowledge about the way to approach the problem is required for implementing SA. This allows the use of SA in a variety of problems without changing the basic structure of the algorithm. SA aims at finding the global optimum of a cost function over a set of feasible solutions.

In the call graph matching problem the search space is defined over all the
possible bijective mappings \( \phi \) between two graphs. The SA process starts from an arbitrary bijective mapping as an initial state. Then a neighbouring state in the search space is selected randomly. Neighbouring states are created by choosing a pair of vertices in one of the graphs and swapping their matching vertices. The difference in the cost function for the two states determines whether the current state must be replaced by the new state or not. We denote the difference in the cost function evaluated for two states by \( \Delta(\lambda_{\phi_i}, \lambda_{\phi_{i+1}}) \). If the new state (bijective mapping) gives a lower value for the cost function, it replaces the current state. Otherwise, the move is accepted with probability \( e^{-\beta \Delta(\lambda_{\phi_i}, \lambda_{\phi_{i+1}})} \). SA is allowed to run for a predefined number of steps before the value of \( \beta \) is increased.

The annealed parameter \( \beta \) is the inverse temperature used in statistical physics. For small values of \( \beta \) almost any move is accepted in the process. For \( \beta \to \infty \) the process is essentially a downhill move in which the SA state will be replaced by the new bijective mapping only if the new state gives a lower cost. The reason to introduce the annealed parameter is to overcome the problem of getting stuck in local minima by allowing non preferential moves.

The sequence of \( \beta \) can be considered an annealing schedule. The annealing schedule contains the initial and final values of the annealed parameter, denoted by \( \beta_0 \) and \( \beta_{\text{final}} \), together with the cooling rate, \( \epsilon \), which determines the changes in \( \beta \). In our implementation we chose the cooling rate to be a multiplier factor in \( \beta \) which takes values on the interval \([0, 1]\). Then the
sequence of the values of $\beta$ is determined by $\beta_{t+1} = \beta_t / \epsilon$. We will refer to the number of times that $\beta$ changes with the term relaxation iterations.

There are two terminating conditions for SA. The first is achieving the minimum graph edit distance. But since this is the problem SA is called to solve, a lower bound is computed and used instead [52]. The second terminating condition comprises of terminating the SA process when no better solution has been identified within a certain number of the most recent neighboring solutions; this is the no_progress() function in Algorithm 3.

**Algorithm 3** Simulated Annealing for computing GED

```
1: function SIMULATED_ANNEALING(Graph: G, H; Annealed parameter values: $\beta$, $\beta_{final}$; Cooling ratio: $\epsilon$; Iterations per relaxation: $m$)
2:     $\phi_i \leftarrow$ random_phi()
3:     $\beta \leftarrow \beta_0$
4:     while $\beta < \beta_{final}$ do
5:         for $m$ iterations do
6:             $\phi_{i+1} \leftarrow$ neighbor_solution($\phi_i$)
7:             $\Delta(\lambda_{\phi_i}, \lambda_{\phi_{i+1}}) \leftarrow \lambda_{\phi_{i+1}} - \lambda_{\phi_i}$
8:             if $\Delta(\lambda_{\phi_i}, \lambda_{\phi_{i+1}}) < 0$ then
9:                 $\phi_{i+1} \leftarrow \phi_i$
10:            else with probability $e^{-\beta \Delta(\lambda_{\phi_i}, \lambda_{\phi_{i+1}})}$
11:                $\phi_{i+1} \leftarrow \phi_i$
12:            end if
13:        if $\min_{\phi} \lambda_{\phi} = \lambda_{\phi_i}$ or no_progress() then
14:            return $\phi_i$
15:        end if
16:     end for
17:     $\beta \leftarrow \beta / \epsilon$
18: end while
19: end function
```
3.3.1.3 Improved Simulated Annealing

The computational cost of the simulated annealing algorithm is relatively high, and therefore it is not suitable to compare large dynamic graphs. We employ the stochastic beam search \[64\] to reduce the high computational complexity of this algorithm as well as providing more accurate approximation for computing graph edit distance.

Algorithm 4 Improved Simulated Annealing for computing GED

1: function IMPROVED_SIMULATED_ANNEALING(Graph: G, H; Annealed parameter values: $\beta$, $\beta_{final}$; Cooling ratio: $\epsilon$; Iterations per relaxation: $m$; parameter for choosing number of random solutions: $k$)
2: Generate $k$ random solution($random_\phi()$)
3: $\beta \leftarrow \beta_0$
4: while $\beta < \beta_{final}$ do
5:     for $m$ iterations do
6:         For each random solution generate $k$ neighbor solution (neighbor_solution($\phi_i$))
7:         Calculate cost function ($\lambda$) for all solutions
8:         Sort all cost functions ($\lambda$)
9:         Select $k$ best solutions
10:        for all remaining solutions do
11:           with probability $e^{-\beta \Delta(\lambda_{\phi_i}, \lambda_{\phi_{i+1}})}$
12:              Select the solution
13:        end for
14:        for all selected solutions do
15:           if $min_{\phi} (\lambda_{\phi}) == \lambda_{\phi_i}$ or no_progress() then
16:              return best $\phi$
17:           end if
18:        end for
19:     end for
20:     $\beta \leftarrow \beta / \epsilon$
21: end while
22: end function

In the proposed algorithm, instead of choosing one random solution and calculating only one neighbour solution, $k$ random solutions are generated...
and for each random solution, $k$ neighbour solutions are generated. Then the cost function of all solutions are calculated. The algorithm choose $k$ best solutions based on the ordered cost functions and $k$ solution will be selected with probability $e^{-\beta \Delta(\lambda_{\phi_i}, \lambda_{\phi_i+1})}$. The rest of the algorithm is similar to the original SA Algorithm. modified SA algorithm procedure is given in Algorithm 4.

### 3.4 Classification

Classification module is used to determine whether a newly acquired binary sample is a representative of a known family of malware or it represents a new malware. To keep abreast of the increasing amount of malware in the wild, clustering and classification methods are required to process thousands of samples on a daily basis. Unfortunately, most machine learning methods scale super-linearly with the number of input data and thus are not directly applicable for malware analysis. To address this problem, we follow the approximate classification algorithm proposed by Rieck et al. [67], referred to as nearest prototype classification, which resembles the costly K-Nearest Neighbor algorithm. In this algorithm, instead of considering all instances, representative instances, called prototypes, are extracted and used in the clustering and classification processes. Remaining instances are then labeled as their closest prototype. The term distance used in the following subsections refers to graph edit distance calculated by simulated annealing.
3.4.1 Prototype Extraction

A prototype in our system is a function call graph that can represent its surrounding function call graphs. Extracting an optimal set of prototypes from data set is NP-hard that can be performed by employing either clustering algorithms or super-linear computations. However, they are not suitable for efficient approximation. We use the linear-time prototype extraction algorithm suggested by Gonzalez [36] (Algorithm 5), where distance\(x\) determines the distance between graph \(x\) and its nearest prototype. The algorithm starts by adding first graph in the training set into the list of prototypes. Subsequently, farthest graphs are selected as prototypes one at a time, and distance\(x\) is recalculated for each graph \(x\). This process continues until the distance of all graphs from their closest prototype is less than a specified threshold, \(d_p\). The algorithms run-time linearly increases by the number of graphs and prototypes.

3.4.2 Clustering of Prototypes

The clustering phase, only cluster extracted prototypes into groups of similar malware families and identify the unknown samples. Algorithm 6 describes the employed hierarchical clustering algorithm. The algorithm starts with each prototype being an individual cluster, and then iteratively determines
Algorithm 5 Prototype extraction

1: function Prototype_Extraction(Graphs)
2: prototypes ← ∅
3: distance[x] ← ∞ for all x ∈ graphs
4: while max(distance) > max_dist_prototype(dp) do
5: choose z such that distance[z] = max(distance)
6: for x ∈ graphs and x ≠ z do
7: if distance(x) > Simulated_Annealing(x, z, β, β_final, ϵ, m) then
8: distance(x) ← Simulated_Annealing(x, z, β, β_final, ϵ, m)
9: end if
10: end for
11: add z to prototypes
12: end while
13: end function

and merges the nearest pair of clusters. This procedure is terminated if the distance between the closest clusters is larger than the threshold $d_c$.

Finally, the clusters with fewer than $m$ graphs are rejected and kept for further analysis.

Algorithm 6 Clustering Using Prototypes

1: function Prototypes_Clustering(Prototypes)
2: for x,y ∈ prototypes do
3: distance(x,y) ← Simulated_Annealing(x,y, β, β_final, ϵ, m)
4: end for
5: while min(distance) ≤ min_dist_cluster(dp) do
6: merge clusters x and y with minimum(distance[x, y])
7: update distance using complete linkage
8: end while
9: for z ∈ graphs do
10: x ← nearest prototypes to z
11: assign z to cluster containing x
12: end for
13: reject clusters with less than $m$ members
14: end function
3.4.3 Classification

To identify unknown samples, similar malware families are grouped in the clustering phase but to assign a label to each sample we need a classification approach. We provide an approximation classification approach depicted in Algorithm 7, where classification is simply done by propagating each prototype's label to its corresponding members. Members, whose distance from the closest prototype is more than a predefined threshold $d_r$ are rejected and kept for later incremental analysis. Similar to the prototype extraction, the algorithms run-time linearly increases by the number of graphs and prototypes.

Algorithm 7 Prototype classification

1: function Prototypes_classification(Prototypes)
2:     for $x \in$ prototypes do
3:         $z \leftarrow$ nearest prototype to $x$
4:         if Simulated_Annealing($z, x, \beta, \beta_{final}, \epsilon, m) \geq \text{max}\_\text{dist}\_\text{classify}$ then
5:             reject $x$ as unknown class
6:         else
7:             assign $x$ to cluster containing $z$
8:         end if
9:     end for
10: end function

3.4.4 Incremental classification

While most of the existing approaches have been restricted to batch analysis, but in real environment we face with stream of malware every day. To handle stream of malware, we process the incoming samples in small chunks, for
example on a daily basis. To realize an incremental analysis, we need to keep track of intermediate results, such as clusters determined during previous runs of the algorithm. Fortunately, the concept of prototypes enables us to store discovered clusters in a concise representation and, moreover, provides a significant speed-up if used for classification. Algorithm 8 sketches the incremental classification procedure which starts by checking input graphs against previous prototypes, then re-clustering the remaining graphs.

Algorithm 8 Incremental classification

1: function INCREDMENTAL_CLASSIFICATION(Prototypes)
2: prototypes ← ∅
3: rejected ← ∅
4: for graphs ← new graphs ∪ rejected graphs do
5: classify graphs to known clusters using prototypes
6: extract prototypes from remaining graphs
7: cluster remaining graphs using prototypes
8: prototypes ← prototypes ∪ prototypes of new clusters
9: rejected ← rejected graphs from clustering
10: end for
11: end function

3.5 Concluding Remarks

In this chapter a dynamic malware classifier was proposed to address the limitation of static-based graph generation methods. The main objective of the proposed framework is to classify and cluster streams of large function call graphs. Each incoming sample is executed by our dynamic anaylzer to record its instruction traces. The captured traces are preprocessed to be suit-
able for graph extraction. In the preprocessing phase, traces are converted
to assembly instructions and function call names are added to assembly in-
struction files. A new algorithm is devised to generate dynamic graph from
assembly instructions. The graph comparison measure is the GED and it is
approximated using a revised version of Simulated Annealing algorithm. Fi-
nally, we adopt an stream clustering algorithm to cluster and classify stream
of call graphs.
Chapter 4

Implementation

This chapter provides an architectural overview of the proposed system and also presents a number of different architectural views to depict different aspect of the system. To provide the overall understanding of the system we document two different views of the system: module view and behavioural view. Module view shows how the system is structured as a set of implementation units, and behavioural view shows how the modules interact together. We employ UML diagrams to illustrate different view of the system. The class diagram is used to depict the system module view and the sequence diagram is employed to show the behavioural view of the system.

4.0.1 System Overview

The system has been built upon TEMU (version 1.0) [90], open-source whole-system out-of-the-box fine-grained dynamic binary analysis that provides
whole-system view to facilitate fine-grained instrumentation, and also pro-
vides sufficient efficiency. Our modular system interfaces with Intel's XED2 [44]
library for instruction decoding and Tracecap [90] for reading and writing in-
struction traces.

The proposed classifier consists of five main components as depicted in com-
ponent diagram (Figure 4.1): instruction tracer, instruction decoder, prepro-
cessor, graph generator, graph comparator and classifier. Instruction tracer
captured the execution instruction of incoming binary. Instruction decoder
continues the trace file to assembly file. The generated assembly file will be
preprocessed by preprocessor to add external function call to it. Graph gen-
erator is responsible to generate dynamic control flow and dynamic function
call graphs. The distance between graphs is calculated by graph comparator.
Finally, the classifier classifies graphs based on the calculated distances.

4.0.2 Module View

The module view shows how the proposed system is decomposed into man-
ageable software units. The elements of the module view type are modules.
A module is an implementation unit of the system that provides a coherent
unit of functionality.

The module view of the proposed system in the UML class diagram is illus-
trated in Figure 4.2, followed by the description of each class.
4.0.2.1 Instruction Tracer

Instruction tracer uses Python subprocess to automatically run the TEMU dynamic analyzer to capture the instruction traces and generate the log of function calls. This class includes two main functions:

**temuAutomation**: Use Python subprocess to automate TEMU.

**InputCommand**: Execute TEMU commands using python subprocess to capture instructions.

4.0.2.2 Instruction Decoder

This class employs lib XED2 to decode the generated trace files into assembly instructions files. It includes the following functions:
readTraceFiles: reads the generated trace files.

convertTraceToInstruction: converts hex trace file to assembly instructions file.

4.0.2.3 Preprocessor

This class prepares the generated assembly files for graph generation. Following are the descriptions of the functions:

TraceToAssembly: uses an instance of Instruction Decoder class to decode the hexadecimal trace into assembly instructions.

addExternalFunctionCalls: add function calls generated by log of function calls to assembly file.

4.0.2.4 Graph Generator

Graph Generator is responsible for generating dynamic graphs. Functions of this class are:

GraphGenerator: generated dynamic function calls and control flow graphs.

dotToGdl: since the system only works for graph description language (gdl) graphs the generated plain text graph description language (dot) graphs will convert to gdl format by this function.

4.0.2.5 BitMatrix

BitMatrix class is a two dimensional indexed collection of bit values which has been used to store adjacency matrix of a generated graph in an efficient
way.

4.0.2.6 Graph

The graph class is used to store each generated graph for graph comparison.

4.0.2.7 Handler

The entire process of the proposed system is controlled by a handler starting from the instruction tracer to the classifying the incoming samples.

4.0.2.8 Classifier

This class implements prototype extraction, clustering, and classification algorithms of the classifier module. Main functions include:

**readGraphs**: reads generated graphs for each incoming sample.

**GEDCalculator**: creates the distance matrix of the graphs.

**updateDistance**: updates the distance between two input graphs.

**assignClustertoGraphs**: assigns the graph to its closest cluster.

**Cluster**: clusters the prototypes.

**ExtractPrototype**: extracts a set of prototypes from the graphs.

**minDistance**: returns two graphs with the minimum distance.

**maxDistance**: returns two graphs with the maximum distance.
4.0.2.9 GEDCalculator:

This class compares the incoming graphs. The main functions of this class are:

- **calculateUpdatedCost**: updates cost function after swapping two nodes.
- **relabelingCostForNode**: calculates the cost of relabelling of each node.
- **edgeCostForNode**: calculates the edge cost for each node.
- **madeSameSizeGraph**: compares two graphs and adds the require number of dummy nodes to the smaller graph to make them same size.
- **generateRandomBijectiveFunction**: generates bijective function for two incoming graphs.
- **findNeighborSolution**: finds neighbor solution for the bijective function by swapping two random nodes.
- **costFunction**: computes the overall cost which includes relabelling cost, edge cost and vertex cost.
- **relabelingCost**: computes relabelling cost for the graph.
- **edgeCost**: calculates edge cost for the graph.
- **vertexCost**: compute vertex cost for the graph.
- **lowerBound**: calculate the Lower Bound algorithm for GED.
- **noProgress**: terminates the process.

4.0.2.10 Simulated Annealing

This class implements the Simulated Annealing algorithm to compute graph edit distance.
4.0.2.11  Adapted Simulated Annealing

This class implements the proposed Adapted Simulated Annealing algorithm to compute graph edit distance.

4.0.3  System Behaviour

Sequence diagram has been employed to depict the behaviour of the proposed system and show how modules interact together. As illustrated in Figure 4.3, the sequence diagram shows how the incoming sample goes through different objects of the system to get its final label.

4.0.4  Conclusion

This chapter presents the design and implementation of the proposed system. Two different architectural views are used to show different aspects of the system. The module view, which employs UML class diagram, shows the implementation modules of the system and behavioural view (sequence diagram) depicts the interactions between objects in the sequential order that those interactions occur.
Figure 4.2: Proposed System Class Diagram
Figure 4.3: Proposed System Sequence Diagram
Chapter 5

Experiments and Results

To demonstrate the capability of the proposed system in accurately classifying samples, we compare our system with most recent online static graph based malware classification approach, Classy. In addition to comparative evaluation, static graphs have been applied to the proposed system to present the power of dynamic graphs. This chapter starts by describing the reference dataset used for the experiments, and parameters calibrated for the maximum performance of the system, followed by system evaluation explained in Section 5.3.

5.1 Datasets and Labelling

Assessing performance of any detection/classification approach requires test and evaluation with a dataset that is heterogeneous enough to simulate real
samples to an acceptable level. We built such a dataset by selecting a reasonable mix of different benign and malicious code variants currently popular on the Internet from the following well-known sources: Ether dataset [20], Malicia dataset [65], VirusTotal repository [80], VirusShare [72] and Virus-Sign [73].

Table 5.1: Distribution of Malware Types in the Dataset

<table>
<thead>
<tr>
<th>Malware Type</th>
<th>Percentage</th>
<th>Malware Type</th>
<th>Percentage</th>
<th>Malware Type</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spammer</td>
<td>2.31</td>
<td>Tool</td>
<td>2.51</td>
<td>Dialer</td>
<td>1.08</td>
</tr>
<tr>
<td>Virus</td>
<td>1.26</td>
<td>Worm</td>
<td>2.88</td>
<td>HackTool</td>
<td>2.77</td>
</tr>
<tr>
<td>Rogue</td>
<td>7.91</td>
<td>Adware</td>
<td>1.92</td>
<td>Ransom</td>
<td>2.97</td>
</tr>
<tr>
<td>RemoteAccess</td>
<td>1.8</td>
<td>BrowserModifier</td>
<td>1.97</td>
<td>Trojan</td>
<td>13.88</td>
</tr>
<tr>
<td>VirTool</td>
<td>13.08</td>
<td>Backdoor</td>
<td>6.67</td>
<td>Program</td>
<td>1.66</td>
</tr>
<tr>
<td>PWS</td>
<td>10.78</td>
<td>Trojan (Clicker, Downloader, Dropper, ...)</td>
<td>35.53</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Our dataset contains 9850 benign executables and 40,000 malware from 346 different families. We added diversity by selecting malware from different categories (viruses, rootkits, etc.), and with different packer. Table 5.1 shows the different malware types distribution and the distribution of the 18 most popular packers observed in our dataset is reported Table 5.2. Packers with a frequency of less than 20 samples have been grouped under “Other” category. We scanned all samples by 57 online AVs and selected Microsoft AV for labeling the dataset as it was able to successfully label the maximum number of samples.
Table 5.2: Distribution of Packers Types in the Dataset

<table>
<thead>
<tr>
<th>Packer Type</th>
<th>Percentage</th>
<th>Packer Type</th>
<th>Percentage</th>
<th>Packer Type</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPX</td>
<td>16</td>
<td>WinUnpack</td>
<td>2</td>
<td>Xtreme Protector</td>
<td>2</td>
</tr>
<tr>
<td>Armadillo</td>
<td>18</td>
<td>AsPack</td>
<td>3</td>
<td>AsProtect</td>
<td>3</td>
</tr>
<tr>
<td>InstallShield</td>
<td>5</td>
<td>nPack</td>
<td>1</td>
<td>NsPack</td>
<td>2</td>
</tr>
<tr>
<td>PECompact</td>
<td>5</td>
<td>PESpin</td>
<td>3</td>
<td>PKLITE 32</td>
<td>2</td>
</tr>
<tr>
<td>SVKProtector</td>
<td>1</td>
<td>tElock</td>
<td>3</td>
<td>Themida/WinLicence</td>
<td>7</td>
</tr>
<tr>
<td>Upack</td>
<td>2</td>
<td>Other</td>
<td>5</td>
<td>Unpacked</td>
<td>20</td>
</tr>
</tbody>
</table>

5.2 System Calibration

The use of machine learning techniques frequently involves careful tuning of the learning parameters. Similarly, the discriminating power of the classifier module is significantly influenced by the prototype extraction, clustering and classification algorithms explained in section 3. These algorithms should be provided with parameters that ensure maximum performance, which in our case means the maximum classification accuracy and the minimum number of rejected or unlabeled samples. To avoid over-tuning, i.e. using all samples to calibrate the classifier model that fits the training data “too well”, we only use 20% of the dataset for parameter setting and the remainder of the dataset for the final system testing. Following a brute force search, we perform different rounds of experiments to decide on the optimal values for each parameter. Figure 5.1 shows the relationship between each machine learning parameter and the classification accuracy and Figure 5.2 shows their relationship in terms of rejected samples percentage. As the figures show, to
get the highest accuracy and minimum number of the rejected samples the value of $\text{Max\_dist\_prototype}$ should be between 0.25-0.4, $\text{Min\_dist}$ need to be from 0.85 to 0.9 and $\text{min\_dist\_cluster}$ have to be tuned from 0.7 to 0.9.

5.3 System Evaluation

To evaluate the effectiveness of the proposed system, we have performed several experiments in terms of accuracy and time complexity. We use both static and dynamic graphs as input to evaluate the effectiveness of dynamic FCGs in comparison to static FCGs. Moreover, we have performed comparison experiments with most recent work in online FCG clustering, Classy [52]. The purpose of this experiment is to compare the discrimination power and performance of our dynamic online classification approach in comparison to
Figure 5.2: Rejected samples considering different parameters value

their static online clustering.

To simulate real-world stream of samples, we split the whole dataset into 5 parts, resembling batches of samples received in 5 consecutive days. At the end of each day classification accuracy as the number of samples correctly labeled is calculated and rejected samples are carried forward to the next day's input samples.

5.3.1 System performance and statistics

In this part we consider run-time performance and effectiveness of each module of the proposed system. To do that, we employ static graphs in two states (using unpacker tool before graph extraction and directly extracting graphs without using any unpacker tool) as input to the system in addition to our
dynamic graphs. All experiments are conducted on an Intel Core i7-3770 Quad-Core Processor with 3.4 GHz and 16GB memory.

5.3.1.1 Graph Generation

Since the proposed system uses dynamic analysis to generate FCGs, it requires time to run each binary in a protected environment. Each sample is run for a fix amount of time (3 minutes) to capture its traces. We exclude dynamic analyzer time requirement from evaluating time complexity of graph generation phase. We consider time requirements for preprocessing and extraction steps separately to compare static and dynamic graph extraction. In static graph extraction, preprocessing is referred to unpacking phase of a binary. Figures 5.3 and 5.4 depict the run-time performance of preprocessing and graph extraction phases, respectively.

In static graph extraction preprocessing time depends on the packing techniques and the size of the binary while in dynamic graph extraction it depends on the size of the Trace file. Depending on the packer technique, 55% of samples can be preprocessed in less than 100 seconds while less than 10 percentages of samples require more than 250 seconds for unpacking. The preprocessing time for dynamic graphs is higher than the static one where only 22% of trace files can be preprocessed in less than 100 seconds and around 26% of samples need a preprocessing time more than 250 seconds. It is expected to have high preprocessing time for dynamic graphs since the average size of the trace files is more than 3GB, which requires the significant
amount of time to decode them into assembly file.

In contrast to preprocessing time for static approach in which the preprocessing time is much lower than dynamic preprocessing, the time complexity of dynamic graph extraction is lower than static graph extraction. As Figure 5.3 shows, nearly 90% of dynamic graphs are generated in less than 100 seconds while percentage of extracted static graphs is around 55. This is because the static graph extraction is done by IDA Pro which first disassembles the binary and then generates the corresponding graph in which the disassembling of a binary is a time consuming process.

We also provide statistics regarding the size of the different files generated in each step of the preprocessing. Table 5.3 gives a summary of the output size
at each step of graph generation phase and also size of incoming binaries. As can be seen, the size of trace file and assembly file generated in the dynamic analyser and preprocessor steps are extremely large and can even exceed 10 GB. However, dynamic FCGs generated in the graph generator step reduce the trace file and assembly file 48 and 15 times, respectively. Also, it shows that the size of generated dynamic FCGs are comparable to Static FCGs and in most of the cases have the similar size as static graphs have.

Table 5.3: Size of the output of each phase comparison

<table>
<thead>
<tr>
<th></th>
<th>TraceFile</th>
<th>AssemblyFile</th>
<th>Dynamic FCG</th>
<th>Static FCG</th>
<th>Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>3 GB</td>
<td>3.450 GB</td>
<td>0.4835 MB</td>
<td>0.0305 MB</td>
<td>0.4425 MB</td>
</tr>
<tr>
<td>Minimum</td>
<td>150 MB</td>
<td>200 MB</td>
<td>0.0056 MB</td>
<td>0.0001 MB</td>
<td>0.0014 MB</td>
</tr>
<tr>
<td>Maximum</td>
<td>10 GB</td>
<td>13 GB</td>
<td>3.5604 MB</td>
<td>2.6586 MB</td>
<td>7.5047 MB</td>
</tr>
</tbody>
</table>
The output of preprocessing and graphs extraction steps are the generated function call graphs. The percentage of the extracted graphs are reported in Table 5.4. The proposed dynamic method generates corresponding FCG for each incoming sample as reported in Table 5.4. The percentage of the extracted static graphs depends on the preprocessing step. If the graphs directly extracted from IDA Pro without using any unpacker tools, the percentage of generated graphs is 76.29. IDA Pro cannot disassemble samples which have employed packer techniques and as a result cannot generate the desired FCG. By employing unpacker the percentage of the generated CFGs is increased by around 14%. Even though it improves the percentage of generated graphs but it still cannot unpack most of the samples and therefore, the IDA Pro cannot generate the corresponding FCGs.

<table>
<thead>
<tr>
<th>Method</th>
<th>Extracted Graphs Numbers</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>50000</td>
<td>100</td>
</tr>
<tr>
<td>Static without packer tool</td>
<td>38149</td>
<td>76.29</td>
</tr>
<tr>
<td>Static with packer tool</td>
<td>44705</td>
<td>89.41</td>
</tr>
</tbody>
</table>

### 5.3.1.2 Graph Matching

The most important part of the system is the Graph matching phase which directly affects the performance and effectiveness of the proposed system. Similar to the Classy, the original SA and Improved SA is used with parameter values $\beta_0 = 4.0$ and $\epsilon = 0.9$. Figures 5.5 and 5.6 show the time required in the comparison phase for both static and dynamic graphs respectively.
Figure 5.5: Time requirement comparison original SA

Figure 5.6: Time requirement comparison modified SA
A small increase of computational complexity can be observed when using proposed improved approximation algorithm, where the time requirement for 80% of static graphs is between 1-20 centi-second in proposed algorithm, whereas the original SA calculate GED for around 75% of graphs at the same time. The case is similar for dynamic graphs in which 78% of graphs are compared together in between 100 to 300 centi-second by employing Improved SA while the original one approximates GED for around 74% of the graphs at the same time.

As it can be seen in Figures 5.5 and 5.6 the time required to compare dynamic graphs is approximately 10 times more than that of static graphs. The problem here is that the number of nodes and edges of the generated dynamic graphs are significantly higher than static graphs. Based on the statistics from Figures 5.8 and 5.7, 90% of the static graphs have 10 to 2000 nodes while in dynamic graphs 70% of them have more than 2000 nodes. For the number of edges the case is even worse than number of nodes where the percentage of the graphs that have more than 2000 edges are around 90%. In contrast, around 85% of static graphs have less than 2000 edges. Therefore, due to large number of nodes and edges in dynamic graphs, significant amount of time is required to compare them in comparison to static ones.

5.3.1.3 Classification

The purpose of this part is to show the power of our system to detect different malware families. As shown in Figure 5.9 several experiments have been
Figure 5.7: Distribution of number of edges

Figure 5.8: Distribution of number of nodes
done to prove the effectiveness of all system modules. Based on the results our system (Dynamic Modified SA in Figure 5.9) shows persistent results while meeting performance goals of low rejection rate and close to optimal accuracy. At the end of day 5 our system is able to classify samples with average accuracy of 94% while leaving only 2% of samples unlabeled. In contrast, with static graphs the average accuracy would be 68% with 7% unlabeled samples.

The results also prove that the Improved SA performs better than the original SA by 2-6 percent increase in accuracy. In all cases (Static graph, dynamic graphs), the accuracy of using Improved SA approaches the accuracy of the original SA. This is because the Improved SA discovers the shorter edit path and the GED is better approximated which improves the system accuracy significantly.

Moreover, it can be seen that employing unpacker does not improve the accuracy of the system in comparison to using dynamic graphs. The primary drawback of using static unpackers is that they are limited to a set of known packers techniques and are unable to unpack new and unknown packers. Since in our dataset there are a diverse set of packers, the employed unpacker tool only can unpack a small percentage of binaries that employ UPX, NsPack, and Upack packers.
Figure 5.9: Classification Performance
5.3.2 Comparative Evaluation with Classy

Classy applies in-houseunpacker on binaries to unpack them and then feeds them into IDA pro to extract static FCGs. *Simulated Annealing* algorithm has been used to measure similarity between extracted graphs. Then they devised a new online algorithm to cluster incoming graphs. Since they do not mention the unpacker name, we employ PE-explorer [78] to unpack the incoming binaries. To compare the Classy with the proposed system, we employ both dynamic and static graphs as an input to Classy. The reason is to see the effectiveness of both proposed dynamic FCGs and online machine learning method. Results for the comparison are presented in Figure 5.10.

The proposed system outperforms Classy by reaching 94% accuracy and leaving only a few percentage of samples unlabelled while Classy only can reach 60% of accuracy by using static graphs. The results also prove the fact that dynamic FCGs performs better in comparison to static ones. By employing dynamic graphs as input of Classy the accuracy of the system is increased by more than 10%.

There are three reasons behind the low percentage of the Classy accuracy. The first reason is that they use static graphs, which do not reflect the real structure of the malware samples. Even using unpacker tool to unpack the binaries does not help too much in increasing performance of the system since the unpacker is limited to set of known packers techniques. The second reason is their assumption about the clustering algorithm which makes their clustering ineffective. At the beginning of the first step, to determine the
Figure 5.10: Comparison Study
candidate clusters for the incoming sample, they only consider those graphs that have the same number of node as the incoming sample has. Therefore, the algorithm ignores a lot of possible samples that have a different number of nodes but can have similar behavior. Finally, as we mentioned before, their proposed Simulated Annealing cannot find the optimal GED between FCGs which affects the system accuracy.

5.4 Conclusion Remarks

In this chapter we evaluate our system in terms of efficiency and time complexity by applying the proposed system to a large and diverse dataset of executables. To do that, each module of the proposed system is compared to its static competitor. Moreover, we provide a comparative analysis with state-of-the-art static graph based method, namely Classy to present the discrimination power of our dynamic system.

In terms of time requirement, the proposed system requires more time to compare dynamic graphs in comparison to static graphs since the dynamic FCGs have larger number of nodes and edges, which need a significant amount of time for graph matching. However, employing the dynamic graphs instead of static ones significantly increases the system detection accuracy by approximately 20%. The results also proved that the proposed Improved SA performs better in both time complexity and efficiency since it can find the optimal GED in lower time in contrast to original SA.
A comparative analysis with Classy is performed with the aim of comparing the employed classification algorithm with their proposed online clustering algorithm. It was found that employing the static graphs in combination with their clustering results in a significant drop in the classification performance. The superior classification performance of our system indicates its effectiveness in stream malware classification.
Chapter 6

Conclusion and Future Work

6.1 Conclusion

The Inability of signature-based approaches in identifying new malicious symptoms has motivated many researchers to build automated detection approaches employing dynamic analysis. We found that static graph-based approaches are only partially able to distinguish between different malware families.

In this research, we proposed a new dynamic graph based classification system by employing the power of dynamic analysis to generate function call graphs. We devise a new algorithm to generate dynamic FCGs. The generated graphs are compared together using the improved version of Simulated Annealing which is a GED approximation algorithm. The proposed Improved SA algorithm improves the system accuracy while maintaining low computa-
tional complexity. The generated dynamic graph reflects the real structure of a malware sample and therefore has high discriminative power when used by machine learning techniques.

Our comparative analyses confirm effectiveness of our system in classification of stream of samples by reaching an average accuracy of 94% and a minimum number of unlabeled samples (2% of total samples). In general, superior performance of our system stems from (1) the power of dynamic FCGs to reflect real malware behaviour that can be distinguishable among stream of malware samples; and (2) the ability of estimating the optimal GED by using improved version of Simulated Annealing which increase the overall system accuracy.

6.2 Future Work

- Improve time complexity of graph matching module: The primary drawback of proposed system is the significant amount of time that is required to compare graphs. This is due to both number of graph nodes and edges and also the comparison algorithm which is the Improved SA in our case. We determined that there are two ways to solve this problem. Recently a new GED algorithm has been proposed with the very low time complexity in the pattern recognition area [26, 27]. They employ Hausdorff edit distances to approximate the graph edit distances and their experimental evaluation proved that their algorithm
performs very well in terms of accuracy and time complexity by employing pattern recognition datasets. So as our future work, we would like to employ the proposed algorithm to compare the graphs to get the lower comparison time. Another way to improve our time complexity is using combination of graphics processing unit (GPU) and Hadoop. He et al. [40] show that this combination can improve the time required of many graph problem significantly. Therefore, to make our proposed system more scalable we will reconfigure our system based on the combination of GPU and Hadoop.

- Employ a combination of CFG and FCG: As we discussed in the literature review, several approaches have employed a combination of FCG and CFG to get higher accuracy. Since our proposed system is capable to generate both graph types, the system will be evaluated by employing combination of CFG and FCG graphs as input to system.

- Use of optimization methods to reduce the size of dynamic graphs: As mentioned before the primary overhead of system is related to large size of the graphs. Xu et al. [88] Employs several optimization techniques to reduce the size of the FCG graphs. Employing or proposing several optimization techniques to significantly reduce the size of the graphs will reduce the time complexity of the system significantly.

- Experiment with real streams of malware: A future extension would be to collect and analyze more recent and sophisticated samples.
Bibliography


[40] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In


[92] Zongqu Zhao. A virus detection scheme based on features of control flow graph. In *Artificial Intelligence, Management Science and Electronic*
Vita

Candidate’s full name: Hossein Hadian Jazi

University attended:
Master of Computer Science
University of New Brunswick
2013-2016

Bachelor of Computer Engineering- Software Engineering
Isfahan University of Technology
2004-2008

Publications:
Conference Presentations:


Fakhroddin Noorbehi Bahani, Elaheh Biglar Beigi Samani, and Hossein Hadian Jazi. “A Novel Method for Learner Assessment Based on Learner An-
