Aalto University
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Joris Kinable

Malware Detection Through Call Graphs

Master's Thesis
Espoo, June 30, 2010

Supervisors: Professor Pekka Orponen, Aalto University
Professor Danilo Gligoroski, Norwegian University of Science and Technology
Advisor: Alexey Kirichenko, M.Sc., F-Secure Corporation

AALTO UNIVERSITY

Faculty of Information and Natural Sciences
Degree Programme of Security and Mobile Computing

ABSTRACT OF
MASTER'S THESIS

Author: Joris Kinable

Title of thesis:
Malware Detection Through Call Graphs

Date:	June 30, 2010	**Pages:** 7 + 54
Professorship:	Theoretical Computer Science	**Code:** T-79
Supervisors:	Professor Pekka Orponen Professor Danilo Gligoroski	

Each day, anti-virus companies receive large quantities of potentially harmful executables. Many of the malicious samples among these executables are variations of earlier encountered malware, created by their authors to evade pattern-based detection. Consequently, robust detection approaches are required, capable of recognizing similar samples automatically.
In this thesis, malware detection through call graphs is studied. In a call graph, the functions of a binary executable are represented as vertices, and the calls between those functions as edges. By representing malware samples as call graphs, it is possible to derive and detect structural similarities between multiple samples. The latter can be used to implement generic malware detection schemes, which can proactively detect existing versions of the malware, as well as future releases with similar characteristics.
To compare call graphs mutually, we compute pairwise graph similarity scores via graph matchings which minimize an objective function known as the Graph Edit Distance. Finding exact graph matchings is intractable for large call graph instances. Hence we investigate several efficient approximation algorithms. Next, to facilitate the discovery of similar malware samples, we employ several clustering algorithms, including variations on k-medoids clustering and DBSCAN clustering algorithms. Clustering experiments are conducted on a collection of real malware samples, and the results are evaluated against manual classifications provided by virus analysts from F-Secure Corporation. Experiments show that it is indeed possible to accurately detect malware families using the DBSCAN clustering algorithm. Based on our results, we anticipate that in the future it is possible to use call graphs to analyse the emergence of new malware families, and ultimately to automate implementing generic protection schemes for malware families.

Keywords:	Malware detection, Graph Edit Distance, Graph Similarity, Classification, DBSCAN
Language:	English

Acknowledgements

In front of you lies a completed Master thesis; a product of 5 months intensive writing, coding and experimenting. And even though the front page mentions only a single author, clearly this work could not have been completed without the continuous support of others.
First of all, I would like to express my gratefulness towards my supervisors, and in particular to Professor Pekka Orponen. He provided many good viewpoints, interesting discussions, and above all invaluable comments on my work with an unprecedented precision.
This thesis has been written as part of a mutual effort of Nokia Corporation, F-Secure Corporation, and Aalto University. At times, this collaboration has been challenging since each party had its own priorities, visions and goals, but overall it has been a truly inspiring and rewarding work experience. The data security company F-Secure provided us with the required call graph samples, allowing us to conduct experiments on real-world data. Credit for the coordination and sample production go respectively to Alexey Kirichenko and Gergely Erdélyi. Furthermore, they provided useful feedback during numerous occasions. In this context, also my colleagues play an important role. Especially with Orestis Kostakis I have had many fine conversations.
Next, I would also like to thank Karin Keijzer for her feminine view on this thesis, i.e. suggestions for colors, layout and formatting, and of course her patience to listen to my endless monologues and discussions regarding the thesis content. Finally, also my family and friends have to be acknowledged for their moral support. My father, Dirk Kinable, deserves special mentioning as he has always been willing to share his extensive linguistic knowledge with me.

This work was supported by TEKES as part of the Future Internet Programme of TIVIT (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT).

Abbreviations and Acronyms

ABS	Absorbing States
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
FLIRT	Fast Library Identification and Recognition Technology
GA	Genetic Algorithm
GED	Graph Edit distance
GMI	Generalized Matrix Inversion
IAT	Import Address Table
MCS	Maximum Common Subgraph
MI	Matrix Inversion
RWP	Random Walk Probability (vector)
SE	Sum of error
SSE	Sum of Squared Error
TR	Transient states

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In an era where information is at the center of our society, cyber criminality is faring well. Backed by large organizations, operating across multiple countries, cyber criminals are nearly untraceable [27]. Jurisdictional issues caused by the borderlessness of the Internet further hamper combating cyber criminality effectively, hence rendering it a very attractive crime scene [27]. Security companies fight an ongoing war against this criminality. On a daily basis, tens of thousands samples with potentially harmful executable code are submitted for analysis to the data security company F-Secure Corporation [24]. Similarly, Symantec Corporation report in their latest Internet Threat Report [42] that to protect against malware threats, a total of 5,724,106 new malicious code signatures were added to the signature database in 2009.
Clearly, to deal with these vast amounts of malware, autonomous systems for protection, detection and desinfection are required. However, in practice automated detection of malware is hindered by code obfuscation techniques such as packing or encryption of the executable code. Furthermore, cyber criminals constantly develop new versions of their malicious software to evade pattern-based detection by anti-virus products. In fact there already exist sophisticated self-modifying virusses, as well as tools to quickly produce variations of the same malware [42].
For each incoming sample of executable code, an anti-virus company typically poses three questions:

1. Is the sample malicious or benign?
2. Has the sample been encountered before, possibly in a modified form?
3. Does the sample belong to a known malware family?

Analogous to the human immune system, the ability to recognize malware families and in particular the common components responsible for the malicious behavior of the samples within a family would allow anti-virus products to proactively detect both known samples as well as future releases of samples belonging to the same malware family. To facilitate the recognition of similar samples or commonalities among multiple samples which have been subject to modification, a high-level structure, i.e. an abstraction, of the samples is required. One such abstraction is the call graph. A call graph is a graphical representation of a binary executable in which functions are modeled as vertices, and calls between those functions as edges [40].

This thesis, written as part of a joint effort of Aalto University, F-Secure Corporation, and Nokia Corporation under the Future Internet Programme [1], deals with the detection of malware through call graphs. So far, only a limited amount of research has been published on automated malware identification and classification through call graphs. Flake [13] and later Dullien and Bochum [11] describe approaches to find subgraph isomorphisms within control flow graphs, by mapping functions from one flow graph to the other. Functions which could not be reliably mapped have been subject to change. Via this approach, the authors of both papers can for instance reveal differences between versions of the same executable or detect code theft. Additionally, the authors of [11] suggest that security experts could save valuable time by only analyzing the differences among variants of the same malware. Preliminary work on call graphs specifically in the context of malware analysis has been performed by Carrera and Erdélyi [8]. To speed up the process of malware analysis, Carrera and Erdélyi use call graphs to reveal similarities among multiple malware samples. Furthermore, after deriving similarity metrics to compare call graphs mutually, they apply the metrics to create a small malware taxonomy using a hierarchical clustering algorithm. Briones and Gomez [6] continued the work started by Carrera and Erdély. Their contributions mainly focus on the design of a distributed system to compare, analyse and store call graphs for automated malware classification. Finally, the first large scale experiments on malware comparisons using real malware samples were recently published in [21]. Additionally, the authors of [21] describe techniques for efficient indexing of call graphs in hierarchical databases to support fast malware lookups and comparisons.

In this thesis, we further explore the potentials of call graph based malware identification and classification. A subdivision in three parts is made. The first part (Chapters 2, 3) introduces call graphs in more detail and investigates graph similarity metrics to compare malware via their call graph representations. At the basis of call graph comparisons lie graph matching algorithms. Exact graph matchings are expensive to compute, and hence

we resort to approximation algorithms. Part two (Chapters 4, 5) discusses several heuristics to support the graph matching algorithms. In addition, the accuracy of several graph matching algorithms using varying heuristics are studied. Finally, in part three (Chapter 6) the graph similarity metrics are used for automated detection of malware families via clustering algorithms on a collection of real malware call graphs.

Chapter 2

Introduction to Call Graphs

Many anti-virus products deploy a pattern-based detection approach: virus scanners are built around large databases containing byte sequences which uniquely characterize individual malware samples. These byte sequences are used to recognize malware hidden in files or system areas [43]. Maintaining these databases, as well as rapid detection of malware are no trivial tasks, especially when the malware writers deploy techniques to hinder pattern-based detection [43]. Consequently, robust detection techniques are required which can recognize variants of the same malware instances.
To identify both benign and malicious programs, or variations of the same program, in a generic way, an abstraction of the software has to be derived. One such abstraction is the call graph [40]. A call graph is a directed graph whose vertices, representing the functions a program is composed of, are interconnected through directed edges which symbolize function calls [40]. A vertex can represent either one of the following two types of functions:

1. Local functions, implemented by the program designer.
2. External functions: system and library calls.

Local functions, the most frequently occurring functions in any program, are written by the programmer of the binary executable. 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 [21], call graphs are formally defined as follows:

Definition 1. *(Call Graph): A call graph 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,*

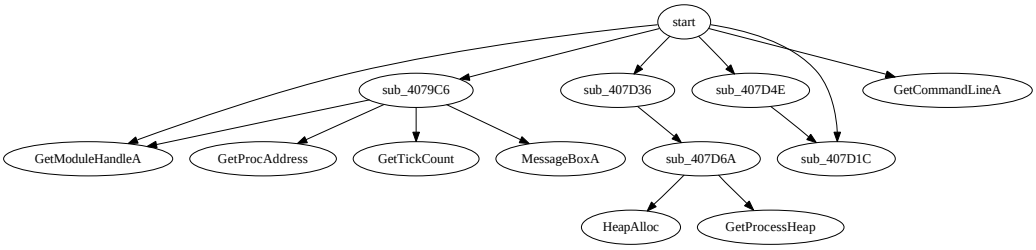


Figure 2.1: Example of a small call graph, derived from a malware sample with IDA Pro. Function names starting with 'sub' denote local functions, whereas the remaining functions are external functions.

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).

Call graphs are generated from a binary executable through static analysis of the binary with disassembly tools [24]. First, obfuscation layers are removed, thereby unpacking and, if necessary, decrypting the executable. Next, a disassembler like IDA Pro [19] is used to identify the functions and assign them symbolic names. Since the function names of user written functions are not preserved during the compilation of the software, random yet unique symbolic names are assigned to them. External functions however, have common names across executables. In case an external function is imported dynamically, one can obtain its name from the Import Address Table (IAT) [35, 28]. When, on the other hand, a library function is statically linked, the library function code is merged by the compiler into the executable. If this is the case, software like IDA Pro's FLIRT [20] has to be used to recognize the standard library functions and to assign them the correct canonical names. Once all functions, i.e. the vertices in the call graph, are identified, edges between the vertices are added, corresponding to the function calls extracted from the disassembled executable.

Chapter 3

Graph Matching

3.1 Basic terminology and notation

This section provides a short overview of the terminology and notation used in this thesis. A *graph* $G = (V, E)$ [49] is composed of *vertices* V and *edges* $E \subseteq V \times V$, representing functions and function calls respectively in the context of call graphs. The *order* of a graph G is the number of vertices $|V(G)|$ in G. In this thesis, we are only dealing with directed graphs; an edge (also known as *arc*) is denoted by its endpoints as an ordered pair of vertices. The first vertex of the ordered pair is the *tail* of the edge, and the second vertex is the *head*. A vertex v is *adjacent* to vertex u, if $(u, v) \in E$. The *out-degree* $d^+(v)$ of vertex v is the number of vertices adjacent to v, i.e. the number of edges which have a tail in v. Similarly, the *in-degree* $d^-(v)$ equals the number of edges with their head in v. Finally, the *degree* $d(v)$ of vertex v equals $d^+(v) + d^-(v)$. The *out-neighborhood* (successor set) $N^+(v)$ of vertex v consists of the vertices $\{w|(v, w) \in E\}$, and the *in-neighborhood* (predecessor set) is the set $\{w|(w, v) \in E\}$.

3.2 Graph matching techniques

Detecting malware through the use of call graphs requires means to compare call graphs mutually, and ultimately, means to distinguish call graphs representing benign programs from call graphs based on malware samples. Mutual graph comparison is accomplished with graph matching.

Definition 2. *(Graph matching): For two graphs, G and H, of equal order, the graph matching problem is concerned with finding a one-to-one mapping*

(bijection) $\phi : V(G) \rightarrow V(H)$ *that optimizes a cost function which measures the quality of the mapping.*

In general, graph matching involves discovering structural similarities between graphs [37] through one of the following techniques:

1. Finding graph isomorphisms
2. Detecting maximum common subgraphs (MCS)
3. Finding minimum graph edit distances (GED)

An exact graph isomorphism for two graphs, G and H, is a bijective function $f(v)$ that maps the vertices $V(G)$ to $V(H)$ such that for all $i, j \in V(G)$, $(i, j) \in E(G)$ if and only if $(f(i), f(j)) \in E(H)$ [49]. Detecting the largest common subgraph for a pair of graphs is closely related to graph isomorphism as it attempts to find the largest induced subgraph of G which is isomorphic to a subgraph in H. Consequently, one could interpret an exact graph isomorphism as a special case of MCS, where the common subgraph encompasses all the vertices and edges in both graphs. Finally, the last technique, GED, calculates the minimum number of edit operations required to transform graph G into graph H.

Definition 3. *(Graph edit distance): The graph edit distance is the minimum number of elementary edit operations required to transform a graph G into graph H. A cost is defined for each edit operation, where the total cost to transform G into H equals the edit distance.*

Note that the GED metric depends on the choice of edit operations and the cost involved with each operation. Similar to [50, 37, 21], we only consider vertex insertion/deletion, edge insertion/deletion and vertex relabeling as possible edit operations.
We can now show that the MCS problem can be transformed into the GED problem. Given is the shortest sequence of edit operations *ep* which transforms graph G into graph H, for a pair of unlabeled, directed graphs G and H. Apply all the necessary destructive operations, i.e. edge deletion and vertex deletion, on graph G as prescribed by *ep*. The maximum common subgraph of G and H equals the largest connected component of the resulting graph. Without further proof, this reasoning can be extended to labeled graphs.

For the purpose of identifying, quantifying and expressing similarities between malware samples, both MCS and GED seem feasible techniques. Unfortunately, MCS is proven to be an NP-Complete problem [16], from which the NP-hardness of GED optimization follows by the prevous argument (The latter result was first proven in [50] by a reduction from the subgraph isomorphism problem). Since exact solutions for both MCS and GED are computationally expensive to calculate, a large amount of research has been devoted to fast and accurate approximation algorithms for these problems, mainly in the field of image processing [15] and for bio-chemical applications [36, 48]. The remainder of this Section serves as a brief literature review of different MCS and GED approximation approaches.

A two-stage discrete optimization approach for MCS is designed in [14]. 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 [14], the authors of [48] also rely on a two-stage optimization procedure, however contrary to [14], their algorithm tolerates errors in the MCS matching. A genetic algorithm approach to MCS is given in [45]. Finally, a distributed technique for MCS based on message passing is provided in [5].

A survey of three different approaches to perform GED calculations is conducted by Neuhaus, Riesen, et. al. in [37, 38, 32]. They first give an exact GED algorithm using A* search, but this algorithm is only suitable for small graph instances [32]. Next, A*-Beamsearch, a variant of A* search which prunes the search tree more rigidly, is tested. 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. In [22], the GED problem is formulated as a Binary Linear Program, but the authors conclude that their approach is not suitable for large graphs. Nevertheless, they derive algorithms to calculate respectively 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 the work of Justice and Hero in [22], the authors of [50] developed new polynomial algorithms which find tighter upper and lower bounds for the GED problem.

3.3 Graph similarity

In general, a virus consists of multiple components, some of which are new and others which are reused from other viruses [24]. The virus writer will test his creations against several anti-virus programs, making modifications along the way until the anti-virus programs do not recognize the virus anymore. Furthermore, at a later stage the virus writer might release new, slightly altered, versions of the same virus. Descriptions of several possible modification techniques used by malware writers to avoid detection by anti-virus software are found in [7] and [43].
In this Section, we will describe how to determine the similarity between two malware samples, based on the similarity $\sigma(G, H)$ of their underlying call graphs. As will become evident shortly, the graph edit distance plays an important role in the quantification of graph similarity. After all, the extent to which the malware writer modifies a virus or reuses components should be reflected by the edit distance.

Definition 4. *(Graph similarity): The similarity $\sigma(G, H)$ between two graphs G and H indicates the extent to which graph G resembles graph H and vice versa. The similarity $\sigma(G, H)$ is a real value on the interval [0,1], where 0 indicates that graphs G and H are identical whereas a value 1 implies that there are no similarities. In addition, the following constraints hold: $\sigma(G, H) = \sigma(H, G)$ (symmetry), $\sigma(G, G) = 0$, and $\sigma(G, K_0) = 1$ where K_0 is the null graph, $G \neq K_0$.*

Before we can attend to the problem of graph similarity, we first have to revisit the definition of a graph matching as given in the previous Section. To find a bijection which 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 vertex sets $V(G)$ and $V(H)$ can be supplemented with dummy vertices ϵ such that the resulting sets $V'(G)$, $V'(H)$ are of equal size. A mapping of a vertex v in graph G to a dummy vertex ϵ 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 ϕ, we can define three cost functions: VertexCost, EdgeCost and RelabelCost.

VertexCost The number of deleted/inserted vertices: $|\{v : v \in [V'(G) \cup V'(H)] \wedge [\phi(v) = \epsilon \vee \phi(\epsilon) = v]\}|$.

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

RelabelCost The number of mismatched functions, i.e. the number of external functions in G and H which are mapped against different external functions or local functions.

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

$$\lambda_\phi(G, H) = VertexCost + EdgeCost + RelabelCost \tag{3.1}$$

Note that, as mentioned before, finding the minimum GED, i.e. $\min_\phi \lambda_\phi(G, H)$, is an NP-hard problem, but can be approximated. The latter is elaborated in the next Section.

Finally, the similarity $\sigma(G, H)$ of two graphs is obtained from the graph edit distance $\lambda_\phi(G, H)$:

$$\sigma(G, H) = \frac{\lambda_\phi(G, H)}{|V(G)| + |V(H)| + |E(G)| + |E(H)|} \tag{3.2}$$

3.4 Graph edit distance approximation

Finding a graph matching ϕ which minimizes the graph edit distance is proven to be an NP-Complete problem [50]. Indeed, empirical results show that finding such a matching is only feasible for low order graphs, due to the time complexity [32]. As a solution, Riesen and Bunke propose to use a $(|V(G)|+|V(H)|) \times (|V(H)|+|V(G)|)$ cost matrix C, which gives the cost of mapping a vertex $v \in V'(G)$ to a vertex $v \in V'(H)$ [38, 37]. Next, Munkres' algorithm [31, 25] (also known as the Hungarian algorithm), which runs in polynomial time, is applied to find an exact one-to-one vertex assignment which minimizes the total mapping cost. Similar to [21], we will use this procedure to find a graph matching for two call graphs. For a given pair of call graphs, we first investigate which external functions they have in common. These functions can be directly mapped one-to-one. For the remaining functions, we create a cost matrix, which is used to find the vertex mapping using Munkres' algorithm. The general structure of the cost matrix C is as

follows [37]:

$$C = \left[\begin{array}{cccc|cccc} c_{1,1} & c_{1,2} & \cdots & c_{1,m} & c_{1,\epsilon} & \infty & \cdots & \infty \\ c_{2,1} & c_{2,2} & \cdots & c_{2,m} & \infty & c_{2,\epsilon} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \infty \\ c_{n,1} & c_{n,2} & \cdots & c_{n,m} & \infty & \cdots & \infty & c_{n,\epsilon} \\ \hline c_{\epsilon,1} & \infty & \cdots & \infty & 0 & 0 & \cdots & 0 \\ \infty & c_{\epsilon,2} & \ddots & \vdots & 0 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \infty & \vdots & \ddots & \ddots & 0 \\ \infty & \cdots & \infty & c_{\epsilon,m} & 0 & \cdots & 0 & 0 \end{array}\right]$$

The top left quadrant of cost matrix C gives the cost $C_{v,w}$ of matching a real vertex $v \in V(G)$ to a real vertex $w \in V(H)$. Detailed descriptions for several cost functions which calculate $C_{v,w}$ are given in the next Chapter. The top right and bottom left quadrants of cost matrix C give the cost of matching a real vertex against a dummy vertex. In particular, $C_{i,\epsilon}$ equals the cost of deleting a vertex, whereas $C_{\epsilon,j}$ represents the cost of inserting a vertex. The authors of [21] propose to choose $C_{i,\epsilon} = C_{\epsilon,j} = 1$. However, from our experiments it appears that better results are obtained if $C_{i,\epsilon}$ and $C_{\epsilon,j}$ are set to $d(v_i), v_i \in V(G)$, and $d(v_j), v_j \in V(H)$ respectively. The choice of these costs is explained by the observation that mapping a real vertex v to a dummy vertex, i.e. vertex deletion, will result in an increase of the $EdgeCost$ parameter in the edit distance metric (Equation 3.1) equal to the degree of v. Finally, the cost of mapping a dummy vertex against another dummy vertex is set to 0 in the bottom right quadrant of C.

3.5 Genetic search

The Hungarian algorithm discussed in the previous Section has a runtime complexity of $O(|V|^3)$, where $|V|$ is the vertex cardinality of the largest graph under comparison [25]. For large call graphs, this potentially poses a problem, since it is imperative that the graph comparison is performed fast to be applicable for malware detection and identification. Another issue is the lack of information about the accuracy achieved when the GED is approximated via the Hungarian algorithm. Therefore, as a counterweight to the Hungarian algorithm, an alternative approach is implemented which relies on a Genetic search algorithm to find a vertex mapping which minimizes the GED.

Genetic algorithms (GAs) are categorized as a special group of search algorithms inspired by Darwin's evolution theory. A GA takes a set of candidate solutions, which is a subset of the entire search space, as input. The set of candidate solutions is called a population or generation, and an individual in the population is called a chromosome. The GA produces successive generations by mutating and recombining parts of the best currently known chromosomes [29].

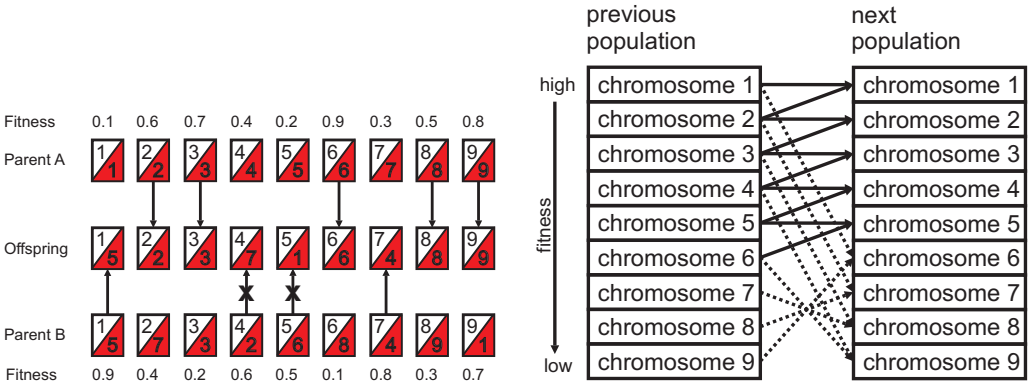
The GA we use to search for a vertex mapping which minimizes the GED is based on the work of Wang and Isshii [47]. For a given pair of graphs, G and H, such that $|V(G)| \leq |V(H)|$, each chromosome represents an injective matching of the vertices from graph G to the vertices in graph H. Each chromosome can be thought of as a list of $|V(G)|$ genes, where each gene represents a unique mapping of a vertex $v \in V(G)$ onto a vertex $w \in V(H)$. The 'fitness' of a gene indicates how well vertex $v \in V(G)$ maps onto vertex $w \in V(H)$. To calculate the fitness of a gene, one can use the same cost functions as used to calculate the entries of the cost matrix C for the Hungarian algorithm as discussed in the previous Section. Examples of possible cost functions are given in the next chapter. Finally, the quality of a complete matching, i.e. the fitness of a chromosome, is assessed via the GED (Equation 3.1). A chromosome has a higher fitness compared to the fitness of another chromosome if the vertex mapping it represents results in a lower GED than that of the other chromosome.

Chromosomes for the initial population are generated at random; vertices in graph G are matched randomly against vertices in graph H, under the restriction that the result is an injective mapping. Furthermore, to ensure diversity of the individuals in the population, no two chromosomes in the initial population can be identical.

A new population is created by performing crossover operations and mutations on the chromosomes of the current population. Crossover operations can be interpreted as a recombination of two parent chromosomes, thereby obtaining a single child chromosome, which inherits the qualities of both parents. For a given pair of chromosomes, A, B, where the fitness of chromosome A is larger or equal to the fitness of chromosome B, the crossover operation is now defined as follows (Figure 3.1a) [47]:

1. Copy all genes from chromosome A which have a higher or equal fitness compared to the corresponding genes in chromosome B to the offspring. Copy the remaining genes from chromosome B to the offspring.

2. The offspring should be an injective mapping; a single vertex in graph H cannot be mapped to multiple vertices in graph G. This require-



(a) Crossover example. A gene, represented by a box, gives a mapping of $v \in V(G)$ (white) against $w \in V(H)$ (red).

(b) Selection of chromosomes for crossover.

Figure 3.1: Genetic selection and crossover (Source: [47])

ment can however be violated when one copies the remaining genes from chromosome B to the offspring after having the fitter genes selected from chromosome A (Figure 3.1a). These violations are solved by selecting an unused vertex from graph H to be matched against the respected vertex in graph G.

3. Finally, the fitness of the new offspring is compared against the fitness of parent A. If the fitness of parent A is higher, we discard the new offspring and we use parent A as the new offspring. This procedure ensures that the total fitness of the population remains the same or improves during each cycle in the evolution.

The selection of chromosomes for the crossover operation is depicted in Figure 3.1b. The general idea behind this selection scheme is to combine chromosomes of lower fitness with chromosomes of higher fitness to discover new chromosomes of even higher fitness, while simultaneously preserving diversity among the chromosomes [47]. First the chromosomes in a population are sorted according to decreasing fitness. Next, the crossover procedure is performed on the ith and $(i+1)$th chromosome for $i = [1, 2, ..., \frac{n+1}{2}]$, as well as on the jth and $(n+1-j)$th chromosome for $j = [1, 2, .., \frac{n}{2}]$, where n equals the population size (Figure 3.1b).
Finally, to further improve the population diversity, mutations are performed. For a fixed number of chromosomes, genes are changed at random, while preserving the requirement that the resulting chromosome should be an injective mapping.

Chapter 4

Vertex Matching

The approximation algorithm for the graph edit distance as discussed in Chapter 3.4 attempts to find the smallest edit distance by solving a least cost assignment problem on a cost matrix C. The entries in this matrix represent the cost of matching $i \in V(G)$ to $j \in V(H)$. Similarly, the GA discussed in Section 3.5 uses these vertex match costs to direct the search toward an optimal solution. This chapter derives several cost functions as estimators of $C_{i,j}$: the cost of matching vertex i to vertex j. Clearly, more accurate cost estimations will enable us to find better graph matchings and hence more accurate edit distances.

4.1 Structural Matching

The cost of matching a pair of nodes, $C_{i,j}$ could equal the relabeling cost as defined for the graph edit distance in Equation 3.1:

$$C^{rel}(i,j) = \begin{cases} 0 & \text{if } V_f(i) = V_f(j) = 0 \\ 0 & \text{if } V_f(i) = V_f(j) = 1 \ \wedge \ V_n(i) = V_n(j) \\ 1 & \text{otherwise} \end{cases} \tag{4.1}$$

Using this relabeling cost function, Munkres' algorithm is capable of matching identical external functions in a pair of graphs, but the local functions pose a problem because the relabeling cost function yields no information about the different local functions. As a solution, the authors of [21, 50] independently suggest to embed structural information in the matching cost of two functions. The following equation achieves the latter by also taking

the neighborhoods of vertices (functions) i and j into consideration:

$$\begin{aligned} C_{i,j} =& C^{rel}(i,j)+ \\ & d^+(i)+d^+(j)-2\times(N^+(i)\wedge N^+(j))+ \\ & d^-(i)+d^-(j)-2\times(N^-(i)\wedge N^-(j)) \end{aligned} \tag{4.2}$$

where the notation $N \wedge M$ denotes the similarity of the neighborhoods N and M, defined as follows:

$$\begin{aligned} N\wedge M = max\{ & \sum_{i\in N}(1-C^{rel}(i,P(i))| \\ & P: N\rightarrow M \quad \textit{(injective)}\} \end{aligned}$$

In short, the above equation makes the assumption that if two functions i, and j are identical, then they should also invoke the same functions. Similarly, if i and j indeed represent the same function, it is likely that they are also called upon by functions with a high mutual similarity.

4.2 Random Walk Probability Vectors

In the previous Subsection, all local functions in the neighborhood of vertices v and w are considered identical due to the lack of information about the functions and their canonical names, and hence there is no relabeling cost induced when two local functions are matched (Equation 4.1). In reality however, two local functions can rarely be considered identical. In this Section, a more fine-grained approach is developed to estimate the similarity between two local functions with a higher accuracy. In this context, similarity is defined as follows:

Definition 5. *(Vertex similarity): The similarity σ(v,w) between two vertices $v \in V(G)$ and $w \in V(H)$ indicates the extent to which function V_n(v) resembles function V_n(w). Similar to the definition of graph similarity, σ(v,w) is a real value on the interval [0,1].*

Before the similarity between two vertices can be calculated, we first need to establish a metric which allows for mutual comparison between two local functions. Let S and T be partitions of V such that $S = \{v \in V(G) | N^+(v) = \emptyset \vee N^+(v) = \{v\}\}$ and $T = V(G) \setminus S$. The vertices in S are terminal nodes: they do not call other functions except possibly themselves, and are therefore

represented as leaves in the graph. Typically, all external functions belong to the set S, in addition to some local functions. This partitioning allows us to interpret the call graph as an absorbing Markov chain, where T contains the transient states, and S the absorbing states (Figure 4.1).

Definition 6. *(Absorbing Markov Chain): An absorbing Markov chain is a weighted directed graph G, where a path exists from each vertex $v \in V(G)$ to an absorbing state s. The weight of an edge $w(i,j)$ in an absorbing Markov chain denotes the probability of moving from state i to state j. A state $s \in V(G)$ is called absorbing if it is impossible to leave it, i.e. $d^+(v) = 0 \vee N^+(v) = \{v\}$ [1]. For all absorbing states i, $w(i,i) = 1$ [17]. Furthermore, $w(i,j) > 0$ for all $(i,j) \in E$ and $\sum_j w(i,j) = 1$ for all $i \in V$.*

To characterize a vertex v_i, a probability vector P^{v_i} is associated with it[2]: $P^{v_i} = (p^{v_i}_{v_1}, p^{v_i}_{v_2}, ..., p^{v_i}_{v_k})$, where $p^{v_i}_{v_j}$ denotes the probability that a random walk from vertex v_i terminates in absorbing state v_j. We will refer to these vectors as Random walk probability (RWP) vectors. Note that for each RWP vector, the following equation holds:

$$\sum_{j=0}^{|P^{v_i}|} p^{v_i}_{v_j} = 1 \tag{4.3}$$

Definition 7. *(Simple Random Walk): Given a graph G. A simple random walk from vertex $v_0 \in G$ is an alternating sequence $v_0, e_1, v_1, ..., e_k, v_k$ of edges and vertices where the probability of moving from vertex v_i via edge e_{i+1} to vertex v_{i+1} equals $\frac{1}{d^+(v_i)}$. Let $p^{v_0}_{v_k}$ denote the probability that a simple random walk which starts in v_0 ends in v_k. The probability $p^{v_0}_{v_k}$ satisfies the recursive formula:*

$$p^{v_0}_{v_k} = \frac{\sum_{w \in N^+(v_0)} p^w_{v_k}}{d^+(v_0)}$$
$$p^{v_k}_{v_k} = 1$$

RWP vectors provide an abstract means to characterize transient functions. Intuitively, two functions with the same RWP vectors have a high probability

[1] Strictly speaking, true absorbing Markov chains do not have states with $d^+(v) = 0$; all absorbing states obey $N^+(v) = \{v\}$. However, this extension of the concept of absorbing states allows us to interpret call graphs as absorbing Markov chains.

[2] This characterization idea is proposed by prof Pekka Orponen [33]. Efficient calculations of these vectors are explored by the author as part of this thesis work.

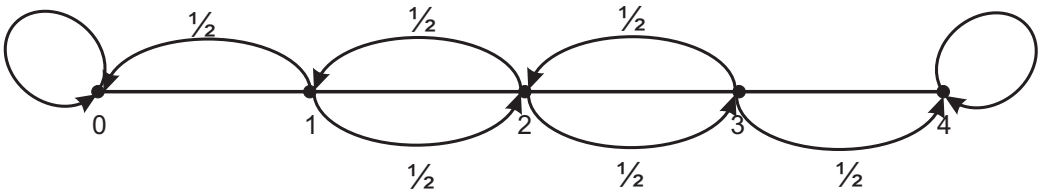


Figure 4.1: Absorbing Markov chain. Vertices 0,4 are absorbing states, whereas 1,2 and 3 are the transient states. Source:[17]

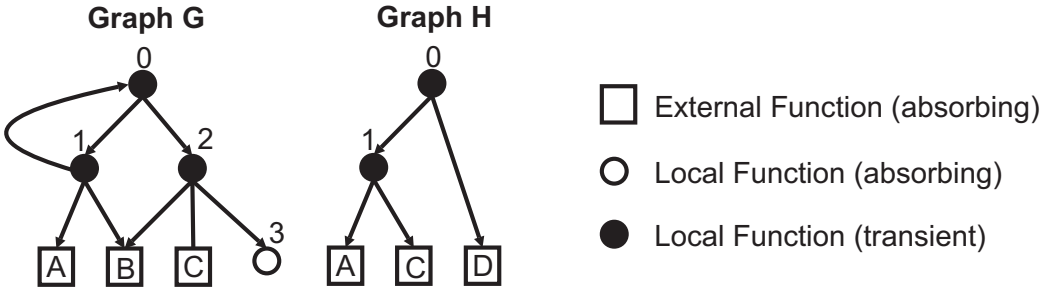


Figure 4.2: Two call graphs with a mutual absorbing state set $S = \{A, C, \epsilon\}$.

to have similar functionality and behave in the same way. Consequently, a high similarity score should be assigned to them compared to two functions with very different RWP vectors.
To calculate the similarity of two vertices from different graphs, their similarity vectors should have the same dimensions. The latter requirement can be met by choosing the set with absorbing states S as $S = \{v \in V(G) | V_f(v) = 1\} \cap \{v \in V(H) | V_f(v) = 1\} \cup \{\epsilon\}$. Here ϵ is a dummy vertex which symbolizes all absorbing states outside the intersection of the external functions in graphs G and H. An example has been depicted in Figure 4.2.

The problem which now arises is how to calculate the RWP vectors for a given graph G in an efficient and scalable fashion. After all, call graphs can contain thousands of vertices and edges. The answer lies in the use of a stochastic transition matrix and the theory behind absorbing Markov chains [17]. First, obtain a $|V(G)| \times |V(G)|$ stochastic transition matrix P, where

$$P_{i,j} = \begin{cases} \frac{1}{d^+(i)} & \text{if } i \in T,\ j \in (S \cup T),\ (i,j) \in E(G) \\ 1 & \text{if } i, j \in S,\ V_f(i) = V_f(j) \\ 0 & \text{otherwise} \end{cases}$$

In accordance with definition 6, each entry $P_{i,j}$ in P represents the probability of moving from state i to j. Now one can reorder the states in the transition matrix so that the transient states come first. The result should

be a transition matrix which can be represented in the following canonical form [17]:

$$P = \begin{array}{c} \\ TR. \\ ABS. \end{array} \begin{array}{c} \begin{array}{cc} TR. & ABS. \end{array} \\ \left(\begin{array}{c|c} Q & R \\ \hline 0 & I \end{array} \right) \end{array} \tag{4.4}$$

Here, 0 is an $|S| \times |T|$ zero matrix, and I an $|S| \times |S|$ identity matrix.

Definition 8. *Let P be the stochastic transition matrix of an absorbing Markov chain. Then $P^n_{i,j}$ of the matrix P^n is the probability to reach state j, starting from state i, in n state transitions [17].*

In particular, for the purpose of the RWP vectors, we are interested in submatrix R^n (Eq. 4.4) of matrix P^n, when $n \to \infty$; the long-term probabilities of reaching absorbing state $j \in S$ from a transient state $i \in T$. A matrix containing these long-term probabilities is obtained via the following equation [17]:

$$N = (I - Q)^{-1} \tag{4.5}$$

$$B = N \times R \tag{4.6}$$

Here, I and Q are the submatrices as defined in Eq. 4.4. In the context of absorbing Markov chains, matrix N (Eq. 4.5) is sometimes referred to as the *fundamental matrix*. An entry $n_{i,j}$ in N gives the expected number of times transient state j occurs in a sequence of state transitions which starts in state i, before the sequence terminates in an absorbing state [17]. Finally, a row vector B_i in the result matrix B (Eq. 4.6) represents the RWP vector for transient state i. As an example, Table 4.1 shows the RWP vectors for graph G in Figure 4.2.

Given two transient states $v \in V(G)$, and $w \in V(H)$, and their corresponding RWP vectors P^v, respectively P^w, the similarity score $\sigma(v, w)$ as defined in Definition 5 can now be calculated using the total variation distance, closely related to the ℓ_1 norm, over the RWP vectors:

$$\sigma(v, w) = \frac{1}{2} \sum_{i=1}^{|P^w|} |p^v_i - p^w_i|, \quad 0 \leq \sigma(v, w) \leq 1 \tag{4.7}$$

For two external functions, $v \in V(G)$ and $w \in V(H)$, $\sigma(v, w) = 0$ if they represent the same external function. In all other cases, $\sigma(v, w)$ equals 1.

		TR.			ABS.		
		0	1	2	A	C	*
TR.	0	0	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
	1	$\frac{1}{3}$	0	0	$\frac{1}{3}$	0	$\frac{1}{3}$
	2	0	0	0	0	$\frac{1}{3}$	$\frac{2}{3}$
ABS.	A	0	0	0	1	0	0
	C	0	0	0	0	1	0
	*	0	0	0	0	0	1

(a) The stochastic transition matrix of graph G depicted in Figure 4.2

	A	C	*
0	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{3}{5}$
1	$\frac{6}{15}$	$\frac{1}{15}$	$\frac{8}{15}$
2	0	$\frac{1}{3}$	$\frac{2}{3}$

(b) The RWP vectors, obtained from table 4.1a

Table 4.1: Example of a stochastic transition matrix and corresponding RWP vectors, based on graph G in Figure 4.2

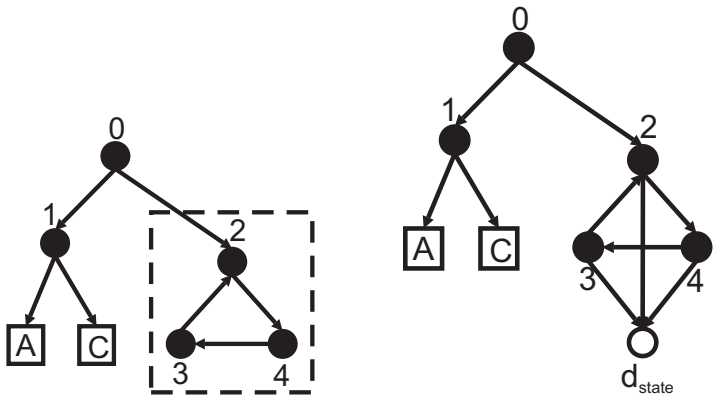
Throughout the reasoning in this Subsection, we made the following implicit assumption: a call-graph can always be converted to an absorbing Markov chain as defined in Definition 6. Unfortunately, there exist call-graphs which violate Definition 6; not all states have a path to an absorbing state. An example of such a violation is depicted in Figure 4.3a; the strongly connected component marked by the dashed box has no outgoing edges to an absorbing vertex. We will refer to these structures as 'nontrivial sinks' because from a structural point of view a nontrivial sink behaves as an absorbing state. Once a state transition reaches a nontrivial sink, it is impossible to get out of the nontrivial sink again.

When a nontrivial sink is present in the call graph, Equation 4.5 is rendered invalid because $I - Q$ results in a singular matrix, which one cannot invert. Two solutions exist to deal with this problem:

1. Relax the matrix inversion in Eq. 4.5 through the use of Generalized matrix inversion [30].

2. Remove the nontrivial sink structures.

The generalized matrix inversion (GMI) [30] preserves most of the properties of the normal matrix inversion (MI). In fact, for a nonsingular matrix the result obtained via GMI is identical to the result of MI. However, for a singular matrix, GMI produces a non-unique estimate of a matrix inverse. After applying GMI to a singular matrix, the result can be directly plugged into equation 4.6. Although this approach does not require any preprocessing of



(a) Sink present (b) Sink removed

Figure 4.3: When a stochastic matrix is created based on the graph in Figure 4.3a, the result is a singular matrix caused by the sink marked by the dashed box. The nontrivial sink can be removed with Algorithm 2 resulting in the graph depicted in Figure 4.3b

the call graph, the downside is that the vectors in matrix B (Eq. 4.6) no longer obey the property of RWP vectors as defined in Eq. 4.3, which makes it much harder to interpret the RWP vectors. Therefore, a more natural and computationally inexpensive solution is to remove the sink structures altogether. The latter can be achieved by identifying the nontrivial sinks, and connecting them with an absorbing local function; after all, a nontrivial sink structure behaves exactly the same as an absorbing local function. Note that this procedure does not change the RWP vectors. Identifying vertices which are part of sink structures can be achieved using Algorithm 1. Next, the nontrivial sinks can be removed by adding a dummy vertex d_{state}, representing an absorbing state, to $V(G)$, and adding edges from all identified vertices to the dummy vertex (Algorithm 2). The result of Algorithms 1 and 2 on the graph in Figure 4.3a is depicted in 4.3b.

Algorithm 1: Identify vertices in sink structures

Input: Call graph G

Output: Set of vertices which belong to sinks in the call graph

```
1  Queue open ← {v ∈ V(G) | d⁺(v) = 0 ∨ N⁺(v) = {v}};
2  visited ← ∅;
   Move in an upward sweep through the graph, starting at the leaves
   (absorbing states), thereby marking all reachable vertices
3  while open ≠ ∅ do
4      v ← pop(open);
5      visited ← visited ∪ {v};
6      foreach w ∈ N⁻(v) do
7          if w ∉ visited then
8              open ← open ∪ {w};
9
10 return V(G)\visited
```

Algorithm 2: Neutralize all sink structures in a graph

Input: Set of vertices S which are part of sinks

Output: A graph G' where all sinks are removed

Add a dummy absorbing state d_{state} to the graph, and connect all vertices in S to the dummy vertex.

```
1  V(G') ← V(G) ∪ {d_state};
2  E(G') ← E(G);
3  foreach v ∈ S do
4      E(G') ← E(G') ∪ {(v, d_state)};
5  return G'(V, E)
```

Chapter 5

Graph similarity: experimental results

Chapter 3 introduced two algorithms (Sections 3.4 and 3.5) which attempt to find a vertex mapping (bijection) for a given pair of graphs which minimizes the Graph Edit Distance (Equation 3.1). In order to find a mapping that approximates the minimum GED as well as possible, both algorithms require good cost estimations of matching one function against the other. Two estimators are presented in Chapter 4. The first estimator (Section 4.1) uses a relabeling cost function and a neighborhood comparator (Equation 4.2), whereas the second utilizes Random Walk Probability vectors (Section 4.2). The main purpose of this chapter is to evaluate the performance of these two estimators, as well as two additional hybrid versions. The evaluation is conducted on a set of 194 call graphs provided by the data security company F-Secure Corporation.

5.1 Graph similarity metrics

In Section 3.4, the Hungarian algorithm has been introduced, which finds a vertex mapping of minimum cost for a given pair of graphs. Using Equations 3.1 and 3.2, one can calculate the GED and corresponding graph similarity score for the resulting vertex mapping.
Figure 5.1 shows the similarity scores obtained via the Hungarian algorithm with four different vertex matching cost estimators on a set of 1000 unique graph pairs selected randomly from our call graph data set. The outcomes of the first estimator, based on relabeling costs and neighborhood comparisons

(Equation 4.2) as presented in Section 4.1 are depicted by the green curve. Since this estimator is also applied in [50, 21], we will use it as a reference against which we situate the outcomes of the three other estimators.
Section 4.2 argues that the relabeling cost function (Equation 4.2) used in the former estimator is not an accurate approach to compare functions, since it cannot distinguish between local functions. Therefore, Section 4.2 introduces Random Walk Probability vectors to uniquely characterize a function. The vertex similarity scores obtained via Equation 4.7 can be directly inserted into the cost matrix used by the Hungarian algorithm. The resulting pairwise graph similarities are shown with purple in Figure 5.1. It is interesting to observe that this approach allows us to find for some graphs lower Graph Edit Distances, and consequently higher pairwise similarities. However, in most comparisons this approach is outperformed by the former vertex match cost estimator.
Based on the previous results, we attempted to combine Equations 4.2 and 4.7 into a new estimator to further improve the accuracy of the similarity scores. Combination of the two Equations is performed by replacing the relabeling cost function (Equation 4.1) in Equation 4.2 with Equation 4.7 which calculates the vertex similarity scores via the RWP vectors. The results of the newly obtained estimator are depicted by the dark blue line in Figure 5.1. Unfortunately, one can observe that the latter results are very similar to those obtained via the relabeling cost function (Figure 5.1, green line). Only for a few graph pairs a smaller GED was found. Finally, purely from an experimental point of view, we replaced the relabeling function by Equation 4.7 only in the neighborhood comparison, i.e. in the 'where' clause, of Equation 4.2, while preserving the relabeling function (Equation 4.1) in the main body of Equation 4.2. The result, depicted by the red line in Figure 5.1 does not show any significant deviation from the previous estimator (light blue line). Just for a small number of graph pairs a marginal accuracy increase is observed.
Comparing the approaches which use the more computationally expensive similarity scores obtained via the RWP vectors against the cheap relabeling cost function as proposed in [50, 21], we have to conclude that the accuracy gain is too low to outweigh the extra computation time involved.
As an alternative to the Hungarian algorithm, a genetic search algorithm is presented in Section 3.5. The algorithm requires two parameters: the population size and a mutation rate. The former has been fixed to 100 chromosomes, and a single gene is mutated in 30% of the chromosomes in the population after crossover has been applied. Unfortunately, the average runtime of the genetic search algorithm appears to be significantly longer than the runtime of the Hungarian algorithm to find results of equal accuracy.

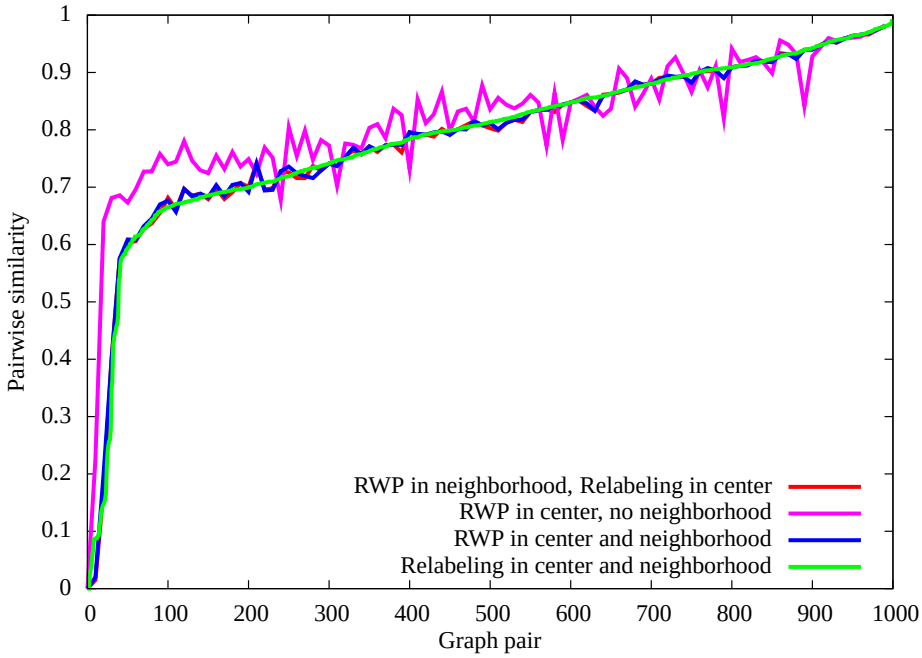


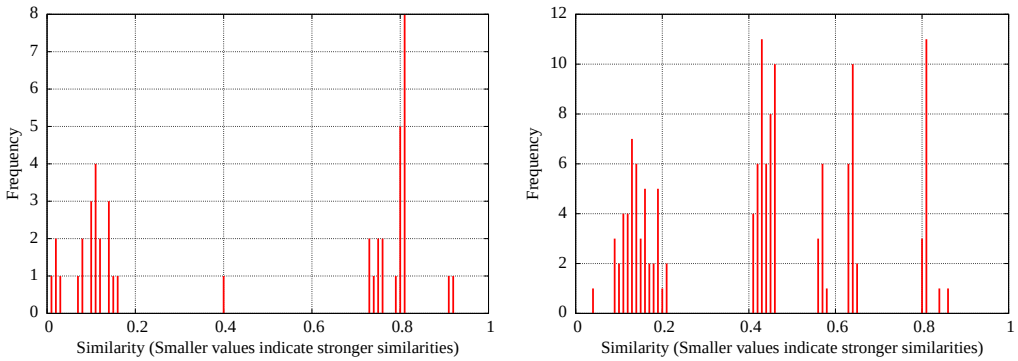
Figure 5.1: Comparison of four estimators used in cooperation with the Hungarian algorithm in an attempt to find, for a given pair of graphs, a vertex matching which minimizes the GED.

Various changes to both the population size as well as the mutation rate did not change these results. The remainder of this thesis will therefore utilize Equation 4.2 in combination with the Hungarian algorithm to estimate the minimum GED and corresponding graph similarity.

5.2 Malware family analysis

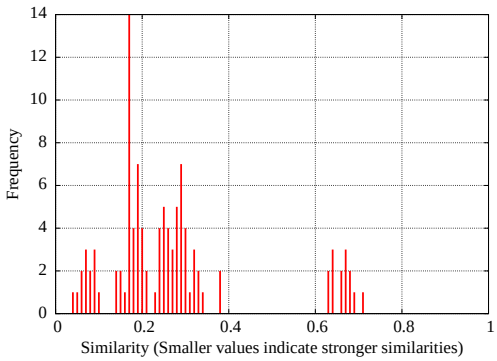
An important goal of the graph comparisons is the ability to recognize malware samples with strong similarities. Before we turn to the subject of fully automated malware identification and classification in the subsequent Sections, this Section briefly investigates the meaning of graph similarity in the context of malware. To this extent, the earlier mentioned data set consisting of 194 call graphs of malware samples has been partitioned into 24 malware families by analysts of F-Secure Corporation. This classification is performed by an analyst via semantic evaluation of the malware behavior. The samples within each of the families are believed to have a mutual similarity. We have taken four of the larger families and compared the graphs within each

family mutually (Figure 5.2). The Baidu family, for instance, consists of 10 samples, and hence we can make $\frac{10^2-10}{2} = 45$ pairwise comparisons. The resulting similarity scores are depicted in the frequency chart (Figure 5.2a). Note that we do not compare a graph against itself, since this always results in a similarity score of 0 ($\sigma(G, G) = 0$), in accordance with Equation 3.2. Ideally, all samples within a family would exhibit a strong mutual similarity, but as one can observe from Figure 5.2 this is not necessary the case. Each of the four families contain some samples which are significantly distinct from the other samples in the same family. For identification purposes, it is not strictly required that a sample has a high similarity to all other samples in its family, as long as there are no samples in other families with a higher resemblance because this would lead to classification errors. Figure 5.3 compares samples between families. Indeed, Figure 5.3 shows that the selected families are significantly dissimilar. A high similarity among the samples within a family, together with a high dissimilarity between different families would highly simplify malware identification and classification. The next chapter will examine graph classification in greater detail; based on the graph similarity scores, we will attempt automated partitioning and family recognition.

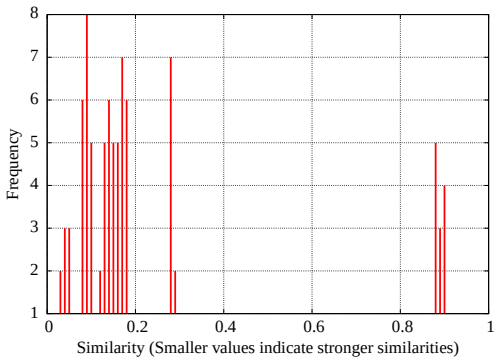


(a) Baidu (10 samples)

(b) Boaxxe (17 samples)

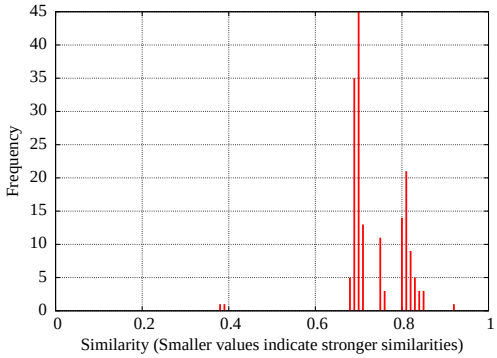


(c) Pushbot (15 samples)

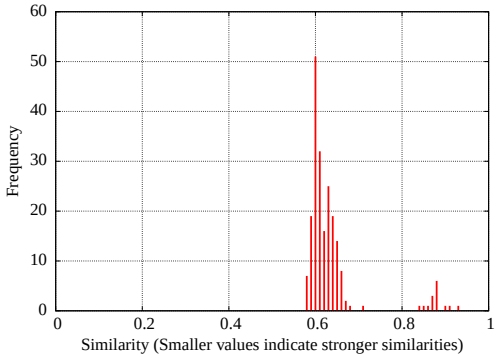


(d) Systemhijack (14 samples)

Figure 5.2: Intra family comparison. The samples inside a family are compared mutually. Typically, one would expect a high similarity among the samples within a single family.



(a) Boaxxe versus Baidu



(b) Systemhijack versus Pushbot

Figure 5.3: Inter family comparison. The samples among families are compared. Typically, one would expect no or few similarities between families.

Chapter 6

Clustering

To support the identification process, an important step is to be able to classify malware samples, thereby grouping similar samples together. This chapter focuses on the clustering of malware samples into malware families.

6.1 *k*-medoids clustering

One of the most commonly used clustering techniques is k-means clustering. The formal description of k-means clustering is summarized as follows [3, 10]:

Definition 9. *(k-means Clustering): Given a data set χ with samples, where each sample $x \in \chi$ is represented by a vector of parameters. k-means clustering attempts to group all samples into k clusters. For each cluster $C_i \in C$, a cluster center μ_{C_i} can be defined, where μ_{C_i} is the mean vector, taken over all the samples in the cluster. The objective function of k-means clustering is to minimize the total squared Euclidean distance $||x - \mu_{C_i}||^2$ between each sample $x \in \chi$, and the cluster center μ_{C_i} of the cluster the sample has been allocated to:*

$$min \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_{C_i}||^2$$

The above definition assumes that for each cluster, it is possible to calculate a mean vector, the cluster center (also known as centroid), based on all the samples inside a cluster. However, with a cluster containing call graphs, it is not a trivial procedure to define a mean vector. Consequently, instead of defining a mean vector, a call graph inside the cluster is selected as the cluster center. More specifically, the selected call graph has the most commonalities,

i.e. the highest similarity, with all other samples in the same cluster. This allows us to reformulate the objective function:

$$min \sum_{i=1}^{k} \sum_{x \in C_i} \sigma(x, \mu_{C_i})$$

where $\sigma(G, H)$ is the similarity score of graphs G and H as discussed in Chapter 3. The latter algorithm is more commonly known as a k-medoids clustering algorithm, where the cluster centers μ_{C_i} are referred to as 'medoids'. Since finding an exact solution in accordance with the objective function has been proven to be NP-hard [9], our k-medoids clustering utilizes an iterative approach [1]. First, the k-medoids clustering algorithm finds an arbitrary solution, after which the algorithm attempts to find better solutions until no more improvements occur. The pseudo-code of the k-medoids algorithm is given in Algorithm 3.

Algorithm 3: The k-medoids clustering algorithm

Input: Number of clusters k, set of call graphs χ.
Output: A set of k clusters C

```
foreach C_i ∈ C do
    Initialize μ_{C_i} with an unused sample from χ;
repeat
    Classify the remaining |χ| − k call graphs. Each sample x ∈ χ
    is put in the cluster which has the most similar cluster medoid;
    foreach C_i ∈ C do
        Recompute μ_{C_i};
until The objective function converges;
return C = C_0, C_1, ..., C_{k−1}
```

In [26], a formal proof on the convergence of k-means clustering with respect to its objective function is given. To summarize, the authors of [26] proof that the objective function decreases monotonically during each iteration of the k-means algorithm. Because there are only a finite number of possible clusterings, the k-means clustering algorithm will always obtain a result which corresponds to a (local) minimum of the objective function. Since k-medoids

[1] The author of this thesis designed this algorithm as a variation of the k-means clustering algorithm. Later he discovered that this variation has also been independently proposed by Park and Jun [34] as an improved version of the Partitioning Around Medoids clustering algorithm [23].

clustering is directly derived from k-means clustering, the proof also applies for k-medoids clustering.
To initialize the cluster medoids, we use three different algorithms. The first approach, Algorithm 4, selects the centroids at random from χ.
Arthur and Vassilvitskii observed in their work [3] that k-means clustering, and consequently also k-medoids clustering, is a fast, but not necessarily accurate approach. In fact, the clusterings obtained through k-means clustering can be arbitrarily bad [3]. In their results, the authors of [3] conclude that bad results are often obtained due to a poor choice of the initial cluster centroids, and hence they propose a novel way to select the initial centroids, which considerably improves the speed and accuracy of the k-means clustering algorithm [3]. The algorithm, referred to by the authors as k-means++, is given in Algorithm 5.
Finally, the last algorithm to select the initial centroids will be used as a means to assess the quality of the clustering results. To assist the k-medoids clustering algorithm, the initial medoids are selected manually. We will refer to this initialization technique as "Trained initialization".

Algorithm 4: Initializing cluster medoids: uniform random medoid selection

Input: Number of clusters k, set of call graphs χ.
Output: k cluster medoids μ_{C_i}

```
for i = 1 to k do
    μ_{C_i} ← random graph x ∈ χ;
    χ ← χ\{μ_{C_i}};
return μ_{C_0}, μ_{C_1}, ..., μ_{C_{k-1}}
```

6.2 Clustering performance analysis

In this Section, we will test and investigate the performance of the clustering approaches, in combination with the graph similarity scores obtained via the GED algorithm discussed in Chapter 3. The data set χ we will use consists of 194 samples which are manually classified by F-Secure Corporation into 24 families. Evaluation of the cluster algorithms is performed by comparing the obtained clusters against these 24 partitions. To get a general impression of the samples, the call graphs in our test set contain on average 234 nodes and 488 edges. The largest sample has 748 vertices and 1875 edges. Family sizes vary from 2 samples to 17 samples.

Algorithm 5: Initializing cluster centroids: k-means++

Input: Number of clusters k, set of call graphs χ.
Output: k cluster medoids μ_{C_i}

1 Take μ_{C_0} at random from χ;
2 **repeat**
3 Take a new medoid μ_{C_i} from χ, choosing $x \in \chi$ with probability $\frac{D(x)^2}{\sum_{y \in \chi} D(y)^2}$;
Here, $D(x)$ is the similarity of $x \in \chi$ and the most similar medoid μ_{C_i} which has been selected so far.
4 **until** *k medoids have been chosen*;
5 **return** $\mu_{C_0}, \mu_{C_1}, ..., \mu_{C_{k-1}}$

Before k-medoids clustering can be applied on the data collection, we need to select a suitable value for k. Let $k_{optimal}$ be the natural number of clusters present in the data set. Finding $k_{optimal}$ is not a trivial task. One could argue that $k_{optimal}$ equals 24, the number of families F-Secure provided us with. This, however, is not necessarily a correct assumption. As an analogy, we could attempt to perform clustering on a large group of people. The clustering criteria could, among others, be eye color, or family ties. Both criteria yield valid, but possibly different clusters of varying sizes. Hence, $k_{optimal}$ depends on the selected cluster criteria. The same holds for the call graph clusters as provided by F-Secure. Although two malware samples could be very dissimilar from a graph-structural point of view, both samples could have similar behavior or malicious purposes. Consequently, a malware analyst might decide to categorize both samples in the same family.
Since $k_{optimal}$ is unknown, we attempt to find it by trying multiple values for k, and measuring the quality of the obtained clustering (Figure 6.1). In Figure 6.1, the average distance $\bar{d}(x_i, \mu_{C_i})$ between a sample x_i in cluster C_i and the medoid of that cluster μ_{C_i} is plotted against the number of clusters in use. Note that each time k-medoids clustering is repeated, the algorithm could yield a different clustering due to the randomness in the algorithm. Hence, for a given number of clusters k, we run k-medoids clustering 50 times, and average $\bar{d}(x_i, \mu_{C_i})$. When the number of clusters k equals 1, then the average distance $\bar{d}(x_i, \mu_{C_i})$ is maximal. $\bar{d}(x_i, \mu_{C_i})$ converges to 0 when k increases towards the number of samples in the data set. Ideally, when one plots $\bar{d}(x_i, \mu_{C_i})$ against an increasing number of clusters, one should observe a quick decreasing $\bar{d}(x_i, \mu_{C_i})$ on the interval $[k = 1, k_{optimal}]$ and a slowly decreasing value on the interval $[k_{optimal}, k = |\chi|]$. Unfortunately, figure 6.1

shows a steadily decreasing curve for $k = [1, 50]$, which makes it impossible to deduce $k_{optimal}$ from Figure 6.1. A more in-depth discussion on how to find the $k_{optimal}$ using alternative quality metrics is given in the next Section. For now, we assume that $k_{optimal}$ equals 24 as follows from the manual partitioning of the samples by F-Secure.
When comparing the different initialization methods of k-medoids clustering, based on Figure 6.1, one can indeed conclude that k-means++ yields better results than the randomly initialized k-medoids algorithm. Furthermore, the best results are obtained with Trained clustering where a member from each of the 24 predetermined malware families is chosen as the initial medoid of a cluster.
Figures 6.2, 6.3 depict heat maps of two possible clusterings of the sample data. Each square in the heat map denotes the presence of samples from a given malware family in a cluster. As an example, cluster 0 in Figure 6.2 comprises 86% Ceeinject samples, 7% of Runonce samples and 7% of Neeris samples. The family names are invented by data security companies and research labs and serve as a means to distinguish families, but a detailed discussion about the characteristics of each family is beyond the scope of this thesis.
Figure 6.2 shows the results of k-medoids clustering with Trained initialization. The initial medoids are selected by manually choosing a single sample from each of the 24 families identified by F-Secure. The clustering results are very promising: nearly all members from each family end up in the same cluster (Figure 6.2). Only a few families, such as Baidu and Boaxxe, are scattered over multiple clusters, which is in accordance with our findings in Section 5.2. Figure 6.3 shows the clustering results of k-means++ [2], without the use of vertex matching. Clearly, the clusterings are not as accurate as with our trained k-medoids algorithm; samples from different families are merged into the same cluster. Nevertheless, in most clusters samples originating from a single family are prominently present. Yet, before one can conclude whether k-means++ clustering is a suitable algorithm to perform call graph clustering, one first needs an automated procedure to discover, or at the minimum estimate with reasonable accuracy, $k_{optimal}$. This will be investigated in the next Section.

[2] A similar figure for randomly initialized k-medoids clustering is omitted due to its reduced accuracy with respect to k-means++.

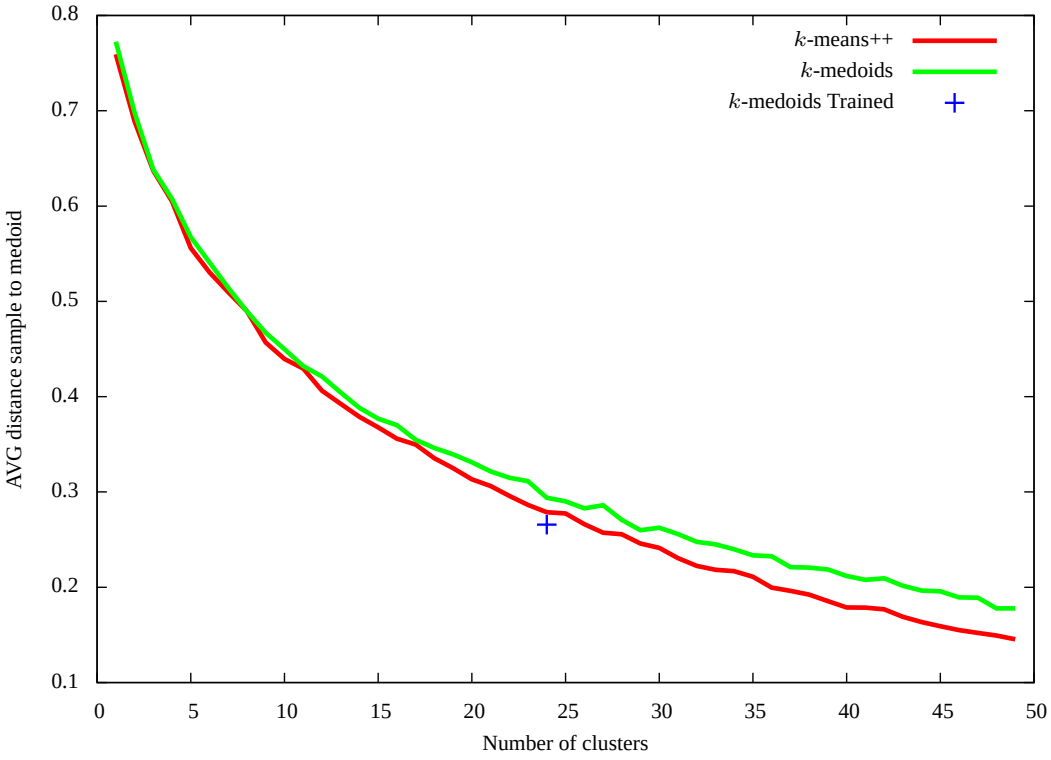


Figure 6.1: Quality of clusters. The average distance $\bar{d}(x_i, \mu_{C_i})$ between a sample x_i in cluster C_i and the cluster's medoid μ_{C_i} is averaged over 50 executions of the k-means algorithm.

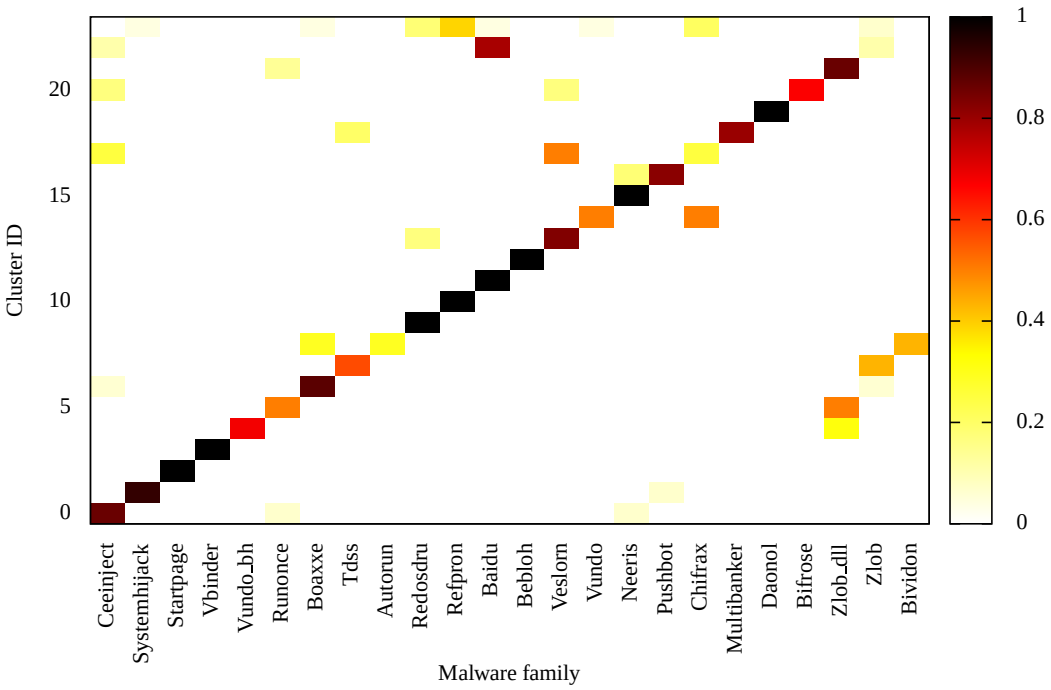


Figure 6.2: A heat map depicting a non-unique clustering of 194 samples in 24 clusters using trained k-means clustering. For this particular result, $\bar{d}(x_i, \mu_{C_i}) = 0.221$. The color of a square depicts the extent to which a certain family is present in a cluster.

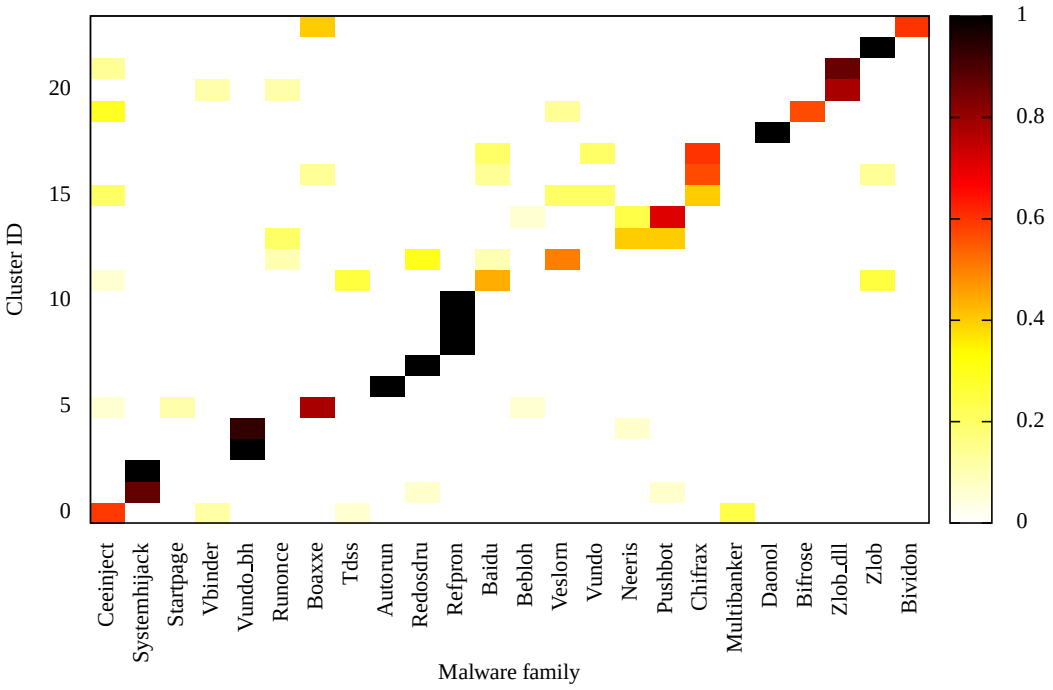


Figure 6.3: A heat map depicting a non-unique clustering of 194 samples in 24 clusters using k-means++ clustering. For this particular result, $\bar{d}(x_i, \mu_{C_i}) = 0.240$.

6.3 Determining the number of clusters

In the previous chapter, a brief discussion about the optimal number of clusters $k_{optimal}$ has been included. However, from the graph depicted in Figure 6.1, it is not evident which value should be chosen as $k_{optimal}$, based on the average distance from a sample to its corresponding cluster centroid. In this Section, three additional techniques for finding $k_{optimal}$ are explored.

6.3.1 Sum of (Squared) Error

The Sum of Error (SE_p), measures the amount of scatter in a cluster. The general formula of SE_p is:

$$SE_p = \sum_{i=1}^{k} \sum_{x \in C_i} (d(x_i, \mu_{C_i}))^p \tag{6.1}$$

In this equation, $d(x, y)$ is a distance metric which measures the distance between a sample and its corresponding cluster centroid (medoid) as a positive real value. Here we choose $d(x_i, \mu_{C_i}) = 100 \times \sigma(x_i, \mu_{C_i})$. The most commonly used value for p in Equation 6.1 equals 2. The resulting equation is known as the Sum of Squared Error (SSE) [44]. The power p can be altered to penalize outliers, i.e. vertices which are relatively distant from their cluster medoids, more or less severely.

6.3.2 Silhouette Coefficient

The average distance between a sample and its cluster medoid as used as a performance measure in Section 6.2, measures the cluster *cohesion* [44]. The cluster cohesion expresses how similar the objects inside a cluster are. The cluster *separation* on the other hand reflects how distinct clusters mutually are. An ideal clustering results in well-separated (non-overlapping) clusters with a strong internal cohesion. Therefore, $k_{optimal}$ equals the number of clusters which maximizes both cohesion and separation. The notion of cohesion and separation can be combined into a single function which expresses the quality of a clustering: the silhouette coefficient [44, 39].
For each sample $x_i \in \chi$, let $a(x_i)$ be the average similarity of sample $x_i \in C_k$ in cluster C_k to all other samples in cluster C_k:

$$a(x_i) = \frac{\sum_{x_j \in C_k} \sigma(x_i, x_j)}{|C_k| - 1} \quad (x_i \in C_k)$$

Furthermore, let $b^k(x_i), x_i \notin C_k$ be the average similarity from sample x_i to a cluster C_k which does not accommodate sample x_i.

$$b^k(x_i) = \frac{\sum_{x_j \in C_k} \sigma(x_i, x_j)}{|C_k|} \quad (x_i \notin C_k)$$

Finally, $b(x_i)$ equals the minimum such $b^k(i)$:

$$b(x_i) = \min_k b^k(x_i) \quad k \in \{0, 1, .., |C|\}$$

The cluster for which $b^k(x_i)$ is minimal is the second best alternative cluster to accommodate sample x_i. From the discussion of cohesion and separation, it is evident that for each sample x_i, it is desirable to have $a(x_i) \ll b(x_i)$ so to obtain a clustering with tight, well-separated clusters.
The silhouette coefficient of a sample x_i is defined as:

$$s(x_i) = \frac{b(x_i) - a(x_i)}{max(a(x_i), b(x_i))} \tag{6.2}$$

It is important to note that $s(x_i)$ is only defined when there are 2 or more clusters. Furthermore, $s(x_i) = 0$ if sample x_i is the only sample inside its cluster [39].
The silhouette coefficient $s(x_i)$ in Equation 6.2 always is a real value on the interval $[-1, 1]$. To measure the quality of a cluster, we can simply compute the average silhouette coefficient over the samples of the respected cluster. An indication of the overall clustering quality is obtained by averaging the silhouette coefficient over all the samples in χ.
For a single sample x_i, $s(x_i)$ reflects how well the sample is classified. Typically, when $s(x_i)$ is close to 1, the sample has been classified well. On the other hand, when $s(x_i)$ is a negative value, then sample x_i has been classified into the wrong cluster. Finally, when $s(x_i)$ is close to 0, i.e. $a(x_i) \approx b(x_i)$, it is unclear to which cluster sample x_i should belong: there are at least two clusters which could accommodate sample x_i well.
The silhouette coefficient provides important information about the optimal number of clusters. When the number of clusters is chosen too small such that several natural clusters are merged together, there will be clusters with a relatively bad cohesion. Consequently, the samples in those clusters have a relatively high $a(x_i)$, resulting in a low silhouette coefficient $s(x_i)$ for these samples. If, on the other hand, the number of clusters is chosen too high, some natural clusters will split into two or more clusters. Samples belonging to those natural clusters typically have low values for $b(x_i)$. The latter again causes low silhouette coefficients.

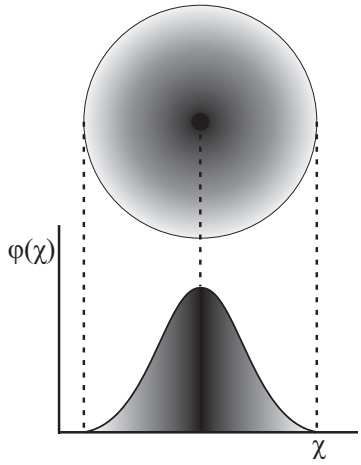


Figure 6.4: Sample point distribution when 2-dimensional data points in a perfectly distributed cluster are projected to a 1-dimensional space.

6.3.3 G-means algorithm

The k-medoids clustering algorithm implicitly assumes that one can partition all samples in such a way that the samples inside a cluster are spherically distributed around a single sample: the cluster medoid. The sample density in the direct vicinity of this sample is high, and decreases the further one moves away from the center. Theoretically, each sample point can be represented by a multidimensional vector, marking the location of the sample with respect to the medoid in a multidimensional space. Consequently, when the samples inside a cluster are projected onto a straight line, a 1-dimensional representation of the cluster is obtained, which hypothetically follows a Gaussian distribution [18] (Figure 6.4).

The G-means clustering algorithm is based on the hypothesis that every cluster has some underlying Gaussian distribution. G-means initiates the k-medoids clustering algorithm with a low value of k. For each resulting cluster, the algorithm tests whether the samples in the cluster follow a Gaussian distribution. If the latter is the case, then the cluster is assumed to be correct, otherwise, the G-means clustering algorithm 'splits' the cluster by selecting two new medoids from the cluster. The k-medoids algorithm is repeated on the entire data collection with the extended set of medoids. It follows that each split operation results in an increase of k by 1. The G-means clustering terminates as soon as each cluster follows a Gaussian distribution. Consequently, by initializing k to 1, the G-means clustering algorithm will automatically attempt to find $k_{optimal}$ [18]. According to experiments conducted by the authors of [18], the G-means algorithm successfully determined

$k_{optimal}$ for several data sets. Unfortunately, no other independent comparative studies on the optimality of G-means clustering have been published. Also note that the resulting $k_{optimal}$ is not necessary identical to the $k_{optimal}$ discovered via the silhouette coefficient or the SE_p as discussed in the previous Sections; there does not exist a consensus on the value $k_{optimal}$ should yield.
To assess whether the data points in a cluster follow a Gaussian distribution, the G-means algorithm utilizes the Anderson-Darling normality statistic [2] which has been corrected for the sample size [41]. The full algorithm has been given in Algorithm 6. Since we do not have a multidimensional vector describing a sample, we use the distance (similarity) from a sample to its medoid, to project our samples onto a 1-dimensional surface. Given the symmetric, bell-shaped Gaussian distribution, we assume that all resulting distances can be placed on the right hand side of the bell curve, and that the sample mean $\bar{X}$ yields a small positive value ϵ, e.g. $0 \leq \epsilon \leq 0.3$. Moreover, the standard normal Cumulative Distribution Function (CDF) $\phi(x)$ approaches 1 for $x = 1$. To split a cluster which does not follow a Gaussian distribution, the authors of [18] suggest two approaches to select two new medoids from the respected cluster. Unfortunately, both approaches require a vector representation of the samples. Using the similarity matrix it would be possible to calculate vector representations of the graphs using multidimensional scaling, however for simplicity we will use the medoid selection procedure as used for the k-means++ algorithm (Algorithm 5) to select the new medoids.

6.3.4 Experimental results

To obtain some insight in the behavior of SE_p and the silhouette coefficient as metrics for establishing $k_{optimal}$, both functions are first applied on an artificial data set. The artificial data set contains 30 objects. Pairwise similarities for these objects (Figure 6.5) are manually chosen such that all objects can be grouped into 5 well-separated clusters of different sizes.

Figure 6.6a plots the number of clusters against the SSE of the artificial data. Since the quality of the clustering is susceptible to the choice of initial centroids of k-means++ clustering, the clustering has been repeated 10000 times for each k. The lowest SSE score obtained in such a sequence, i.e. the best clustering, is used to draw the figure. The optimal number of clusters $k_{optimal}$ is clearly visible in Figure 6.6a due to the so-called 'elbow' at $k = 5$. A similar observation can be made from the silhouette plot in Figure 6.6b: the silhouette coefficient peaks at $k_{optimal}$. Also, the elbow in the graph of

Algorithm 6: Anderson-Darling test for Normality, with sample size correction

Input: List of data samples $X : [x_0, x_1, ..., x_i]$, sample size $n = |X|$
Output: true if X is Gaussian, false otherwise

1 Calculate sample mean $\bar{X}$ and standard deviation σ;
2 Standardize the values: $Y = [\frac{x-\bar{X}}{\sigma} | x \leftarrow X]$;
3 Sort Y ascending;
In the following equation, $\phi(x)$ is the standard normal CDF:
4 $A^2 = -\frac{1}{n}\sum_{i=1}^{n}(2i-1)(\ln\phi(Y_i)) + (\ln(1-\phi(Y_{n+1-i}))) - n$;
Adjust for the sample size [41]:
5 $A^2_* = A^2(1 + \frac{4}{n} - \frac{25}{n^2})$;
Test whether X follows a Gaussian distribution with a significance level of 1%. The critical value 1.8692, which corresponds with the 1% significance level, is taken from [2].
6 **return** $A^2_* \leq 1.8692$

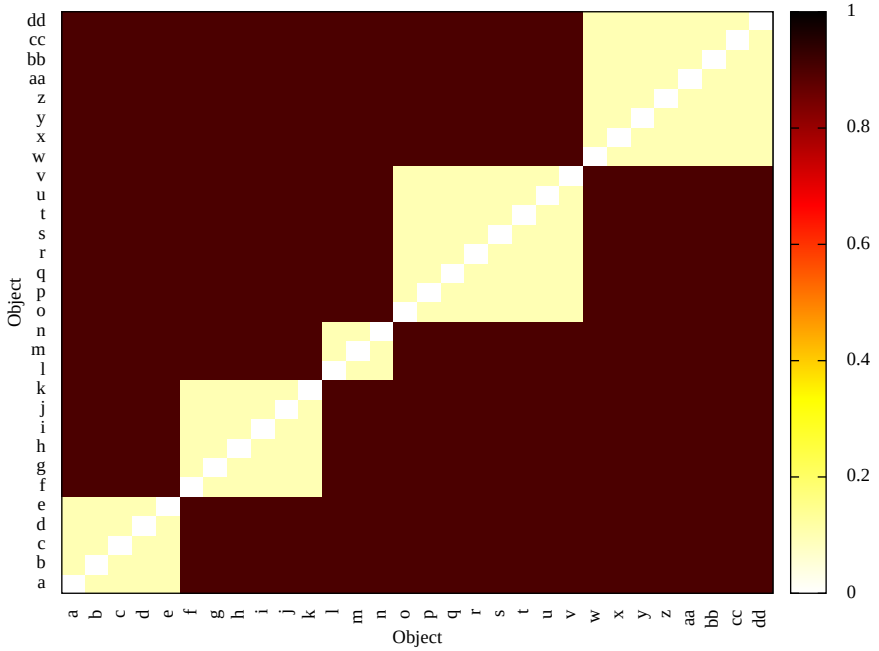
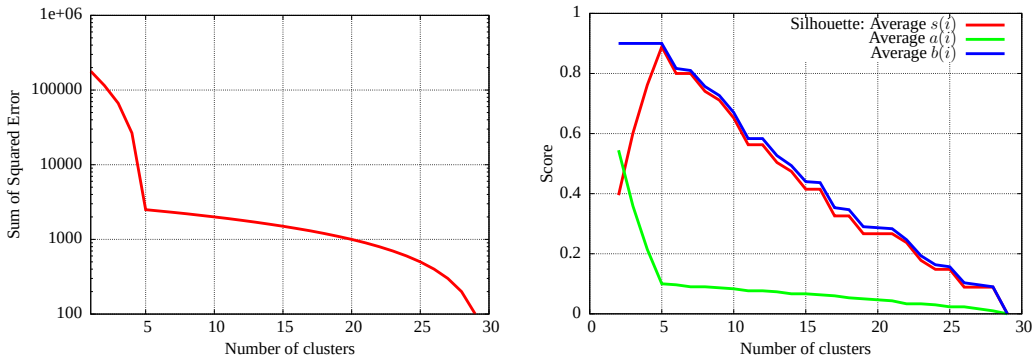


Figure 6.5: Artificially created similarity matrix of 30 objects. There are 5 well-separated, tight clusters of varying size. The colors reflect pairwise similarity scores of the objects.



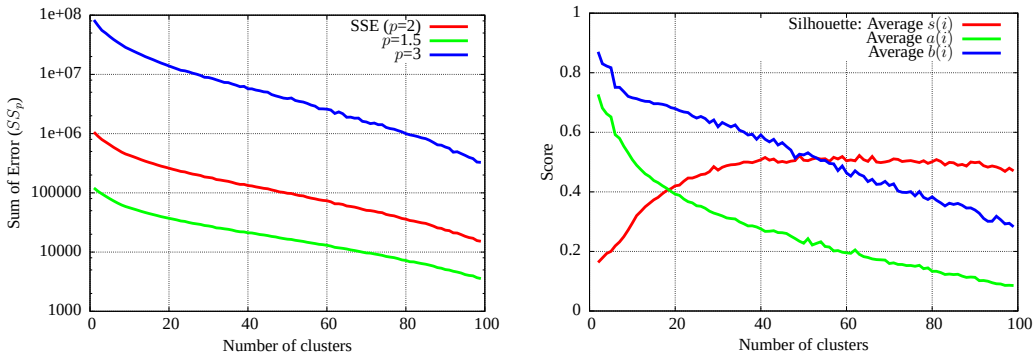
(a) SSE (b) Silhouette Coefficient

Figure 6.6: Finding $k_{optimal}$ in an artificial data set

the average $a(x_i)$ score is informative. When $k < k_{optimal}$, natural clusters are merged together and an increase of k will cause a rapid decrease of the average $a(x_i)$ score. As soon as k becomes greater or equal to $k_{optimal}$, the slope of the $a(x_i)$ curve decreases (Figure 6.6b).

Figure 6.7 plots the same information as Figure 6.6, but this time for the collection of 194 real malware samples. Interestingly, the SE_p curves for different values of p in Figure 6.7a do not reveal an elbow as can be observed in Figure 6.6a for the artificial data. Similarly, no clear peak in the silhouette plot (Figure 6.7b) is visible either, making it impossible to define $k_{optimal}$. Consequently, we have to conclude that it is infeasible to partition the malware samples in cohesive, well-separated clusters based on our graph similarity scores, and hence we cannot obtain the partitioning of the samples in the 24 families as proposed by F-Secure Corporation in an automated fashion.

The absence of well-separated clusters where all samples are ordered in a spherical fashion around the cluster center is also supported by the results obtained via the G-means clustering algorithm. Several applications of the G-means cluster algorithm resulted in 100 or more clusters, which is not a realistic number for just 194 samples. The G-means algorithm could be adapted to test for other distributions besides the Gaussian distribution, but based on our experiments we do not believe that the sample to medoid distances follow any well known distribution in particular.



(a) SE_p for various p (b) Silhouette Coefficient

Figure 6.7: Finding $k_{optimal}$ in the set with 194 pre-classified malware samples.

6.4 DBSCAN clustering

In the previous chapter, we have concluded that the entire sample collection cannot be partitioned in well-defined clusters, such that each cluster is both tight and well-separated. Central to the k-medoid clustering algorithm stands the selection of medoids. A family inside the data collection is only correctly identified by k-medoids if there exists a medoid with a high similarity to all other samples in that family. This, however, is not necessary the case with malware. Instead of assuming that all malware samples in a family are mutually similar to a single parent sample, it is more realistic to assume that malware evolves. In such an evolution, malware samples from one generation are based on the samples from the previous generation. Consequently, samples in generation n likely have a high similarity to samples in generation $n+1$, but samples in generation 0 are possibly quite different from those in generation $n, n \gg 0$. This evolution theory suggests that there are no clusters where the samples are positioned around a single center in a spherical fashion, which makes it much harder for a k-means based clustering algorithm to discover clusters. Although it is not possible to partition all 194 samples in well defined clusters, both Figure 6.2 and Figure 6.3 nevertheless reveal a strong correspondence between the clusters found by the k-medoids algorithm, and the clusters as predefined by F-Secure Corporation. This observation motivates us to investigate partial clustering of the data. For this purpose, we apply the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) clustering algorithm [44, 12]. DBSCAN clustering searches for dense areas in the data space, which are separated by areas of low density. Samples in the low density areas are considered noise and are therefore discarded, thereby ensuring that the clusters are well-separated. An advantage

of DBSCAN clustering is that the high density area's can have an arbitrary shape; the samples do not necessarily need to be grouped around a single center.
DBSCAN distinguishes between three types of sample points:

- Core points. A sample is a Core point if it has more than a predetermined number $MinPts$ of samples in its direct vicinity. The vicinity is specified by a radius Rad: a distance or a similarity score.
- Border points: samples which are not Core points themselves, but are within the radius Rad of a Core point.
- Noise points: all samples which are neither Core points nor Border points.

Formally, we can define the three categories of samples as follows:

- Core points: $P_c = \{x \in \chi, |N_{Rad}(x)| > MinPts\}$, where $N_{Rad}(x) = \{y \in \chi, \sigma(x, y) \leq Rad\}$
- Border points: $P_b = \{x \in (\chi \backslash P_c), \exists y \in P_c : \sigma(x, y) \leq Rad\}$
- Noise points: $P_n = \chi \backslash (P_c \cup P_b)$

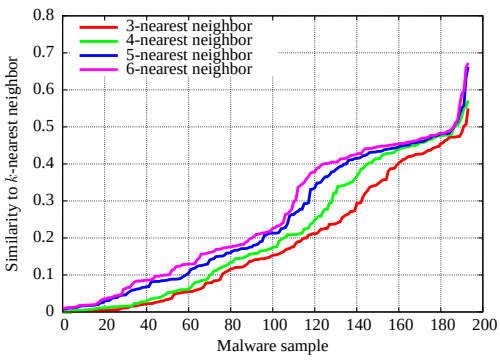
An informal description of the DBSCAN clustering algorithm is given in Algorithm 7.

Algorithm 7: DBSCAN clustering algorithm

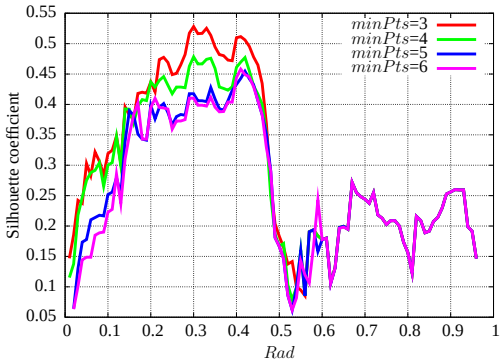
Input: Set of call graphs χ, $MinPts$, Rad
Output: Partial clustering of χ

1 Classify χ in Core points, Border points and Noise;
2 Discard all samples classified as noise;
3 Connect all pairs (x, y) of core points with $\sigma(x, y) \leq Rad$;
4 Each connected structure of core points forms a cluster;
5 For each border point identify the cluster containing the nearest core point, and add the border point to this cluster;
6 **return** *Clustering*

The question now arises how to select the parameters $MinPts$ and Rad. Based on experimental results, the authors of [12] find $MinPts = 4$ to be a good value in general. To determine a suitable value for Rad, the authors



(a) Similarity to k-nearest neighbor, for different values of k, sorted in ascending order.



(b) Silhouette Coefficient for different combinations of Rad and $MinPts$.

Figure 6.8: Finding Rad and $MinPts$

suggest to create a graph where the samples are plotted against the distance (similarity) to their k-nearest neighbor in ascending order. Here k equals $MinPts$. The reasoning behind this is as follows: Core or Border points are expected to have a nearly constant similarity to their k-nearest neighbor, assuming that k is smaller than the size of the cluster the point resides in, and that the clusters are roughly of equal density. Noise points, on the contrary, are expected to have a relatively larger distance to their k-nearest neighbor. The latter change in distance should be reflected in the graph, since the distances are sorted in ascending order.

Figure 6.8a shows the similarity of a malware sample to its k-nearest neighbor, for various k. Arguably, one can observe rapid increases in slope both at $Rad = 2.2$ and $Rad = 4.8$ for all k. A $Rad = 4.8$ radius can be considered too large to apply in the DBSCAN algorithm since such a wide radius would merge several natural clusters into a single cluster. Even though $Rad = 2.2$ seems a plausible radius, it is not evident from Figure 6.8a which value $Minpts$ should yield. To circumvent this issue, DBSCAN clustering has been performed for a large number of $Minpts$ and Rad combinations (Figure 6.8b). For each resulting partitioning, the quality of the clusters has been estimated with the silhouette coefficient. From Figure 6.8b one can observe that the best clustering is obtained for $Minpts = 3$ and $Rad = 0.3$. While comparing Figure 6.8b against Figure 6.8a, it is not clear why $Rad = 0.3$ is a good choice. We however believe that the Silhouette coefficient is the more descriptive metric.

Finally, Figure 6.9 gives the results of the DBSCAN algorithm for $Minpts = 3$ and $Rad = 0.3$ in a frequency diagram. Each colored square gives the

frequency of samples from a given family present in a cluster. The top two lines of the diagram represent respectively the total size of the family, and the number of samples from a family which were categorized as noise. For example, the Boaxxe family contains 17 samples in total, which were divided over clusters 1 (14 samples), 6 (1 sample), and 17 (2 samples). No samples of the Boaxxe family were classified as noise. The fact that the Boaxxe family is partitioned in multiple clusters is not surprising; Figure 5.2b already revealed that the Boaxxe family as defined by human analysts at F-Secure Corporation contains several samples which structurally differ significantly from the other samples in the family.

The results from the DBSCAN algorithm on the malware samples are very promising. Except for three clusters, each cluster identifies a family correctly without mixing samples from multiple families. Furthermore, the majority of samples originating from larger families were classified inside a cluster and hence were not considered noise. Families which contain fewer than $Minpts$ samples are mostly classified as noise (e.g. Vundo, Blebloh, Startpage,etc), unless they are highly similar to samples from different families (e.g. Autorun). Finally, only the larger families Veslorn (8 samples) and Redosdru (9 samples) were fully discarded as noise. Closer inspection of these two families indeed showed that the samples within the families are highly dissimilar from a call graph point of view.

Finally, Figure 6.10 depicts a plot of the diameter and the cluster tightness, for each cluster in Figure 6.9. The diameter of a cluster is defined as the similarity of the most dissimilar pair of samples in the cluster, whereas the cluster tightness is the average similarity of a pair of samples. Most of the clusters are found to be very coherent. Only for clusters 2, 6, and 7, the diameter differs significantly from the average pairwise similarity. For clusters 2 and 6, this is caused by the presence of samples from 2 different families which are still within Rad distance from each other. Cluster 7 is the only exception where samples are fairly different and seem to be modified over multiple generations. Lastly, a special case is cluster 16, where the cluster diameter is 0. The call graphs in this cluster are isomorphic; one cannot distinguish between these samples based on their call graphs, even though they come from different families. Closer inspection of the samples in cluster 16 by F-Secure Corporation revealed that the respected samples are so-called 'droppers'. A dropper is an installer which contains a hidden malicious payload. Upon execution, the dropper installs the payload on the victim's system. The samples in cluster 16 appear to be copies of the same dropper, but each with a different malicious payload. Based on these findings, the call graph extraction has been adapted such that this type of dropper is recognized in the future. Instead of creating the call graph from the possible harmless installer code,

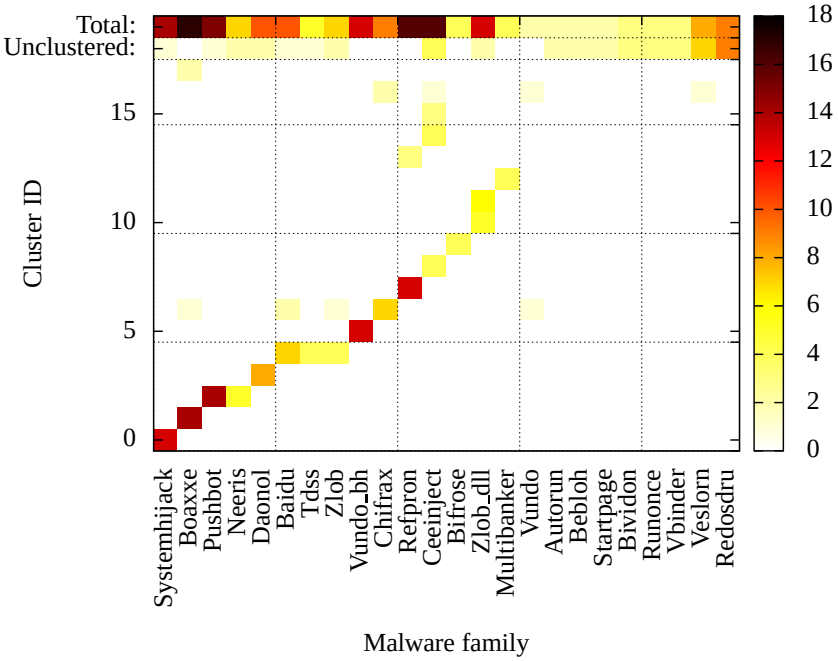


Figure 6.9: DBSCAN clustering with $Minpts = 3$, $Rad = 0.3$. The colors depict the frequency of occurrence of a malware sample from a certain family in a cluster.

the payload is extracted from the dropper first, after which a call graph is created from the extracted payload.

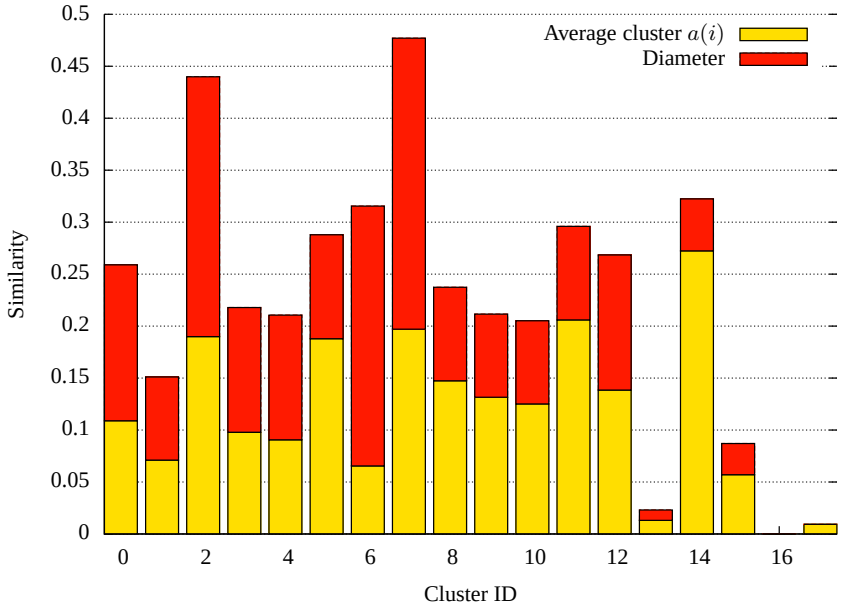


Figure 6.10: Plot of the diameter and tightness of the DBSCAN clustering.

Chapter 7

Conclusion

The main subject of this thesis has been the investigation of means to compare malware mutually via their call graph representations. Moreover, we explored automated identification and classification of malware families. New samples which are found to be very similar to known malicious code, are likely mutations of the same malicious code. Automated recognition of similarities as well as differences among these samples will ultimately aid and accelerate human analysis, rendering it no longer necessary to write detection patterns for each individual sample within a family. Instead, anti-virus engines can employ generic signatures targeting the mutual similarities among samples in a malware family.

After an introduction of call graphs in Chapter 2 and a brief description on the extraction of call graphs from malware samples, Chapter 3 discusses methods to compare call graphs mutually. Graph similarity is expressed via the Graph Edit Distance, which, based on our experiments in Chapter 6, seems to be a viable metric. Nevertheless, using this metric, it remains to be seen how well it is possible to distinguish malware from benign software. Especially benign software infected with malicious code could pose a serious challenge, because the malicious sample would possess a high similarity to the originally benign version of the software.

Another issue of concern regarding the GED is its expressiveness of malware similarity. Currently, the GED metric only considers elementary local operations such as vertex (edge) deletion and addition or vertex relabeling. It is however unclear how well these edit operations capture the reality. Possibly the virus writer could make simple high level modifications on the virus, which result in a large number of elementary edit operations from a call graph perspective. An overview of frequently applied malware mutations is given in [7]. We suggest to incorporate some of these mutations in the GED

metric.
Pairwise graph similarity is calculated by finding a graph matching which minimizes the GED. Unfortunately, exact graph matching algorithms are intractable for large graph instances. Consequently, Sections 3.4 and 3.5 elaborate on two approximation algorithms for finding accurate graph matchings. The first algorithm is based on the Hungarian algorithm, whereas the second one is a Genetic Search algorithm. To obtain accurate graph matchings, both algorithms require information about pairwise vertex (function) similarties for the graphs under comparison. This information is provided through cost functions as discussed in Chapter 4. As it turns out, the computationally least expensive cost function based on neighborhood comparisons and relabeling cost (Equation 4.2) as applied in [21, 50] provides the best results, i.e. matchings with the lowest Graph Edit Distance.
Currently, the cost functions in Chapter 4 only consider structural properties. Therefore, a natural extension of the cost functions is to compare the content of the functions semantically. The latter can for instance be accomplished by the approach proposed by Walenstein, Venable, et. al. in [46]. They suggest to describe each function as a vector of features. As an example of a feature, they give the frequencies of opcode n-grams. Next, the similarity of a pair of functions is estimated by taking the cosine similarity of their respected feature vectors. The use of opcode frequencies to characterize functions is also encouraged by the results of Bilar as he discovered that opcodes can be used to distinguish malware from benign software [4].
To facilitate the discovery of malware families, Chapter 6 applies several clustering algorithms on a set of malware call graphs. Verification of the classifications is performed against a set of 194 unique malware samples, manually categorized in 24 malware families by the data security company F-Secure Corporation. The clustering algorithms used in the experiments include various versions of the k-medoids clustering algorithm, as well as the DBSCAN algorithm. One of the issues encountered with k-medoids clustering is the specification of the desired number of clusters. Metrics to determine the optimal number of clusters did not yield conclusive results, and hence it followed that k-means clustering is not effective to discover malware families. Much better results on the other hand are obtained with the density-based clustering algorithm DBSCAN; using DBSCAN we were able to successfully identify malware families. At the date of writing, automated classification is also attempted on larger data sets consisting of a few thousand samples. F-Secure is currently analyzing the results, but since this is a time consuming process, the results could not be included in time in this thesis.
Future goals are to link the malware identification and family recognition software to the live stream of daily incoming samples. Observing the emergence

of new malware families, as well as automated implementation of protection against malware families, belong to the long term prospectives of malware detection through call graphs.

Bibliography

[1] "The Future Internet Programme," 2010, Visited on 01-2-2010. [Online]. Available: http://futureinternet.fi/programme.htm

[2] T. Anderson and D. Darling, "Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes," *Annals of Mathematical Statistics*, vol. 23, pp. 193–212, 1952.

[3] D. Arthur and S. Vassilvitskii, "k-means++: the advantages of careful seeding," in *SODA '07: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[4] D. Bilar, "Opcodes as predictor for malware," *International Journal of Electronic Security and Digital Forensics*, vol. 1, no. 2, pp. 156–168, 2007.

[5] S. Bradde, A. Braunstein, H. Mahmoudi, F. Tria, M. Weigt, and R. Zecchina, "Aligning graphs and finding substructures by message passing," May 2009, Visited on March 2010. [Online]. Available: http://arxiv.org/abs/0905.1893

[6] I. Briones and A. Gomez, "Graphs, entropy and grid computing: Automatic comparison of malware," in *Proceedings of the 2008 Virus Bulletin Conference*, 2008, Visited on May 2010. [Online]. Available: http://www.virusbtn.com/conference/vb2008

[7] D. Bruschi, L. Martignoni, and M. Monga, "Code normalization for self-mutating malware," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 46–54, 2007.

[8] E. Carrera and G. Erdélyi, "Digital genome mapping-advanced binary malware analysis," in *Virus Bulletin Conference*, 2004, Visited on May 2010. [Online]. Available: http://www.virusbtn.com/conference/vb2004

[9] S. Dasgupta, "The hardness of k-means clustering," Tech. Rep., 2008.

[10] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification (2nd Edition)*. Wiley-Interscience, November 2000, ch. 10, pp. 517–598.

[11] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," in *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, 2005, Visted on May 2010. [Online]. Available: http://actes.sstic.org/SSTIC05/Analyse_differentielle_de_binaires/

[12] M. Ester, H.-P. Kriegel, J. S, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of 2nd International Conference of Knowledge Discovery and Data Mining*. AAAI Press, 1996, pp. 226–231.

[13] H. Flake, "Structural comparison of executable objects," in *Proceedings of the IEEE Conference on Detection of Intrusions, Malware and Vulnerability Assessment (DIMVA)*, 2004, pp. 161–173.

[14] N. Funabiki and J. Kitamichi, "A two-stage discrete optimization method for largest common subgraph problems," *IEICE Transactions on Information and Systems*, vol. 82, no. 8, pp. 1145–1153, 19990825. [Online]. Available: http://ci.nii.ac.jp/naid/110003210164/en/

[15] X. Gao, B. Xiao, D. Tao, and X. Li, "Image categorization: Graph edit distance+edge direction histogram," *Pattern Recognition*, vol. 41, no. 10, pp. 3179 – 3191, 2008.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, January 1979.

[17] C. M. Grinstead and J. L. Snell, *Introduction to Probability*. American Mathematical Society, July 1997, ch. 11, pp. 405–470, Visited on April 2010. [Online]. Available: http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/amsbook.mac.pdf

[18] G. Hamerly and C. Elkan, "Learning the k in k-means," in *Advances in Neural Information Processing Systems*, vol. 17. MIT Press, 2003.

[19] Hex-rays, "The IDA Pro disassembler and debugger," http://www.hex-rays.com/idapro/, Visited on 12-2-2010.

[20] ——, "Fast library identification and recognition technology," http://www.hex-rays.com/idapro/flirt.htm, 2010, Visited on 12-2-2010.

[21] X. Hu, T. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs." in *ACM Conference on Computer and Communications Security*, E. Al-Shaer, S. Jha, and A. D. Keromytis, Eds. ACM, 2009, pp. 611–620.

[22] A. Justice, D. Hero, "A binary linear programming formulation of the graph edit distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, pp. 1200–1214, 2006. [Online]. Available: http://people.ee.duke.edu/~lcarin/JusticeHero.pdf

[23] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis (Wiley Series in Probability and Statistics)*. Wiley-Interscience, March 2005, pp. 68–125.

[24] A. Kirichenko, "Research Collaboration Manager F-Secure Ltd." personal communication, 2010.

[25] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. Mineola (NY): Dover Publications, 2001, ch. 5, pp. 201–207, originally published: New York : Holt, Rinehart, and Winston, c1976.

[26] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, 1st ed. Cambridge University Press, 2008, ch. 16.

[27] T. Micro, "The business of cybercrime - a complex business model," *A Trend Micro White Paper*, 2009, Visted on May 2010.

[28] Microsoft, "Microsoft portable executable and common object file format specification," 2008, Visted on 12-2-2010. [Online]. Available: http://www.microsoft.com/whdc/system/platform/firmware/PECOFFdwn.mspx

[29] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997, ch. 9, pp. 259–280.

[30] E. Moore, "A generalized inverse of matrices." in *Proceedings of the Cambridge Philosophical Society*, 1955, pp. 394–395.

[31] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[32] M. Neuhaus, K. Riesen, and H. Bunke, "Fast suboptimal algorithms for the computation of graph edit distance," in *Structural, Syntactic, and*

Statistical Pattern Recognition. LNCS, vol. 4109/2006. Springer, 2006, pp. 163–172.

[33] P. Orponen, "Professor Aalto University, Head of Dept. of Information and Computer Science," personal communication, 2010.

[34] H.-S. Park and C.-H. Jun, "A simple and fast algorithm for k-medoids clustering," *Expert Systems with Applications*, vol. 36, no. 2, Part 2, pp. 3336 – 3341, 2009.

[35] M. Pietrek, "An in-depth look into the win32 portable executable file format," 2002, Visited on 12-2-2010. [Online]. Available: http://msdn.microsoft.com/nl-nl/magazine/cc301805%28en-us%29.aspx

[36] J. W. Raymond and P. Willett, "Maximum common subgraph isomorphism algorithms for the matching of chemical structures," *Journal of Computer-Aided Molecular Design*, vol. 16, p. 2002, 2002.

[37] K. Riesen and H. Bunke, "Approximate graph edit distance computation by means of bipartite graph matching," *Image and Vision Computing*, vol. 27, no. 7, pp. 950 – 959, 2009, 7th IAPR-TC15 Workshop on Graph-based Representations (GbR 2007).

[38] K. Riesen, M. Neuhaus, and H. Bunke, "Bipartite graph matching for computing the edit distance of graphs," in *Graph-Based Representations in Pattern Recognition*, 2007, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72903-7_1

[39] P. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *J. Comput. Appl. Math.*, vol. 20, no. 1, pp. 53–65, 1987.

[40] B. Ryder, "Constructing the call graph of a program," *Software Engineering, IEEE Transactions on*, vol. SE-5, no. 3, pp. 216 – 226, may 1979.

[41] M. A. Stephens, "Edf statistics for goodness of fit and some comparisons," *Journal of the American Statistical Association*, vol. 69, no. 347, pp. 730–737, 1974. [Online]. Available: http://dx.doi.org/10.2307/2286009

[42] Symantec Corporation, "Symantec Global Internet Security Threat Report Volume - Trends for 2009 - Volume XV," April 2010. [Online]. Available: http://www.symantec.com

[43] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005, ch. 6.

[44] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Addison Wesley, May 2005, ch. 8, pp. 487–568.

[45] M. Wagener and J. Gasteiger, "The determination of maximum common substructures by a genetic algorithm: Application in synthesis design and for the structural analysis of biological activity," *Angewandte Chemie International Edition*, vol. 33, pp. 1189 – 1192, 1994.

[46] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia, "A.: Exploiting similarity between variants to defeat malware: vilo method for comparing and searching binary programs," in *Proceedings of BlackHat DC 2007*, 2007.

[47] Y. Wang and N. Ishii, "A genetic algorithm and its parallelization for graph matching with similarity measures," *Artificial Life and Robotics*, vol. 2, no. 2, pp. 68–73, 1998.

[48] N. Weskamp, E. Hullermeier, D. Kuhn, and G. Klebe, "Multiple graph alignment for the structural analysis of protein active sites," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 4, no. 2, pp. 310–320, 2007.

[49] D. B. West, *Introduction to Graph Theory (2nd Edition)*. Prentice Hall, August 2000.

[50] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance," *PVLDB*, vol. 2, no. 1, pp. 25–36, 2009.