Malware Classification based on Call Graph Clustering[*]

Joris Kinable
Orestis Kostakis
Aalto University, Department of Information and Computer Science
Helsinki Institute for Information Technology
P.O. Box 15400, FI-00076 Aalto, Finland
Joris.Kinable@tkk.fi, Orestis.Kostakis@tkk.fi

August 27, 2010

Abstract

Each day, anti-virus companies receive tens of thousands samples of potentially harmful executables. Many of the malicious samples are variations of previously encountered malware, created by their authors to evade pattern-based detection. Dealing with these large amounts of data requires robust, automatic detection approaches.

This paper studies malware classification based on *call graph clustering*. By representing malware samples as call graphs, it is possible to abstract certain variations away, and enable the detection of structural similarities between samples. The ability to cluster similar samples together will make more generic detection techniques possible, thereby targeting the commonalities of the samples within a cluster.

To compare call graphs mutually, we compute pairwise graph similarity scores via graph matchings which approximately minimize the *graph edit distance*. Next, to facilitate the discovery of similar malware samples, we employ several clustering algorithms, including *k-medoids* and *DBSCAN*. Clustering experiments are conducted on a collection of real malware samples, and the results are evaluated against manual classifications provided by human malware analysts.

Experiments show that it is indeed possible to accurately detect malware families via call graph clustering. We anticipate that in the future, call graphs can be used to analyse the emergence of new malware families, and ultimately to automate implementation of generic detection schemes.

KEYWORDS: Call Graph, Clustering, DBSCAN, Graph Edit Distance, Graph Matching, k-medoids Clustering, Vertex Matching

1 Introduction

Tens of thousands of potentially harmfull executables are submitted for analysis to data security companies on a daily basis. To deal with these vast amounts of samples in a timely manner, autonomous systems for detection, identification and categorization 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 [31].

For each sample a data security company receives, it has to be determined whether the sample is malicious or has been encountered before, possibly in a modified form. Analogous to the human immune system, the ability to recognize commonalities among malware which belong to the same malware family would allow anti-virus products to proactively detect both known samples, as well as future releases of the malware samples from the family. To facilitate the recognition of similar samples or commonalities among multiple samples which have been subject to change, 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 directed edges [30].

This paper deals with mutual comparisons of malware via their call graph representations, and the classification of structurally similar samples into malware families through the use of clustering algorithms. So far, only a limited amount of research has been devoted to malware classification and identification using graph representations. Flake [11] and later Dullien and Rolles [8] describe approaches to finding subgraph isomorphisms within control flow graphs, by mapping functions from one flow

[*]This research has been supported by TEKES – the Finnish Funding Agency for Technology and Innovation as part of its ICT SHOK Future Internet research programme, grant 40212/09.

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 [8] 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 [5]. 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 [3] continued the work started by Carrera and Erdélyi. 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 [17, 20]. Additionally, the authors of [17] describe techniques for efficient indexing of call graphs in hierarchical databases to support fast malware lookups and comparisons.

In this paper we explore the potentials of call graph based malware identification and classification. First call graphs are introduced in more detail as well as graph similarity metrics to compare malware via their call graph representations in Sections 2 and 3. At the basis of call graph comparisons lay graph matching algorithms. Exact graph matchings are expensive to compute, and hence we resort to approximation algorithms (Sections 3, 4). Finally, in Section 5, the graph similarity metrics are used for automated malware classification via clustering algorithms on a collection of real malware call graphs. A more extensive report on the work is available in [20].

2 Introduction to Call Graphs

A call graph models a binary executable as a directed graph whose vertices, representing the functions the executable is composed of, are interconnected through directed edges which symbolize function calls [30]. 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.

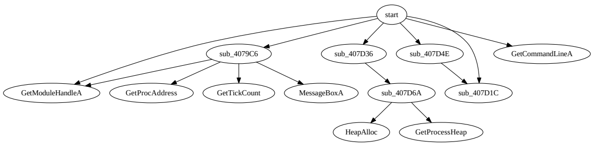


Figure 1: Example of a small malware call graph. Function names starting with 'sub' denote local functions, whereas the remaining functions are external functions.

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 [17], 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.*

Call graphs are generated from a binary executable through static analysis of the binary with disassembly tools [9]. First, obfuscation layers are removed, thereby unpacking and, if necessary, decrypting the executable. Next, a disassembler like IDA Pro [15] 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) [25, 23]. 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 [16] 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.

3 Graph Matching

3.1 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 derived from 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) $\phi : V(G) \to V(H)$ that optimizes a cost function which measures the quality of the mapping.*

In general, graph matching involves discovering structural similarities between graphs [27] 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 $(f(i), f(j)) \in E(H)$ [36]. 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 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 [37, 27, 17], 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 [14], from which the NP-hardness of GED optimization follows by the prevous argument (The latter result was first proven in [37] 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 [13] and for biochemical applications [26, 35]. The remainder of this Subsection serves as a brief literature review of different MCS and GED approximation approaches.

A two-stage discrete optimization approach for MCS is designed in [12]. 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 [12], the authors of [35] also rely on a two-stage optimization procedure, however contrary to [12], their algorithm tolerates errors in the MCS matching. A genetic algorithm approach to MCS is given in [34]. Finally, a distributed technique for MCS based on message passing is provided in [2].

A survey of three different approaches to perform GED calculations is conducted by Neuhaus, Riesen, et. al. in [27, 28, 24]. They first give an exact GED algorithm using A* search, but this algorithm is only suitable for small graph instances [24]. 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 [18], 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 [18], the authors of [37] developed new polynomial algorithms which find tighter upper and lower bounds for the GED problem.

3.2 Graph similarity

In general, malware consists of multiple components, some of which are new and others which are reused from other malware [9]. The virus writer will test his creations against several anti-virus products, 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 [4, 32].
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 $V(H)$ are supplemented with dummy vertices ϵ such that the resulting sets $V'(G)$, $V'(H)$ are both of size $|V(G) + V(H)|$ [37, 17]. 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 $\lambda_\phi(G, H)$:

$$\lambda_\phi(G, H) = VertexCost + EdgeCost + RelabelCost \tag{1}$$

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

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

$$\sigma(G, H) = \frac{\lambda_\phi(G, H)}{|V(G)| + |V(H)| + |E(G)| + |E(H)|} \tag{2}$$

3.3 Graph edit distance approximation

Finding a graph matching ϕ which minimizes the graph edit distance is proven to be an NP-Complete problem [37]. Indeed, empirical results show that finding such a matching is only feasible for low order graphs, due to the time complexity [24]. In [21, 20], the performance of several graph matching algorithms for call graphs is investigated. Based on the findings in [21], the fastest and most accurate results are obtained with an adapted version of Simulated Annealing; a local search algorithm which searches for a vertex mapping that minimizes the GED. This algorithm turns out to be both faster and more accurate than for example the algorithms based on Munkres' bipartite graph matching algorithm as applied in the related works [37, 17]. Two steps can be distinguished in the Simulated Annealing algorithm for call graph matching. In the first step, the algorithm determines which external functions a pair of call graphs have in common. These functions are mapped one-to-one. Next, the remaining functions are mapped based on the outcome of the Simulated Annealing algorithm, which attempts to map the remaining functions in such a way that the GED for the call graphs under consideration is minimized. For more details, refer to [21].

4 Clustering

Writing a malware detection signature for each individual malware sample encountered is a cumbersome and time consuming process. Hence, to combat malware effectively, it is desirable to identify groups of malware with strong structural similarities, allowing one to write generic signatures which capture the commonalities of all malware samples within a group. This Section investigates several approaches to detect malware families, i.e. groups of similar malware samples, via clustering algorithms.

4.1 *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 [1, 7]:

Definition 5. *(k-means Clustering): Given a data set χ, 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 Section 3. The latter algorithm is more commonly known as a *k-medoids* clustering algorithm [19], 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 [6], an approximation algorithm is used (Algorithm 1).

Algorithm 1: The k-medoids clustering algorithm

```
Input: Number of clusters $k$, set of call graphs $\chi$.
Output: A set of $k$ clusters C
foreach $C_i \in C$ do
    Initialize $\mu_{C_i}$ with an unused sample from $\chi$;
repeat
    Classify the remaining $|\chi| - k$ call graphs. Each sample $x \in \chi$ is put in the cluster which has the most similar cluster medoid;
    foreach $C_i \in C$ do
        Recompute $\mu_{C_i}$;
until The objective function converges;
return $C = C_0, C_1, ..., C_{k-1}$
```

In [22], a formal proof on the convergence of k-means clustering with respect to its objective function is given. To summarize, the authors of [22] prove 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 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 selects the medoids at random from χ. Arthur and Vassilvitskii observed in their work [1] 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 [1]. In their results, the authors of [1] 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 [1]. In summary, the authors describe an iterative approach to select the medoids, one after another, where the choice of a new medoid depends on the earlier selected medoids. For a detailed description of their k-means++ algorithm, refer to [1]. Finally, the last algorithm to select the initial medoids 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 by anti-virus analysts. We will refer to this initialization technique as "Trained initialization".

4.1.1 Clustering performance analysis

In this Subsection, we will test and investigate the performance of k-medoids clustering, in combination with the graph similarity scores obtained via the GED algorithm discussed in Section 3. The data set χ we use consists of 194 malware call graph samples which have been manually classified by analysts from 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 unique call graph samples.

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, and is analysed in depth in the next Subsection. For now, we assume that $k_{optimal} = 24$; the number of clusters obtained after manual classification. Note however that $k_{optimal}$ depends on the cluster criteria used to obtain a clustering. In Figure 2, 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. 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 comparing the different initilization methods of k-medoids clustering, based on Figure 2, one can indeed conclude that k-means++ yields better results than the randomly initialized k-medoids algorithm because k-means++ discovers tighter, more coherent clusters. Furthermore, the best results are obtained with Trained clustering where a member from each of the 24 predetermined malware families are chosen as the initial cluster medoids.

Figures 3a, 3b 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 3a comprises 86% Ceeinject samples, 7% Runonce samples and 7% Neeris samples. The family names are invented by data security companies and serve only as a means to distinguish families.

Figure 3a 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 3a). Only a few families, such as Baidu and Boaxxe, are scattered over multiple clusters. Figure 3b shows the clustering results of k-means++ [1]. 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, $k_{optimal}$. The latter issue is investigated in the next Subsection.

4.2 Determining the number of clusters

The k-medoids algorithm requires the number of clusters the algorithm should deliver as input. Two quality metrics are used to analyse the natural number of clusters, $k_{optimal}$, in the sample set: Sum of Error and the silhouette coefficient. For a more elaborate discussion, and additional metrics, refer to [20].

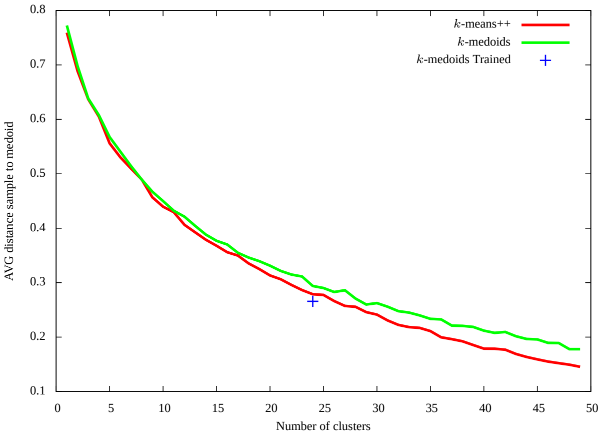


Figure 2: Quality of clusters. The average distance $\bar{d}(x_i, \mu_{C_i})$ between a sample x_i in cluster C_i and the cluster's medoid μ_{C_i} is averaged over 50 executions of the k-means algorithm.

4.2.1 Sum of (Squared) Error

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

$$SE_p = \sum_{i=1}^{k} \sum_{x \in C_i} (d(x_i, \mu_{C_i}))^p \qquad (3)$$

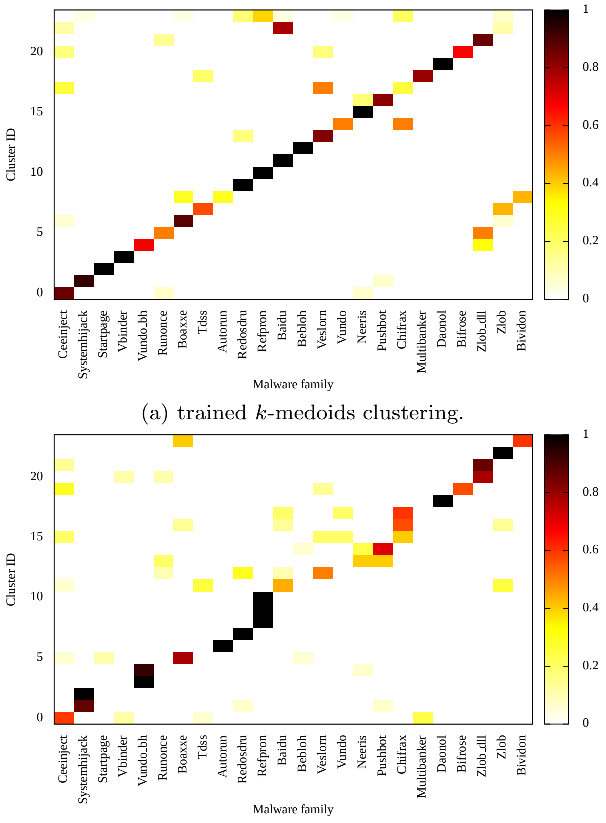
In this equation, $d(x, y)$ is a distance metric which measures the distance between a sample and its corresponding cluster centroid (medoid) as a positive real value. Here we choose $d(x_i, \mu_{C_i}) = 100 \times \sigma(x_i, \mu_{C_i})$.

Ideally, when one plots SE_p against an increasing number of clusters, one should observe a quick decreasing SE_p on the interval $[k = 1, k_{optimal}]$ and a slowly decreasing value on the interval $[k_{optimal}, k = |\chi|]$ [33].

4.2.2 Silhouette Coefficient

The average distance between a sample and its cluster medoid measures the cluster *cohesion* [33]. The cluster cohesion expresses how similar the objects inside a cluster are. The cluster *separation* on the other hand reflects how distinct the clusters are mutually. 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 notions of cohesion and separation can be combined into a single function which expresses the quality of a clustering: the *silhouette coefficient* [33, 29].

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



(a) trained k-medoids clustering.

(b) k-means++ clustering

Figure 3: Heat maps depicting non-unique clusterings of 194 samples in 24 clusters. The color of a square depicts the extent to which a certain family is present in a cluster.

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 $b^k(x_i)$:

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

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:

$$s(x_i) = \frac{b(x_i) - a(x_i)}{max(a(x_i), b(x_i))} \qquad (4)$$

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 [29].

The silhouette coefficient $s(x_i)$ in Equation 4 always yields a real value on the interval $[-1, 1]$. To measure the quality of a cluster, the average silhouette coefficient over the samples of the respective cluster is computed. An indication of the overall clustering quality is obtained by averaging the silhouette coefficient over all the samples in χ. To find $k_{optimal}$, one has to search for a clustering that yields the highest silhouette coefficient.

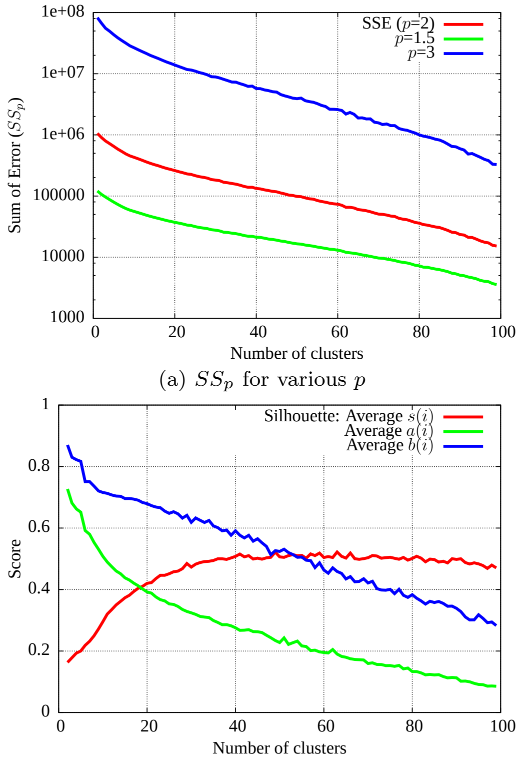
For a single sample x_i, $s(x_i)$ reflects how well the sample is classified. Typically, when $s(x_i)$ is close to 1, the sample has been classified well. On the other hand, when $s(x_i)$ is a negative value, then sample x_i has been classified into the wrong cluster. Finally, when $s(x_i)$ is close 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.

4.2.3 Experimental results

The SE_p and silhouette coefficients obtained after clustering the 194 malware samples for various numbers of clusters are depicted in Figure 4. Since the results of the clustering are subject to the randomness in k-medoids clustering, each clustering is repeated 10000 times, and the best obtained results are used in Figure 4. Interestingly, the SE_p curves for different values of p in Figure 4a do not show an evident value for $k_{optimal}$. Similarly, no clear peak in the silhouette plot (Figure 4b) is visible either, making it impossible to derive $k_{optimal}$. Consequently, using a k-means based algorithm, it is not possible to partition all samples in cohesive, well-separated clusters based on the graph similarity scores, such that the result corresponds with the manual partitioning of the samples by F-Secure. Furthermore, experimental results show that for some samples it is unclear to which cluster they should be assigned too, hence making it difficult to automatically reproduce the 24 clusters as proposed by F-Secure.

4.3 DBSCAN clustering

In the previous Section, 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, using a k-means based clustering algorithm. 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) SS_p for various p

(b) Silhouette Coefficient

Figure 4: Finding $k_{optimal}$ in the set with 194 preclassified malware samples.

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 the k-medoids algorithms failed to partition all 194 samples in well defined clusters, both Figure 3a and Figure 3b nevertheless reveal a strong correspondence between the clusters found by the k-medoids algorithm and the clusters as predefined by F-Secure. This observation motivates us to investigate partial clustering of the data which discards samples for which it is not clear to which cluster or family they belong. For this purpose, we apply the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) clustering algorithm [33, 10]. 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 areas can have an arbitrary shape; the samples do not necessarily need to be grouped around a single center.

To separate areas of low density from high density areas, DBSCAN utilizes two parameters: $MinPts$ and Rad. Using these parameters, DBSCAN is able to distinguishes between three types of sample points:

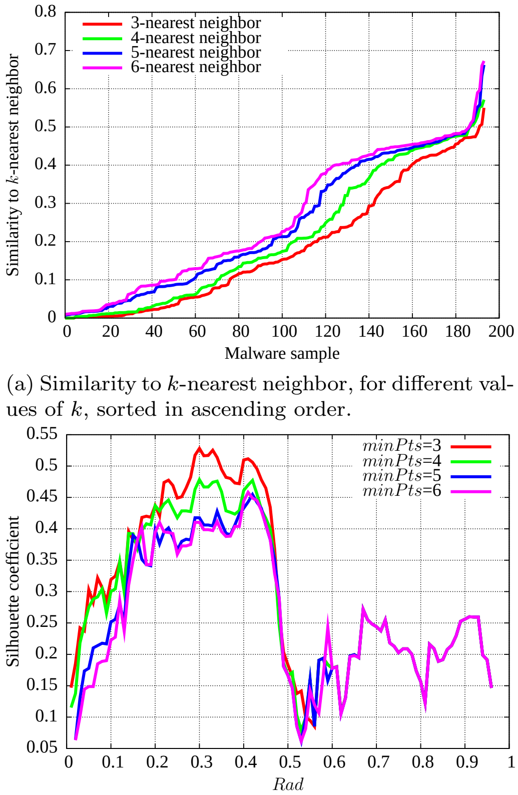
- Core points: $P_c = \{x \in \chi, |N_{Rad}(x)| > MinPts\}$, where
  $N_{Rad}(x) = \{y \in \chi, \sigma(x, y) \leq Rad\}$
- Border points: $P_b = \{x \in (\chi \backslash P_c), \exists y \in P_c : \sigma(x, y) \leq Rad\}$
- Noise points: $P_n = \chi \backslash (P_c \cup P_b)$

An informal description of the DBSCAN clustering algorithm is given in Algorithm 2.

Algorithm 2: DBSCAN clustering algorithm

```
Input: Set of call graphs $\chi$, $MinPts$, $Rad$
Output: Partial clustering of $\chi$
Classify $\chi$ in Core points, Border points and Noise;
Discard all samples classified as noise;
Connect all pairs $(x, y)$ of core points with $\sigma(x, y) \leq Rad$;
Each connected structure of core points forms a cluster;
For each border point identify the cluster containing the nearest core point, and add the border point to this cluster;
return Clustering
```

The question now arises how to select the parameters $MinPts$ and Rad. Based on experimental results, the authors of [10] find $MinPts = 4$ to be a good value in general. To determine a suitable value for Rad, the authors 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 procedure 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.



(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 5: Finding Rad and $MinPts$

Figure 5a shows the similarity of our malware samples to their k-nearest neighbors, 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 5a which value of $Minpts$ should be selected. To circumvent this issue, DBSCAN clustering has been performed for a large number of $Minpts$ and Rad combinations (Figure 5b). For each resulting partitioning, the quality of the clusters has been established with the silhouette coefficient. From Figure 5b one can observe that the best clustering is obtained for $Minpts = 3$ and $Rad = 0.3$. While comparing Figure 5b against Figure 5a, it is not evident why $Rad = 0.3$ is a good choice. We however believe that the silhouette coefficient is the more descriptive metric.

Finally, Figure 6 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 rows 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 observation that the Boaxxe family is partitioned in multiple clusters is analysed in more detail; closer analysis of this family revealed that there are several samples within the family with a call graph structure which differs significantly from the other samples in the family.

The results from the DBSCAN algorithm on the malware samples are very promising. Except 3 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 perspective.

Finally, Figure 7 depicts a plot of the diameter and the cluster tightness, for each cluster in Figure 6. 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 over all pairs 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 respective 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 possibly harmless installer code, the payload is extracted from the dropper first, after which a call graph is created from the extracted payload.

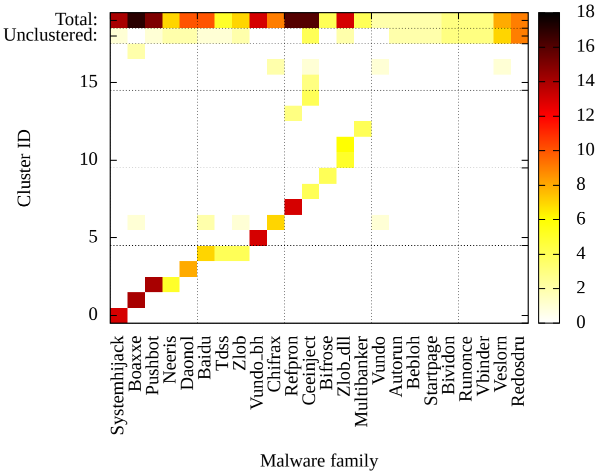


Figure 6: DBSCAN clustering with $Minpts = 3$, $Rad = 0.3$. The colors depict the frequency of occurrence of a malware sample from a certain family in a cluster.

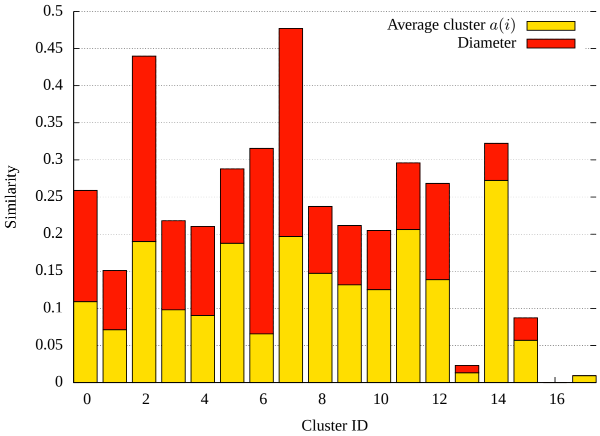


Figure 7: Plot of the diameter and tightness of the DBSCAN clustering.

5 Conclusion

In this paper, automated classification of malware into malware families has been studied. First, metrics to express the similarities among malware samples which are represented as call graphs have been introduced, after which the similarity scores are used to cluster structurally similar malware samples. Malware samples which are found to be very similar to known malicious code, are likely mutations of the same initial source. Automated recognition of similarities as well as differences among these samples will ultimately aid and accelerate the process of malware 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 Section 2 and a brief description on the extraction of call graphs from malware samples, Section 3 discusses methods to compare call graphs mutually. Graph similarity is expressed via the graph edit distance, which, based on our experiments, seems to be a viable metric. To facilitate the discovery of malware families, Section 4 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 anti-virus 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 main 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. However, manual analysis of the results is a time consuming process, and hence the results could not be included in time in this paper. Future goals are to link our malware identification and family recognition software to a live stream of incoming malware samples. Observing the emergence of new malware families, as well as automated implementation of protection schemes against malware families belong to the long term prospects of malware detection through call graphs.

Acknowledgements

The authors of this paper would like to acknowledge F-Secure Corporation for providing the data required to perform this research. Special thanks go to Pekka Orponen (Head of the ICS Department, Aalto University), Alexey Kirichenko (Research Collaboration Manager F-Secure), Gergely Erdelyi (Research Manager Anti-malware, F-Secure) for their valuable support and many usefull comments. 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).

References

[1] 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.

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

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

[4] 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.

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

[6] S. Dasgupta, "The hardness of k-means clustering," Tech. Rep. CS2008-0916, 2008.

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

[8] 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, Retrieved in May 2010. [Online]. Available: http://actes.sstic.org/SSTIC05/Analyse_differentielle_de_binaires/

[9] G. Erdélyi, "Senior Manager, Anti-malware Research F-Secure Corporation," personal communication, 2010.

[10] 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.

[11] 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.

[12] 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/

[13] 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.

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

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

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

[17] 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.

[18] 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

[19] 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.

[20] J. Kinable, "Malware Detection Through Call Graphs," Master's thesis, Department of Information and Computer Science, Aalto University, Finland, July 2010.

[21] O. Kostakis, J. Kinable, G. Erdélyi, and K. Mustonen, "Improved Call Graph Comparison Using Simulated Annealing," *to appear in Proceedings of the Network and IT Security Conference (NDSS 2011)*, February 2011.

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

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

[24] 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.

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

[26] 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.

[27] 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).

[28] 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

[29] 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.

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

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

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

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

[34] 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.

[35] 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.

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

[37] 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.