APPLICATION OF SUBSPACE CLUSTERING TO SCALABLE MALWARE CLUSTERING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY
BY

FATİH IŞIKTAŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
DEPARTMENT OF INFORMATION SYSTEMS

January 2019

Approval of the thesis:

APPLICATION OF SUBSPACE CLUSTERING TO SCALABLE MALWARE CLUSTERING

Submitted by **FATİH IŞIKTAŞ** in partial fulfillment of the requirements for the degree **of Master of Science in Information Systems Department, Middle East Technical University** by,

Prof. Dr. Deniz ZEYREK BOZŞAHİN
Dean, **Graduate School of Informatics** _____

Prof. Dr. Yasemin YARDIMCI ÇETİN
Head of Department, **Information Systems** _____

Assoc. Prof. Dr. Aysu BETİN CAN
Supervisor, **Information Systems Dept., METU** _____

Examining Committee Members:

Assoc. Prof. Dr. Altan KOÇYİĞİT
Information Systems Dept., METU _____

Assoc. Prof. Dr. Aysu BETİN CAN
Information Systems Dept., METU _____

Assoc. Prof. Dr. Cengiz ACARTÜRK
Cognitive Science Dept., METU _____

Assoc. Prof. Dr. Banu GÜNEL KILIÇ
Information Systems Dept., METU _____

Assoc. Prof. Dr. Sevil ŞEN
Computer Engineering Dept., Hacettepe University _____

Date: 14.01.2019

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Fatih IŞIKTAŞ

Signature : _____

ABSTRACT

APPLICATION OF SUBSPACE CLUSTERING

TO SCALABLE MALWARE CLUSTERING

Işıktaş, Fatih

MSc., Department of Information Systems

Supervisor: Assist. Prof. Dr. Aysu Betin Can

January 2019, 71 pages

In recent years, massive proliferation of malware variants has made it necessary to employ sophisticated clustering techniques in malware analysis. Choosing an appropriate clustering approach is very important especially for rapidly and accurately mining clustering information from a large malware set with high number of attributes. In this study, we propose a clustering method that is based on subspace clustering and graph matching techniques and presents an enhanced clustering ability and scalable runtime performance for the analysis of large malware sets. Unlike traditional signature-based clustering techniques, we aimed to obtain more accurate malware clusters by comparing internal structures of malware binaries. We also integrated a subspace clustering technique in order to scale and speed up the clustering process. To be able to verify our method, we developed a system prototype that can perform the mentioned clustering processes. This prototype provides a graphical user interface which allows users to navigate over malware binaries and generated clusters for a detailed analysis. We performed clustering experiments on real malware sets by using our system prototype. The experiment results showed that using a clustering method based on comparison of internal structure of malware binaries reveals clustering outputs with a 98% accuracy. Besides, the experiment results demonstrated that our method significantly improves the runtime performance of the clustering process without degrading clustering accuracy.

Keywords: Malware Clustering, Subspace Clustering, Graph Similarity

ÖZ

ALT UZAY GRUPLAMANIN ÖLÇEKLENEBİLİR KÖTÜCÜL YAZILIM GRUPLAMASINA UYGULANMASI

Işıktaş, Fatih
Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Doç. Dr. Aysu Betin Can

Ocak 2019, 71 sayfa

Son yıllarda, kötücül yazılım değişkenlerinin çok hızlı çoğalması, kötücül yazılım analizinde daha gelişmiş gruplama tekniklerinin kullanımını bir ihtiyaç haline getirmiştir. Özellikle, çok fazla niteliğe sahip olan büyük kötücül yazılım setlerinden gruplama bilgisini hızlı ve doğru bir şekilde elde edebilmek için uygun gruplama yaklaşımlarının tercih edilmesi çok önemlidir. Biz bu çalışmada, çok büyük kötücül yazılım setlerinin analizi için, altuzay gruplama ve grafik karşılaştırma tekniklerine dayanan ve gelişmiş gruplama yeteneği ve ölçeklenebilir çalışma süreleri sunan bir gruplama yöntemi öneriyoruz. Geleneksel imza tabanlı gruplama tekniklerinden farklı olarak, grafik karşılaştırma ile kötücül yazılımların iç yapılarını karşılaştırarak daha doğru kötücül yazılım grupları elde etmeye amaçladık. Bu gruplama işlemini hızlandırmak ve ölçekleyebilmek amacıyla da bir altuzay gruplama tekniğini yöntemimize entegre ettik. Yöntemimizi doğrulayabilmek için bahsettiğimiz gruplama işlemlerini gerçekleştirebilen bir prototip geliştirdik. Bu prototip, daha detaylı bir kötücül yazılım analizi için, kötücül yazılımlar ve üretilmiş gruplar üzerinde navigasyon imkanı sağlayan grafiksel bir kullanıcı arayüzü sunmaktadır. Geliştirdiğimiz prototipi kullanarak, gerçek kötücül yazılım setleri üzerinde gruplama deneyleri gerçekleştirdik. Deney sonuçları, kötücül yazılımların iç yapılarının karşılaştırılmasına dayanan bir gruplama yönteminin yüzde 98'lik bir doğruluk oranıyla gruplama çıktıları verdiğini gösterdi. Deney sonuçları ayrıca yöntemimizin, gruplama doğruluğunu bozmadan çalışma süresi performansını kayda değer bir şekilde geliştirdiğini gösterdi.

Anahtar Sözcükler: Kötücül Yazılım Gruplama, Altuzay Gruplama, Grafik Benzerliği

To My Family

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Assoc. Prof. Dr. Aysu Betin Can, for her guidance, encouragement and patience through this study. Her constructive feedbacks and invaluable suggestions enriched this work. Next, I would like to express my gratitude to Dr. Ali Arifoğlu and Dr. Atilla Özgit for their mentoring throughout my graduate studies. I would also like to thank all my professors at METU for teaching me valuable information.

I would like to express my love and gratitude to my beloved wife Süheyla for her understanding, support and endless love. I would also like to express my sincere gratitude to my family for their endless love and support through my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

DBSCAN	Density-based spatial clustering of applications with noise
GED	Graph Edit Distance
Ida	Interactive Disassembler
INSCY	Indexing Subspace Clusters with In-Process-Removal of Redundancy

1. INTRODUCTION

Digital products and services which provide internet-based rich contents have become indispensable tools for both personal and professional lives. Individuals and organizations are getting involved in information technologies domain in many ways thanks to the faster internet connectivity, developments in mobile technology and digital social platforms. The exponential increase in usage of information technologies in many areas has significantly increased the economic importance of services presented over internet on global marketplace. Comprehensive investments on digital services obviously attracts the attention of malicious parties to the market as well. Annual security reports published by various security companies present figures indicating the increasing impact of cyber threats on many different sectors.

There are several reasons that make these cyber threats successful. Firstly, product vendors and service providers are in a rush to create sophisticated products with new features to gain more competitive advantage. This race pushes producers to create more valuable products in a short number of periods that may results in deficient products which are unprotected to cyber-threats. Malicious parties target to security vulnerabilities of new products and to exploit them before the security gaps are unveiled and patched. Vendors then try to fix them by releasing updates and hotfixes. Even if the producers can take an immediate action to fix their products, it is not true for the end users to get the latest updates for their products in most cases. As a result, the success rate of cyber-attacks has been dramatically increased.

The tools used in cyber-attacks is another important success factor for cyber threats. Various sophisticated malicious programs are written by cyber attackers who have different motivations. One of the motivations is financial gain which can be done by gathering personally identifiable information from compromised computers or setting up a botnet to ransom corporations by threating them with ddos attacks (Hu, Chiueh, & Shin, 2009). Another one is data exfiltration, which can be motivated either by economic or intelligence reasons. In either case, there are large incentives for malware authors to continue to develop new threats (Anderson, 2014). Cyber-criminals use these tools to attack unprotected information systems. Evolving forms of cyber-attacks cause serious damage to information systems leading significant financial loss.

Combatting malicious software is a challenging task because malware variants are continuously emerging and evolving. Simple and signature-based detection techniques are not successful enough to identify malware variants. To deal with, a couple of analysis techniques are employed. One of these techniques is clustering malware instances based on their features and behaviors. Clustering aims to summarize objects in a dataset in a manner which ensures that similar objects are grouped together while dissimilar ones are separated (Assent, Krieger, Müller, & Seidl, 2007). Various clustering approaches have been successfully implemented in different disciplines such as bioinformatics, astronomy, physics, business management and marketing. Recently,

advances in data collection and management have led to large amounts of data being collected, giving rise to datasets with a high number of attributes (Sim, Gopalkrishnan, Zimek, & Cong, 2013). Traditional clustering algorithms are not capable of analyzing of datasets with a high number of features. As a result, new clustering approaches such as subspace clustering have been offered to address high dimensional data.

Clustering can be used for two cases in terms of malware analysis. First, given the increasing numbers and diversity of malware samples, clustering helps in creating malware classes which will be used for further malware analysis such as classification. Secondly, after a ground truth is created for the classification, clustering may also be used for emerging unknown samples. For example, Antivirus companies, IT security vendors or government agencies deal with numerous malware samples every day. They process the collected samples with classification methods to place them into right classes. However, they may need to create new classes for the new malware types or variants. In such cases, a clustering method can be utilized to create new classes for further classifications.

It is important to determine a suitable clustering method in order to cluster a large malware set. First, the number of clusters generated by the clustering method must be reasonable. Depending on the selected clustering algorithm, too many or few clusters can be generated that may not give useful and meaningful results to investigators. Clustering approach should have the capability of adjusting clustering space so that it allows analysts to process and identify malware samples in a structured manner. For this reason, scalability is a key feature which should be provided by the clustering algorithm. Besides, clustering method should also be applicable in terms of runtime. Clustering too many samples based on the features retrieved by static or dynamic analysis obviously takes too much time. To reduce runtime, firstly, feature selection method for the clustering should be chosen carefully. For example, dynamic malware analysis might give more accurate information about a sample for feature creation used in clustering, however it requires too much time for feature extraction since the sample is needed to be executed. On the other hand, static malware analysis might give faster results, however various obfuscation techniques can defeat static analysis. Secondly, clustering methods presenting faster mining approaches should be preferred for better clustering accuracy and runtime.

1.1 Research Goals

As mentioned earlier, analysis of large amounts of malicious software is a challenging task for information security professionals. Efficiently mining meaningful information about malware sets with high numbers of attributes requires using appropriate data mining methods such as clustering. Our main goal in this work is to develop a prototype system that will present an enhanced clustering ability and scalable runtime performance for the analysis of a large malware set. We aim to use a subspace

clustering method in malware clustering field in order to create a reference data for supervised learning techniques. This prototype will also have a graphical user interface which allows users to navigate over malware binaries and generated clusters for a detailed analysis. We propose that the subspace clustering based on internal function call graphs of malware binaries can be used to generate more accurate relational information to the security investigators and to improve the runtime of finding similar binaries relevant to an analysis within a large malware set.

1.2 Methodology

In this study, we designed and implemented a system by combining a group of methods and tools in order to improve clustering accuracy and runtime performance of malware clustering. We also developed a user interface which allows analysts to perform flexible and detailed malware clustering analysis. Our system uses call graph of local functions and dlls extracted from malware binary samples as the similarity metric in the clustering process. Hence, it provides a better clustering accuracy than signature-based clustering approaches.

Our system transforms malware binaries into graphs which are composed of nodes and edges. Each function of a malware binary is translated into a node while the caller-callee relationships between the functions compose edges. Functions of a binary code can be categorized as dynamically linked functions, statically-linked library functions and local functions. The function call graphs in this study are composed of these three types of functions.

Using internal structure of a malware as a comparison parameter provides more accurate clustering results; however, it significantly increases the clustering runtime. To overcome this runtime issue, we employ a subspace clustering method to improve runtime performance of the expensive graph matching algorithms. The subspace clustering generates clusters based on the static features extracted from malware binary codes such as local function count, local function call count, dll count and dll call count etc. Hence, the malware variants that have similar features can be grouped prior to the graph matching process.

We run the system over a set of malware binaries that we collected from two different web resources to observe and verify the system performance in terms of clustering accuracy and runtime. The experiment results demonstrate that our method improves runtime of the clustering process without degrading clustering accuracy. In other words, pre-clustering process, subspace clustering, makes the targeted binary set ready for the graph matching process with preserving true clustering information.

1.3 Thesis Outline

The outline of this thesis is as follows. In Chapter 2, we explain the terms and concepts of malware analysis and present related works carried out in this field. In Chapter 3, we explain the underlying tools and algorithms that we used in our system. In Chapter 4, we describe how we designed the prototype system and we explain the technical details of the system components. We also introduce the system's graphical user interface. In Chapter 5, we present and interpret the results of the experiments that we carried out for the validation of our proposal and the system implementation. In Chapter 6, we conclude and summarize our study and outline future directions.

2. BACKGROUND AND RELATED WORK

Malware analysis is a challenging task that requires a deep knowledge in various research fields, especially in computer science. Many approaches have been proposed to make this task easier by addressing different aspects of this problem. Malware clustering, for example, is one of these research topics that is used to reveal similarity of samples in a malware set and to group them based on their similarity. Similarly, malware classification aims to group malware samples into their families. All these efforts show that a remarkable progress has been made in this field. Before we present our approach, we introduce the terms and the concepts related to malware analysis in this chapter. We also summarize the previous studies performed in malware clustering.

2.1 What is Malware

Malware refers to malicious software written for infiltrating computer systems. There are many different forms of malware that are used as main tools in most cybercrimes and cyber wars. This fact has significantly increased the importance of development of known malware binaries and has triggered the search for new ways to create more sophisticated ones. For this reason, each of these types of malware has been continuously evolving with various motivations. Gaining illegal profit, competition between companies, national security concerns are among these drivers which attract many attackers who have different level of computer skills and knowledge.

When we look at the profiles of the malware writers, we could see this difference. On one side, many advanced and complex malware samples are written by individual software experts or teams. They are leading the others by adding new features and functions on current malware samples and by creating new malware generations. To be able to prolong of lifespan of their software, they even use professional software development techniques as seen in legitimate software design (Cesare, 2010). On the other side, people who have very limited computer knowledge are capable of using and modifying these malware samples. Although the complexity of malware has dramatically increased as a result of constant development, the barrier of using and creating malware has decreased in recent years. The main reason of this is that malware toolkits which provide easy to use platforms allowing users to automate malware creation and modification.

Thanks to these factors, malware writers can automatically and rapidly modify out-of-date malware, allowing them to gain advantageous against security authorities. They can create malware variants easily before new signatures are generated and distributed by antivirus vendors (Hu, 2011). Another facilitating factor affecting this barrier is that there are lots of free resources and tutorials explaining malware usage and

creation in detail on the internet. All these facilities result in a massive proliferation of malware.

2.2 Malware Analysis

Malware analysis refers to the techniques that help analysts to find out behaviors of malware instances and the risks that they may cause. Attributes extracted with malware analysis can be used to cluster unknown malwares into appropriate malware families (Gandotra, Bansal, & Sofat, 2014). Information gathered from well-structured malware analysis can also be used to learn tendency of malware development and to take measures to prevent future threats.

Most companies and individual users have different types of antivirus and internet security programs to protect their computer systems. Almost all current antivirus vendors employ signature or hash-based detection methods to identify and classify malicious programs. Signature-based identification approaches are popular because malware signatures can easily be created and distributed to the end users without bringing computational burden to their systems. However, various code obfuscation methods can easily bypass these schemes (Anderson, 2014). In other words, these approaches are not effective against new malware variants until new signatures are generated.

Malware authors try to develop different techniques to extend the life of their malware against currently updated antivirus detection systems. Polymorphic or metamorphic malware creation is one of those hiding techniques. Polymorphic malware changes its appearance by using encryption on each execution, but its main code structure does not change. Metamorphic malware, on the contrary, automatically changes its code by adding new instructions or changing registers every time it propagates. Malwares using these techniques can evade signature-based detection tools. Besides, some malware types can even sense dynamic analysis methods such as debugging and virtualization, and can hide its malicious execution paths. These obfuscation methods are making it hard to deal with constant threat of malwares.

In addition to the evading techniques, the exponential increase in malware variants has become a challenging factor for the security analysts. As the number of newly released malware variants and the complexity of the obfuscation techniques that are applied to the original malware increase, the process of detecting new variants and creating mitigation techniques become more difficult and time consuming. Hence, more effort is needed to develop new technologies that automate and facilitate the analysis and classification of the thousands of new malware variants that are released on a daily basis (Rad, Masrom, & Ibrahim, 2012). While work has been done to help optimize the analysis of known malware and questionable files, there is much to be done in the field of speeding up detection and automating these processes (Dowd, 2014). As a result, high-quality signatures can be generated rapidly with automatic analysis methods unlike

the time-consuming manual analysis. Such techniques help security analysts for taking fast actions in the examination of security threats.

Malware analysis is generally categorized as dynamic and static analysis. Dynamic analysis refers to the analysis performed during malware execution while static analysis is defined as the process of inspecting internals of a malware binary without execution (Egele, Scholte, Kirda, & Kruegel, 2012).

2.3 Dynamic Analysis

Dynamic analysis is done by running malware program and monitoring the program's behavior under different execution conditions (Awad, 2014). A sandbox environment is generally used to observe the behavior of a malware. Hence, the damaging risk of the malware against the host machine or other systems on the network can be eliminated. However, some malware types can detect the virtual environment on which they run and can change the execution path.

To gather various information about a system with dynamic analysis, a security analyst, in general, records the initial system state first, then executes the program to be analyzed and examines the system state during and after execution and makes note of all changes (Böhne, 2008). Dynamic analysis can reveal different kinds of information such as file paths, registry changes, IP addresses, memory writes and so on. Additionally, it can monitor network interfaces of the infected host machine to identify an unusual traffic flow. For example, a communication between a trojan file infected on a host and its command and control server could be spotted with this way. By tracing system and library calls on a system, the effects of a program on that system can be captured.

Dynamic analysis is strong against some obfuscation methods like binary packing and encryption or metamorphic and polymorphic malware creation, etc. However, it has some limitations. Dynamic analysis may not reveal all the execution paths since it observes the behavior of a program under some specific execution conditions, thus, complete behavior of the program may not be learned. Secondly, detection functions are used by malware creators in order to check for the presence of virtual environments. When such an environment is detected by the malware, the malware program behaves differently (non-maliciously) which leads to an incorrect analysis (Kang, Yin, Hanna, McCamant, & Song, 2009). Moreover, working on a high number of malware set may take too much time, which makes dynamic analysis not feasible for large-scale malware analysis.

2.4 Static Analysis

Unlike dynamic analysis, malware samples are not executed in static analysis. Binary code or source code of a malware instance form the basis of static analysis. But

mostly source code is not available for researchers to analyze; therefore, the binary form of a malware instance is usually used (Awad, 2014). Disassembler tools are used to examine internal functions, API calls and data segment of binary files. Static analysis can be used to gather a variety of information about a malware sample, e.g., high-level information such as its file size, a cryptographic hash, its file format, imported shared libraries, the compiler used to generate it, a list of human-readable strings that are contained in the file, or, low-level information gathered by disassembling or decompiling the sample (Böhne, 2008). All this information gives an idea about choosing proper disassembler and packer tools and classifying samples into their families.

Static analysis approaches have some advantages over dynamic analysis methods. Static analysis allows analysts to scan all code parts of a sample, hence, all execution paths can be discovered. Secondly, the analysis system is more protected because the sample is not executed. Static analysis approaches may extract so many features in a short amount of process time from a malware sample set with the help of automation methods. Moreover, the operating system on which the analysis runs does not have to be same with the target operating system of a malware sample. This provides flexibility in choosing analysis environment for malware analysts. As mentioned previously, the main disadvantage of static analysis is that it is hard for analysts to address obfuscation techniques such as self-modifying code and packed binaries.

2.5 Disassembly of Binaries

Disassembly refers to the process of analyzing an executable in order to obtain its assembly code in a text format. Disassembly tools are used to understand a program's binary structure when its source code is missing. Scanning and parsing instructions of a binary file is generally the first phase of static analysis. There are two main algorithms which are recursive-traversal and linear-sweep approaches in static disassembly.

In the linear-sweep algorithm, disassembly begins with the first byte in a code section and moves, in a linear fashion, through the section, disassembling one instruction after another until the end of the section is reached, and no effort is made to understand the program's control flow through recognition of nonlinear instructions such as branches (Eagle, 2011). Its main advantage is that it scans all code sections of a program, thus, it covers the whole binary code. However, meaningful data or junk bytes inserted into code sections is treated as instructions by the algorithm that might cause wrong interpretation of the binary code.

The Recursive Traversal algorithm employs control flow analysis to disassemble programs. Control flow analysis tries to find out what code pieces would be executed and in what sequence. This method is used to identify the possible execution paths through the binary code. Connections between basic blocks compose the control flow. A basic block is an instruction sequence that does not contain any branch instructions in

the body of the block except at the beginning and at the end of the block (Böhne, 2008). It means that if the block is called, it executes all its instructions from beginning to end. The algorithm follows the control flow in order to decode instructions of the binary. One of the principle advantages of the recursive traversal algorithm is its superior ability to distinguish code from data (Eagle, 2011).

The last method, speculative disassembly, runs recursive-traversal and linear-sweep respectively to find and decode all non-scanned code pieces in the binary.

2.6 Malware Clustering

The increasing number of new malware samples and variants requires further automated approaches and efficient data mining techniques. Data mining is the process of analyzing data based on different criteria and turning it into meaningful information. The overall goal of the data mining process is to extract information from a data set and transform it into an understandable structure for further use. Automated data mining methods have been increasingly employed for data analysis in many application domains as datasets have expanded in complexity and size. One of the primary data mining tasks is clustering which is intended to help a user discover and understand the natural structure or grouping in a dataset (Kailing, Kriegel, & Kröger, 2007).

Clustering aims at summarizing objects in a dataset in a manner which ensures grouping similar elements and distinguishing dissimilar ones (Assent, Krieger, Muller, & Seidl, 2007). Many clustering approaches have been implemented in different disciplines such as bioinformatics, astronomy, physics, business management and marketing. Recently, advances in data collection and management have led to large amounts of data being collected, giving rise to datasets with a high number of attributes (Sim, Gopalkrishnan, Zimek, & Cong, 2013).

One of the important application domains that requires improved data mining techniques is malware analysis. Malware analysis aims to identify behavior of a malware by performing a structured and scalable procedures such as classification and clustering. Classification is known as supervised learning technique in terms of data mining. Classification tries to determine the class of an object by retrieving information from already labeled or classified objects. Clustering, on the other hand, is an unsupervised learning method which aims to find and group similar objects in a set of objects without a need of predefined classes.

In malware analysis, malware clustering refers to group malware samples by using the features extracted by static or dynamic analysis methods. Considering the rapidly increasing malware types and variants, clustering helps in creating malware classes which will be used as reference classes for the classification process. In other words, clustering malware samples gives a starting point or ground truth for supervised learning techniques in malware analysis.

2.7 Related Studies

Many automation methods for clustering and classification of malware variants have been proposed to deal with the increasing number of malware threats. Some of them related to our study are summarized under this header.

In the study conducted in 2008, the authors proposed a classification approach that is based on static analysis using Ida tool (Tian, Batten, & Versteeg, 2008). They aimed to use the functions as the basis of a classification system for malware. They used two aspects of these functions: one is the length of the function as measured by the number of bytes of code in it; the other is the frequency with which function lengths occur within any sample of malware. They showed that these two features are significant together to classify malware variants. Secondly, function length information is unlikely to be an effective input for classifying some other types of malware, such as viruses, where the malicious code is difficult to extract.

A malware classification method based on string information of malware binaries was proposed in this article (Tian R. , Batten, Islam, & Versteeg, 2009). The first step is to unpack malware samples to deliver them to Ida tool for disassembling. Then the disassembly analysis is exported to a database for the next step which is classification. Information describing the printable strings contained in each sample becomes the input to various classification algorithms, including tree-based classifiers, a nearest neighbor algorithm, statistical algorithms and AdaBoost. The achieved classification results indicate a correct classification accuracy of 97% in the study.

Han, et al proposed an approach to analyze and classify malware instances (Han, Kang, & Im, 2011). They designed a binary comparison method based on instruction frequencies of malware code. They rely on the assumption that malware variants share similar instruction sequences. They asserted that malware structures are different from normal programs and this decrease the false positives in detection rates. Their results showed the method can be effectively used to distinguish malware from benign program, but not to effectively classify malware variants into malware families.

In his dissertation, Hu proposed a combination of four systems to deal with a large-scale malware analysis and clustering (Hu, 2011). The first system is called SMIT which is designed to check the similarity of malware instances based on the malware's function call graphs. SMIT uses static analysis to extract function call graphs from malwares. Secondly, the dissertation develops an automatic malware clustering system called MutantX. MutantX uses prototype-based standard agglomerative hierarchical clustering which allows malware analysts to focus on representative samples from each cluster and automatically generate labels for unknown samples based on their association with existing groups. Third, the author introduces a malware signature-generation system, called Hancock, that automatically creates string signatures of malwares. Finally, the dissertation proposes a system called DUET, that optimally integrates malware clusterings based on both static features and dynamic behaviors.

Zhong et al, aimed to improve runtime performance of malware analysis in their study (Zhong, Yamaki, & Takakura, 2012). In their proposal, they compare and classify malware instances based on their static function features. They first analyzed known malware families and created a signature database. After building the database, unknown malware samples are analyzed with the static analysis method to extract their features and compare them with the samples in the database. Although the result of the study does not reflect high classification accuracy, it improves the similarity calculation runtime.

Cesare et al, proposed a classification approach that is based on static analysis (Cesare, Xiang, & Zhou, 2013). Their method uses control flow graphs to identify malware instances. They first run their system on known malware families to build the malware control flow graph database. To classify new unknown malware samples, their control flow graphs are extracted and compared with the known graphs in the database. Edit distance method was used for the similarity comparison of control flow graphs. Their system showed a good accuracy but since the graph edit distance was used, the method would be so expensive in terms of runtime for large scale malware sets.

In his thesis study, Awad proposed that the structured control flow can be used as the invariant feature to automatically cluster malware variants (Awad, 2014). He used several tools and algorithms to automatically cluster malware samples. In his method, the malware instances of the sample were first analyzed by the FX tool to generate the structured control flow regex strings of their individual local functions. Then these generated regex strings are sent to the clustering tool. After generating the initial clusters, the local functions of the malware instances were mapped back to the malware instance they belong to, and the number of shared functions that have similar structured control flows was calculated between all pairs of malware instances in the sample set. The percentage of the shared functions determines clusters of the actual malware samples.

Arefkhani and Soryani introduce a pre-clustering method based on static analysis to categorize the huge number of malwares to an extremely smaller number of clusters in their paper (Arefkhani & Soryani, 2015). Their clustering method is based on image processing techniques. The idea is that the system converts the raw bytes of a Malware to a vector and then resize and reshape this vector to a two-dimensional vector that is considered as an image. Then image processing methods are applied to this image to extract textures. Visual similarities between malwares give an idea about their families. In the study, they used image processing Local Sensitive Hashing which does not need any comparison and hash values can be interpreted as cluster IDs. They tested their method with two data sets in order to show clustering accuracy and performance.

Singh and Khurmi focused on clustering large number of malware samples in a fast way in their article (Singh & Khurmi, 2016). According to the study, a malware file is a binary file composed of different byte values ranging from 0-255. Reading a binary file, byte by byte, the count of each byte value can be calculated, and it gives byte frequency of a malware file. These byte frequency vectors, uniquely identify each

malware sample like MD5 and CRC32 checksum, are then used for clustering. They asserted that their system has three contributions. Firstly, it uses the important parts of binary files, i.e., it only uses data from code section of a binary instead of dealing with the whole content. Secondly, the proposed method uses entropy as index which makes it fast and scalable since it compares only those cluster strings which have same index. Lastly, they evaluated the proposed system on real world malware samples and achieved 0.92 precision and 0.96 recall.

Chanderan and Abdullah proposed a method for clustering malware behaviors with discovering unknown variants of malware in an efficient manner in their paper (Abdullah & Chanderan, 2017). They used the hierarchical density-based algorithm (HDBSCAN) which has several capabilities such as automatic calculation of cluster count, ability to handle clusters of different density and shapes, ability to handle noise and outliers. They used dynamic analysis to generate reports from malwares. The idea is that the features and values are selected and abstracted from the reports. The system creates fix-length tokens which refer to as w-shingling, an overlapping word-based n-gram. The behavioral sequence pattern is designed in such a way that it will implicitly capture the program semantic. The report can then be embedded into a vector space. The Jaccard distance is used as distance metric to measure the similarities between reports, and to apply the metrics for clustering.

In his thesis study, Spizler investigated the applicability of the Lempel-Ziv Jaccard Distance (LZJD), a recently introduced similarity metric on arbitrary binaries, for hierarchical clustering (Spizler, 2018). He performed experiments with three separate datasets and analyze cluster quality from a hierarchical density-based clustering algorithm. He found that LZJD does not perform well with hierarchical clustering and does not result in well separated clusters. He proposed a new method called Partitioned Lempel-Ziv Jaccard Distance, but it underperforms LZJD, with decreasing accuracy and higher uncertainty as the number of partitions increase.

Similar to these studies, we designed and implemented a system by combining a set of methods in order to contribute to the solution for malware clustering accuracy and runtime problem. We use function call graphs of malware samples as a similarity metric which is seen in the studies (Hu, 2011) and (Cesare, Xiang, & Zhou, 2013). Unlike previous studies, we employ a subspace clustering method to improve runtime performance of the expensive graph matching algorithms. Secondly, we develop a user interface which allows analysts to perform flexible and detailed malware clustering analysis and to navigate through inputs and outputs of the system. Lastly, we performed clustering tests on real malware sets to observe and verify the accuracy and runtime performance of our system. The experiment results show that our method improves the runtime of clustering process without degrading clustering accuracy.

3. THE UNDERLYING TOOLS AND ALGORITHMS

In this chapter, we explain the tools and the algorithms that we used as the building blocks of our system. The INSCY, DBSCAN and GED algorithms and the IDA Tool are explained in detailed in this chapter. We designed a malware clustering system prototype composed of three functions which are binary feature extraction, subspace clustering and graph matching.

Our system uses the INSCY and DBSCAN algorithms together to identify and index supspace clusters in a malware set based on the static features obtained from each sample in the set. Secondly, the system runs a graph matching algorithm on the subspace clusters that are generated in the initial clustering process to create the final clustering. This second process uses the GED algorithm to measure the distance between malware pairs. The last piece is the Ida pro tool which is a disassembler used to extract the information from the malware set.

3.1 The INSCY Algorithm

Subspace clustering aims to find all clusters in all subspace projections. One of the challenges of mining all subspaces is that the number of subspace projections increases exponentially as the dimensionality of the space increases. This "curse of dimensionality" crucially affects the efficiency of finding subspace clusters. Finding all possible clusters within a high dimensional space is an unfeasibly expensive task. The second problem in subspace clustering is that many redundant clusters may be generated in the high dimensional object space. Clusters which appears in different subspaces are often redundant, and may contain essentially the same information as the maximal high dimensional one (Müller, Assent, & Seidl, 2009). Figure 1 illustrates this problem. In this 2-dimensional space, C1 and C2 are one dimensional cluster whereas the C3 is a 2-dimensional cluster. C3 contains the information that C1 and C2 provide. To increase the quality of the resulting clustering, excessive numbers of redundant clusters must be pruned. In addition, the higher dimensional projections should be given greater importance instead of less informative lower dimensional projections.

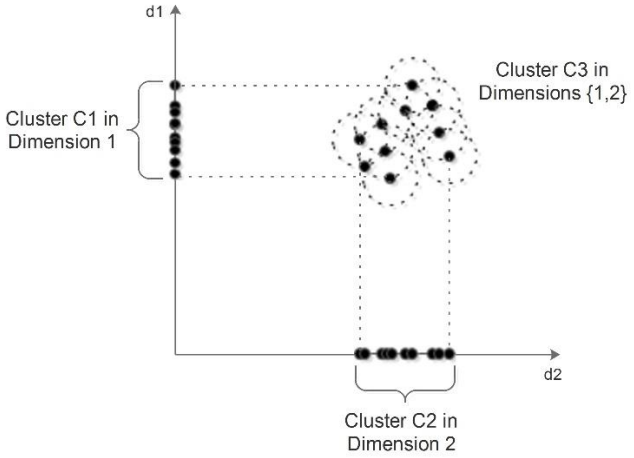


Figure 1: Projections in dimension 1 and 2

The INSCY (INdexing Subspace Clusters with in-process-removal of redundancY) algorithm uses a depth-first approach to address the previously described challenges. This strategy has two key advantages: First, because the maximal high dimensional projection is evaluated first, the algorithm is able to immediately prune all its redundant low dimensional projections and this leads to major efficiency gains (Assent et al, 2008). INSCY algorithm has an in-process redundancy pruning function which can prune lower redundant subspace projections immediately whenever higher dimensional subspace clusters are identified. Thus, costly density-based clustering computations are done only for the maximal high dimensional subspace projections. The breadth-first search algorithm, in contrast, starts mining from the lowest dimensional subspace projections. Since the higher dimensional subspace projections would be processed at the end of such an analysis, redundancy pruning could only be performed after all the subspace projections had been mined, but this would be too late in terms of computational time complexity. The breadth-first search approach would result in extremely large result sizes that would be filled with redundant clusters and that would have to be pruned and this would imply very high runtimes.

The second advantage of the INSCY algorithm is that potential subspace cluster regions can be indexed by using its index structure, namely the SCY-tree. The complete space can be turned into a SCY-tree data structure with only a single database scan. This provides an efficient way to perform clustering quickly. The SCY-Tree also supports top-down query of arbitrary subspaces without having to mine their lower dimensional projections. Indexing subspace clusters in a breadth-first manner, in contrast, would require building index structures for each of the exponentially many subspace combinations, which is clearly not feasible (Assent et al., 2008).

3.1.1.1 The SCY-Tree Structure

In the INSCY algorithm, all subspace regions are indexed by using the SCY-tree data structure, so that any subspace region can be queried without costly neighborhood computations. Figure 2 illustrates a SCY-tree which represents a 3-dimensional space. Each level represents a dimension and is divided into intervals to identify different segments of that dimension in the original space. In our example, there are three intervals in each of the three dimensions. Each sub region is described with at least one path which starts from the root node of the SCY-tree and ends at a leaf node. Leaf nodes of the paths store the count of objects which reside within the sub regions associated with the paths. Multiple paths can represent a sub region. In this case, the count of objects in the sub region is calculated by summing all leaf counts of associated paths.

Nodes are the building blocks of the SCY-tree, which is organized in a hierarchical manner. The main fields of each node in the SCY-tree are: a descriptor and a count value. A descriptor is composed of an integer dimension *d* and an interval number *i* within that dimension and expressed with the pair (d, i). The nodes of a SCY-tree are ordered based on their descriptors. When a specific subspace region is to be found for analysis, the related SCY-tree is restricted by using the corresponding descriptors. Once the restriction process is done, density of the subspace is evaluated according to the count of objects within that specific region and the geometric value of the region.

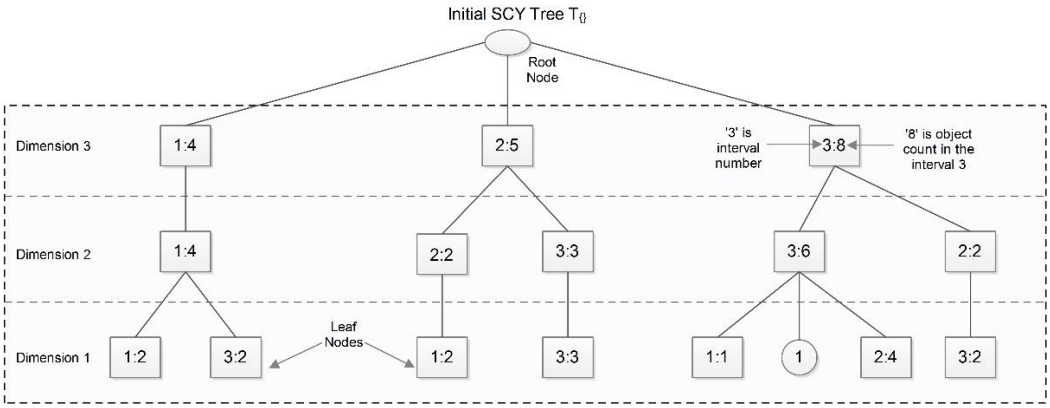


Figure 2: SCY-tree representing a three-dimensional space

Definition 1: SCY-Tree Structure

"*A SCY-Tree T_D represents a region $D = \{(d_1,i_1),...,(d_k,i_k)\}$ in an arbitrary subspace. The SCY-tree consists of nodes, each of which stores:*

- *a descriptor (d, i) for integer dimension d and interval number i of the region, and the count of c of objects within it*

- *a pointer to the parent node and a list of child pointers to child nodes*
- *a pointer of a linked list of nodes with the same descriptor* " (Assent I. , Krieger, Müller, & Seidl, 2008).

A two-dimensional projection of the three-dimensional space is seen in Figure 3-a. The colored sub region in the two-dimensional projection is represented with the colored paths in the SCY-Tree. When a sub region is to be located for cluster analysis, the SCY-tree is restricted by using the corresponding descriptors. In Figure 3-b, the restricted tree which represents all objects residing in segment 2 in dimension 1 has been depicted. The restricted tree consists of three paths, because in this region, all objects are in the interval numbers 1,2 and 3 of dimension two and three. The general idea is that one can restrict a region to a subset of its dimensions by restricting the corresponding SCY-tree representing this region (Assent et al., 2008).

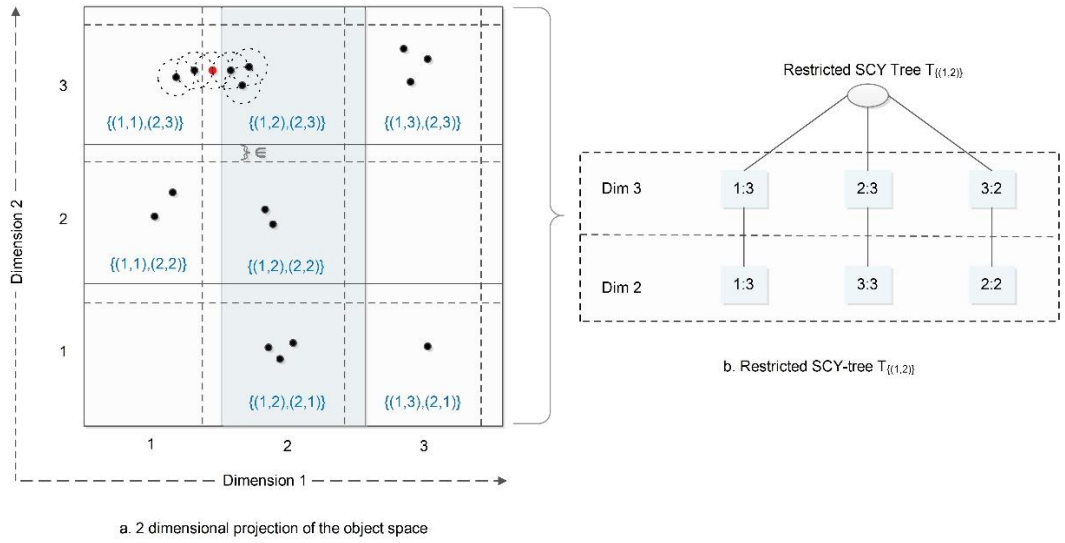


Figure 3: Two-dimensional projection

Between each of the cells in the Figure 3-a, a special ε width region is included called *S-connector* to ensure that density-based clusters which span across multiple regions can be found. S-connectors are set up at the upper border of each region. If there is at least one point inside of an S-connector, the neighbor regions are merged. For example, in the colored part of Figure 4-a, the red point which resides in the S-connector of the interval 3 in dimension 1. The two neighboring regions (interval 1 and interval 2 of dimension 1) are merged into a single region so that the density-based cluster spreading across the two regions can be found. S-connectors must act as sensors for detecting clusters that span across multiple regions.

While a space is being converted into a SCY-tree, S-connectors are represented as nodes that are missing a count value. In Figure 4-b, we can see a S-connector node, the red round node. If a path contains an S-connector node, this fact indicates that the two neighboring S-connected regions should be merged to obtain an aggregated cluster as shown in Figure 4-a. For multiple regions, this merging operation on neighboring SCY-trees can be done iteratively until no further object is contained in any surrounding ε region (Assent et al., 2008).

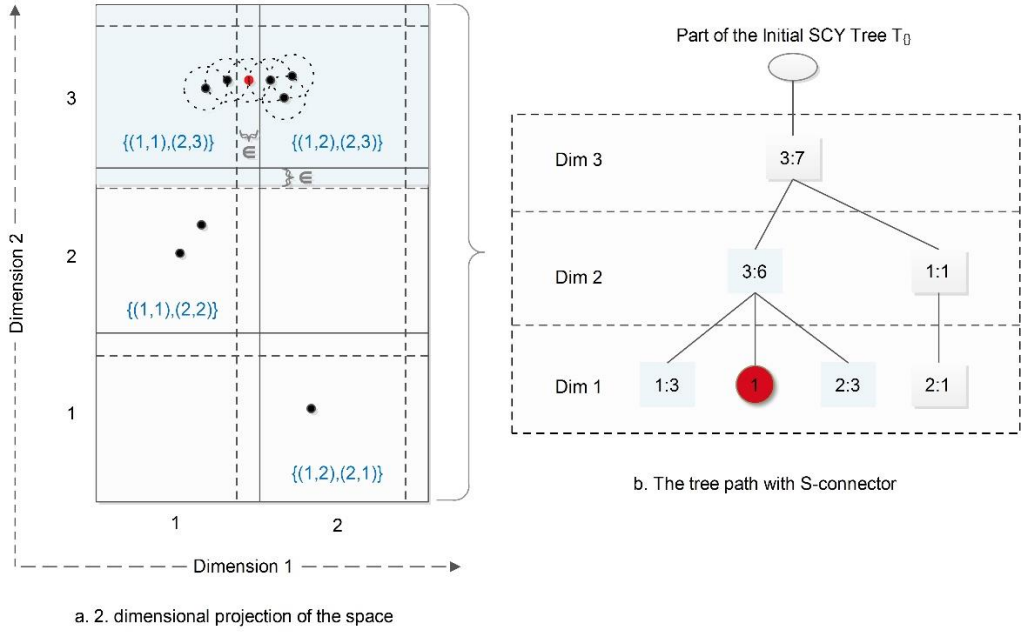


Figure 4: S-connected region in a two-dimensional projection

Using SCY-tree data structure when clustering objects has advantages over grid-based clustering algorithms. Grid based algorithms may lose clusters due to grid resolution and offsets. Clusters spreading over multiple grids or regions are clusters that spread across multiple regions are cut apart artificially in grid-based algorithms. This adversely affects the cluster quality. In the INSCY algorithm, on the other hand, all border regions that house points are indexed by S-connectors in the SCY-tree. This indexing allows the INSCY algorithm to find clusters without loss of accuracy. Essentially, the SCY-tree approach exhibits better efficiency than grid-based algorithms while avoiding the loss of quality (Müller et al., 2009).

3.1.1.2 The Algorithm

Once the SCY-tree has been constructed with a single database scan, the INSCY algorithm mines clusters by reading the SCY-tree to avoid additional expensive database

scans. By recursively restricting the subspaces in a depth-first fashion, maximal high dimensional subspace clusters can be quickly detected. Redundant clusters in the lower dimensional subspace projections can be pruned during the step back phase of recursive cluster mining. As seen in Algorithm 1, INSCY takes a SCY-tree and an empty list as initial inputs. It, then recursively calls restriction and pruning functions to mine the clusters.

Algorithm 1: INSCY (SCY-tree, result)

"*foreach descriptor in scy-tree do*
restricted-tree := restrict(scy-tree, descriptor);
restricted-tree := mergeWithNeighbors(restricted-tree);
pruneRecursion(restricted-tree); //prune sparse regions
INSCY(restricted-tree,result); //depth-first via recursion
pruneRedundancy(restricted-tree); //in-process-removal
result := DBClustering(restricted-tree) ∪ *result;* " (Assent I. , Krieger, Müller, & Seidl, 2008).

3.1.1.2.1 Restricting SCY-trees; Searching different subspaces

In the previous section, we defined descriptors (d,i) which are used to specify a region of a space. Within each descriptor, 'd' represents the integer dimension, and 'i' represents an interval number. Any level of descriptors can be selected to be used in the restriction of the SCY-tree. Restriction means that only a specific sub region is evaluated for possible clusters. The INSCY algorithm restricts the SCY-tree by considering all descriptors in order and detecting subspace clusters in all possible combinations of dimensions.

In Figure 5-a we see the SCY-tree representation of a three-dimensional space. To obtain a one-dimensional projection of the space in the interval 3 of dimension 1, we need to add up object counts within the nodes which lie in the region whose descriptor is (1,3). These nodes are shown in blue circles in Figure 5-a. The restriction of the SCY-tree results in the paths from the blue circled nodes to the root node which are copied into the restricted SCY-tree $T_{\{(1,3)\}}$ and labeled with the counts within those nodes. The result of this restriction process is the restricted SCY-tree shown in Figure 5-b. The paths that aren't related to the descriptor (1,3) are pruned.

This sort of restriction process can be performed recursively. When a second restriction is performed to the restricted SCY-tree $T_{\{(1,3)\}}$, a two dimensional projection is obtained, as seen in Figure 5-c. In this example, the SCY-tree is restricted by descriptor (2,3) to get a sub region. The resulting SCY-tree $T_{\{(1,3)\times(2,3)\}}$ has only one path and contains a single node. The output of the second restriction shows that there are 5 objects within interval number 2 of dimension 3.

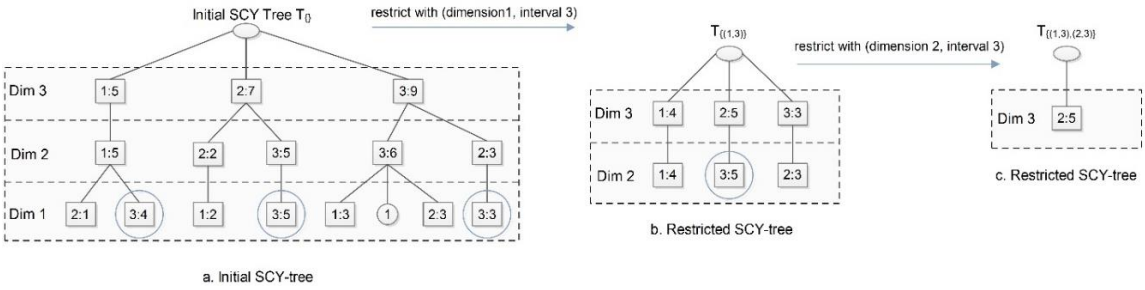


Figure 5: SCY-tree restriction process

We can also observe the restriction process on a three-dimensional space. Figure 6-a depicts an example of a three-dimensional space. Each dimension is equally divided into 5 intervals as opposed to 3 intervals in the previous example. The colored rectangular columns specify two-dimensional sub regions which are obtained by two successive restrictions. For example, the blue column is defined in terms of descriptors (1,5) and (2,5). The yellow column is defined by descriptors (1,4) and (2,3). When all points in the space are projected to the sub region that is the intersection of dimension 1 and 2, we obtain the two-dimensional projections shown in Figure 6-b. Then, density-based clustering analysis can be performed on these projections of subspace regions.

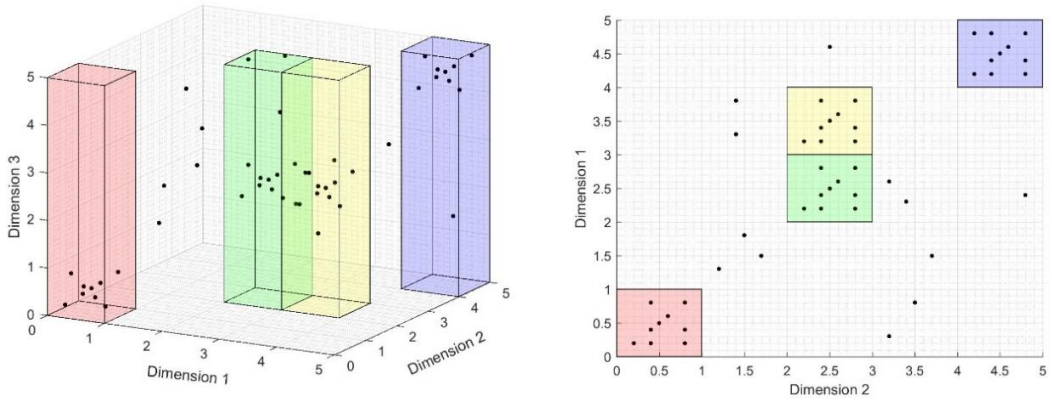


Figure 6: Two dimensional projections

If another second restriction is applied to the sub regions seen in Figure 6-a, maximal high dimensional subspaces are obtained for this space as shown in Figure 7. For example, the blue cube which represents a three dimensional projection that is in the 5th interval in all dimensions, and can be expressed with SCY-tree $T_{\{(1,5)\times(2,5)\times(3,5)\}}$. In other words, the blue region can be obtained by restricting the space using the descriptors (1,5), (2,5) and (3,5). If the threshold value for clustering is set to 5 for instance, then there will be four three dimensional regions which have enough points to

be classified as clusters. These four regions are colored red, yellow, green and blue in Figure 7.

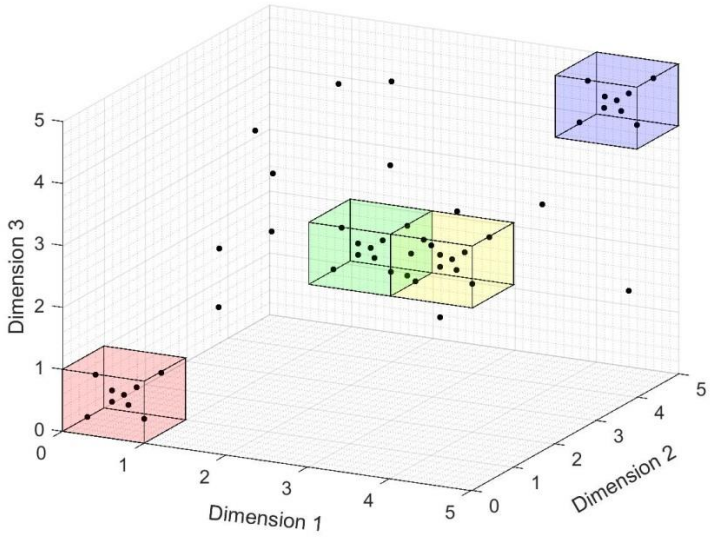


Figure 7: Three-dimensional projection

As recursive calls to INSCY consider successively high dimensional subspaces with recursive calls, the regions being considered become volumetrically smaller. Thus, if any projection does not have enough points in it, further restrictions could only result in low counts that would also not pass the threshold required to be classified as a cluster. Consequently, such spare regions can be safely pruned from the search tree when they are first discovered. The threshold value for a region to be classified as a cluster is denoted as *minPoints* parameter in the INSCY algorithm.

3.1.1.2.2 Merging SCY-trees; Growing S-connected regions

As we mentioned in the previous section, merging sub regions allows the algorithm to find clusters that span multiple sub regions. Figure 10 depicts how this merging is performed using the SCY-tree by the INSCY algorithm. Merging of S-connected restricted SCY-trees requires simply inserting all paths of one tree into the other and aggregating the count values which a lie along common paths, possibly inserting new nodes (Assent I. , Krieger, Müller, & Seidl, 2008). S-connected regions are coded as special paths in the SCY-tree. For example, in Figure 8, the path with blue node represents a S-connected region in dimension 2.

When the tree is restricted with descriptor (2,1), the restricted SCY-tree $T_{\{(1,3)\times(2,1)\}}$ is obtained as shown in Figure 8-b1. Similarly, the restricted SCY-tree $T_{\{(1,3)\times(2,2)\}}$ shown in Figure 8-b2 can be obtained by restricting the tree with descriptor

(2,2). The total count of objects in the SCY-trees are 5 and 6 respectively. The two SCY-trees are merged into the SCY-tree $T_{\{(1,3)\times(2,1-2)\}}$ that represents both intervals in dimension 2. This is depicted in Figure 8-c. Note that the merged tree has enough objects to induce further restrictions.

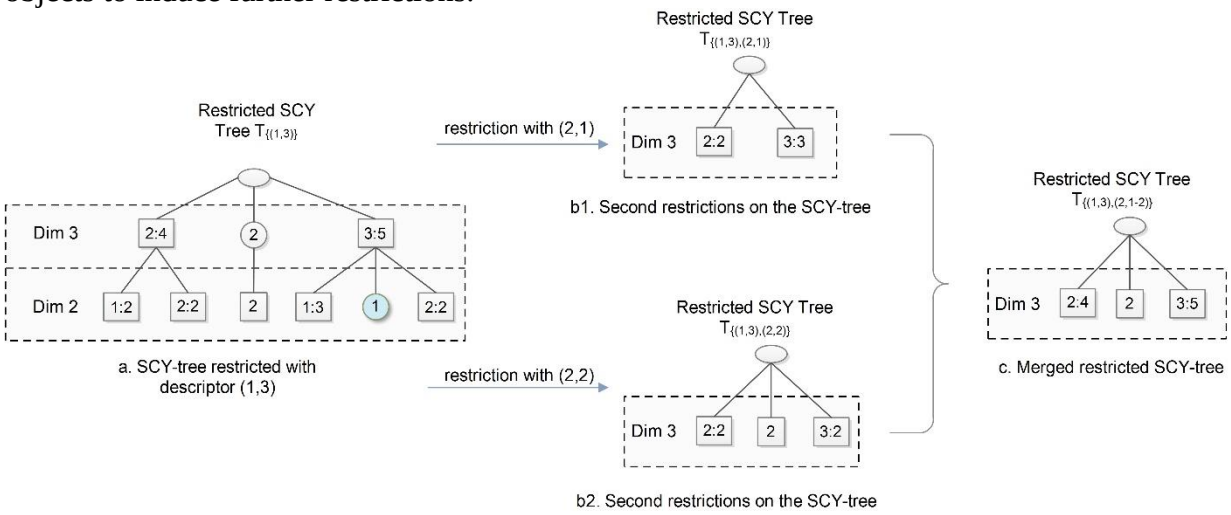


Figure 8: SCY-Tree merging process

We can also give an example of the merging using the three-dimensional S-connected neighboring regions in Figure 7. The yellow and green cubes are the maximal neighboring projections. Even if there is only one point at the border surface of these two cubes, the INSCY algorithm merges them into a single subspace so that it can find the true cluster which spans the two cubes. Let's assume the yellow sub region is not dense enough to have clusters in it. If the two sub regions were not merged, the system would have failed to detect the true cluster which spans both regions. Merging neighboring sub regions using S-connectors can provide better clustering quality.

3.1.1.2.3 Clustering; Mining actual subspace clusters

In this phase, the density-based clustering is carried out on the restricted SCY-tree. The actual data is accessed to identify neighborhoods and to check conditions necessary to determine that a cluster exists in this step (Assent I. , Krieger, Müller, & Seidl, 2008). For instance, if the SCY-tree in Figure 8-c is restricted, there is a merging of regions in dimension 3. The resulting SCY-tree is $T_{\{(1,3)\times(2,1-2)\times(3,2-3)\}}$ and has a count of 11. Since the region is a maximal subspace projection, no further restrictions can be applied to the SCY-tree. Therefore, we have reached the terminal restriction and density-based clustering can be performed on the region $(1,3)\times(2,1-2)\times(3,2-3)$.

3.1.1.2.4 Redundancy pruning; In-process removal

In-process-removal of redundant clusters is an important feature that makes the INSCY algorithm efficient in finding non-redundant maximal clusters. Due to the depth-first mining approach of the INSCY algorithm, maximal high dimensional clusters are added to the result set first. Over time, lower dimensional projections are analyzed for possible clusters. The algorithm decides whether to add lower dimensional clusters in the result set or discard them based on a redundancy parameter *R*. In Figure 10, one of the maximal high dimensional projections of the SCY-tree is $T_{\{(1,3)x(2,1\text{-}2)x(3,2\text{-}3)\}}$. This projection has 11 objects which exceed the value of the *minPoint* (the minPoint is set to 6 in this example). The minPoint parameter is used by DBSCAN algorithm and specifies the minimum object count necessary to form a cluster. Let's assume that a cluster has been found in this projection after the density-based clustering algorithm runs. If this were so, the cluster would be added to the result set as a maximal high dimensional cluster. In the next step, INSCY steps back and analyzes the lower dimensional projection $T_{\{(1,3)x(2,1\text{-}2)\}}$. This region has the same 11 objects in it, and for this reason will not be added to the result set because there is already a higher dimensional cluster that has been discovered, and accounts for the same set of objects.

3.1.1.2.5 Arbitrary restrictions; Detecting all subspace clusters

All possible combinations of dimensions of a clustering space are mined by the INSCY algorithm. As the dimension count of a clustering space increases, the total number of possible projections in that space increases exponentially. Because the main purpose of the INSCY algorithm is to find clusters in all possible projections, in-process-pruning has an important role. For example, in the previous example, a cluster found in a maximal high dimensional projection $T_{\{(1,3)x(2,1\text{-}2)x(3,2\text{-}3)\}}$ is added to the results set. When the algorithm steps back in the recursion, the lower dimensional projection $T_{\{(1,3)x(2,1\text{-}2)\}}$ which is described with just dimension 1 and 2 is processed. However, the lower dimensional cluster is unable to pass the redundancy check and is pruned. Similarly, all possible combinations of the lower dimensional projections are processed, and a redundancy check is performed for all lower dimensional projections. When a lower dimensional projection passes the redundancy check, the density-based clustering is performed for that projection. In this way, the result set contains clusters from the maximal high dimensional projections and non-redundant lower dimensional projections.

3.1.2 The DBSCAN Algorithm

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a clustering algorithm proposed by Ester et al (Ester, Kriegel, Sander, & Xu, 1996). By

using density distribution of objects in a database, DBSCAN can categorize these objects into separate clusters. DBSCAN takes only two parameters and finds arbitrarily shaped clusters as seen in Figure 9. Another advantage of the algorithm is that it uses the notion of 'noise' for the detecting of outlier objects. The algorithm is based on six definitions and two lemmas.

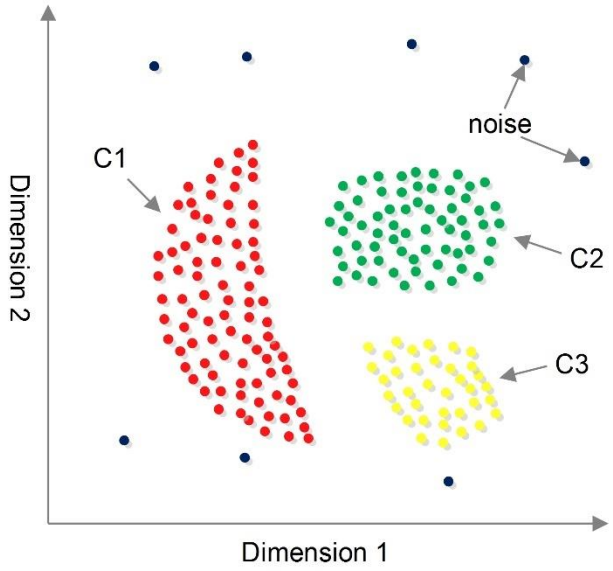


Figure 9: Arbitrary shaped clusters

3.1.2.1 Essential Definitions

The main idea of the algorithm is that for each point in a cluster, a neighborhood of given radius must contain at least a minimum number of points, that is, the density in a neighborhood must exceed some threshold. To calculate the distance between two points, any distance function can be used (e.g. Euclidian and Manhattan distance measures). The notion of the "Eps-neighborhood of a point" is described in definition 1.

"***Definition 1: (Eps-neighborhood of a point)***
The Eps-neighborhood of a point p, denoted by $N_{Eps}(p)$*, is defined by*
$N_{Eps}(p) = \{q \in D \mid dist(p,q) \leq Eps\}$" (Ester, Kriegel, Sander, & Xu, 1996).

Clusters contain two types of points which are the core points and the border points as shown in Figure 10-a. In general, core points have more neighboring points within their Eps-neighborhood than do the border points. The algorithm requires that for every point p in a cluster C, there is a point q in C so that p is in the Eps-neighborhood of q and $N_{Eps}(q)$ contains at least *MinPts* points.

"***Definition 2: (directly density-reachable)***
A point p is directly density-reachable from a point q with respect to (Eps, MinPts) if
1) $p \in N_{Eps}(q)$ and
2) $|N_{Eps}(q)| \geq MinPts$ (core point condition) " (Ester, Kriegel, Sander, & Xu, 1996).

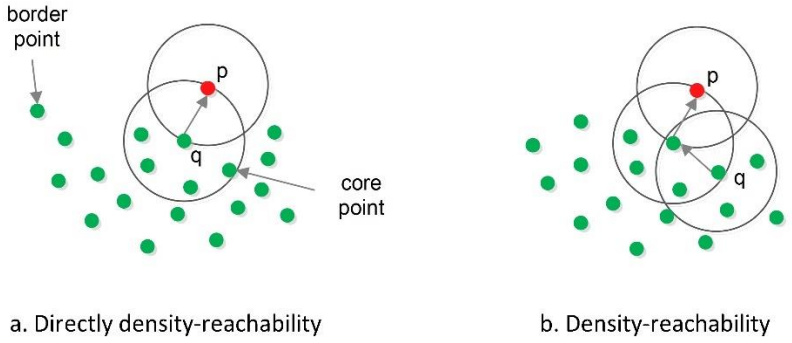


Figure 10: Density-reachable of points

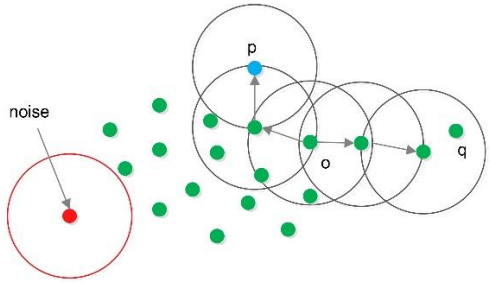
For the core point pairs, directly density-reachable is a symmetric relation whereas it the relation is not symmetric for the pairs composed of one core point and one border point. For instance, point p is directly density-reachable from point q as depicted in Figure 10-a, but the reverse is not true, because point p doesn't meet the second condition of definition 2. Density-reachable (which is defined with Definition 3) is basically an extension of the directly density-reachable.

"***Definition 3:*** *(density-reachable)*
A point p is density-reachable from a point q with respect to (Eps, MinPts) if there is a chain of points p_1, ..., p_n, $p_1 = q$, $p_n = p$ such that p_i+1 is directly density-reachable from p_i" (Ester, Kriegel, Sander, & Xu, 1996).

Figure 10-b depicts an example of density-reachability. Point p is density-reachable from point q, but not vice versa. Two border points of a cluster may not be density-reachable from each other. The density-connectivity notion is introduced in Definition 4 to express the relation of density-reachable between border points.

"***Definition 4: (density-connected)***
A point p is density-connected to a point q with respect to (Eps, MinPts) if there is a point 'o' such that both, p and q are density-reachable from o with respect to (Eps, MinPts) " (Ester, Kriegel, Sander, & Xu, 1996).

Density-connectivity is a symmetric relation. As seen in Figure 11, points p and q are density-connected to each other by point o.



c. Density-connectivity

Figure 11: Density-connectivity of points

A cluster can be defined based on the previous definitions. Condition 1 of definition 5 specifies that if a point p belongs to a cluster C and point q is density-reachable from point p with respect to a given (Eps, MinPts), then q also belongs to cluster C. The second condition expresses that each pair of points within a cluster are density-connected to each other.

"***Definition 5:*** *(cluster)*

Let D be a database of points. A cluster C with respect to (Eps, MinPts) is a non-empty subset of D satisfying the following conditions:

1) ∀ p, q: if p ∈ C and q is density-reachable from p with respect to (Eps, MinPts), then q ∈ C. (Maximality)

2) ∀ p, q ∈ C: p is density-connected to q with respect to (Eps, MinPts). (Connectivity) " (Ester, Kriegel, Sander, & Xu, 1996).

The red point in Figure 11 is an example of a noise which is the group of points in database D which did not placed to any cluster.

"***Definition 6:*** *(noise)*

Let C_1 ,. . ., C_k be the clusters of the database D with respect to parameters (Eps_i , $MinPts_i$), i = 1, . . ., k. Then we define the noise as the set of points in the database D not belonging to any cluster C_i , i.e. noise = {p ∈ D | ∀ i: p ∉ C_i}" (Ester, Kriegel, Sander, & Xu, 1996).

To validate the algorithm, the following two lemmas were presented by its developers. The first lemma specifies that an arbitrary point which meets the second condition of definition 2 with respect to a given (Eps, MinPts) should be found first. Then, the second lemma basically says that all points density-reachable from that seed can be retrieved.

"***Lemma 1:***

Let p be a point in D and $|N_{Eps}(p)| \geq MinPts$. Then the set O = {o | o ∈ D and o is density-reachable from p wrt. (Eps, MinPts) } is a cluster with respect to (Eps, MinPts)

Lemma 2:

Let C be a cluster wrt. (Eps, MinPts) and let p be any point in C with $|N_{Eps}(p)| \geq$ *MinPts. Then C equals to the set O = {o | o is density-reachable from p with respect to (Eps, MinPts) }*" (Ester, Kriegel, Sander, & Xu, 1996).

3.1.2.2 The Algorithm

The DBSCAN algorithm takes two arguments, namely *epsilon* and *minimum points*. It begins with choosing an arbitrary point from the dataset to be clustered. The *regionQuery* function returns the eps-neighborhood of the selected point. If the point has enough neighboring points within its Eps-neighborhood, it creates a new cluster. If not, the point is marked as noise. However, this point might be placed in a cluster later on, if it is density-reachable from some other point in the dataset.

The *expandCluster* function finds all density-reachable points from the seed point which was chosen in the first step. Once the density-connected cluster has been enumerated fully, the algorithm chooses another point which is unvisited before to find next possible cluster. These steps are repeated until all density-based clusters have been found in the database.

Algorithm 2: DBSCAN (D, eps, MinPts)

```
" C = 0
  for each unvisited point P in dataset D
    mark P as visited
    NeighborPts = regionQuery(P, Eps)
    if sizeof(NeighborPts) < MinPts
      mark P as NOISE
    else
      C = next cluster
      expandCluster(P, NeighborPts, C, eps, MinPts)

expandCluster(P, NeighborPts, C, eps, MinPts)
  add P to cluster C
  for each point P' in NeighborPts
    if P' is not visited
      mark P' as visited
      NeighborPts' = regionQuery(P', eps)
      if sizeof(NeighborPts') >= MinPts
        NeighborPts = NeighborPts joined with NeighborPts'
    if P' is not yet member of any cluster
      add P' to cluster C
```

regionQuery*(P, eps)*
return all points within P's eps-neighborhood (including P)" (Ester, Kriegel, Sander, & Xu, 1996).

The complexity of DBSCAN depends on the point count of a dataset and region queries performed for each point. This results in an average complexity of O(n * log n).

3.1.3 The Graph Matching

In our study, the last piece of the Malware Analysis System is a graph matching module which is responsible for computing the graph similarity of malware samples. The subspace clustering part (second module) generates the subspace clusters using specific features extracted from malware samples. The graph matching process runs on each of the resulting subspace cluster to obtain the actual clusters. In the graph matching phase, malware samples will be compared and clustered based on their function call graphs.

3.1.3.1 What is a Graph

A graph is a structure composed of vertices and edges which connects the vertices. Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, formed by pairs of vertices (Ruohonen, 2013). For example, vertices set can be shown as $V = \{v_1, \dots, v_5\}$, and edges set can be represented as $E = \{(v_4, v_3),(v_1, v_4),(v_2, v_4)\}$. If the elements in the edge set are ordered, then it gives a directed graph. If there is no order definition between vertices, then it gives an undirected graph as shown in Figure 12.

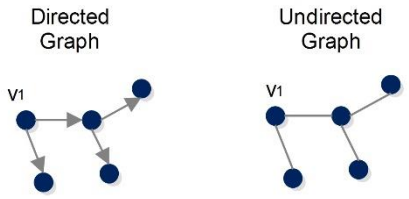


Figure 12: Directed and undirected graphs

3.1.3.2 Graph Matching Problem

Graph matching refers to the process of finding a structural similarity between two graphs. Graph matching can give accurate information about graph similarity; however, runtime complexity is a major problem seen in many graph matching methods. Various solutions have been proposed to obtain a linear runtime for the matching process. These methods can be classified under exact and inexact graph matching headers.

Exact matching between graphs is characterized by the fact that the mapping between the vertices of the two graphs must be edge-preserving, that is, if two vertices in the first graph are adjacent, they are mapped to two vertices in the second graph that are adjacent as well (Livi & Rizzi, 2013). Graph isomorphism, which is an exact matching method, seeks to find a structure and semantic similarity between two graphs.

Exact graph matching may not be applied, if two graphs have different number of vertices. It means that no isomorphism can be expected between both graphs, and the graph matching problem does not consist in searching for the exact way of matching vertices of a graph with vertices of the other, but in finding the best matching between them (Bengoetxea, 2002). Besides, exact matching problem is typically NP-complete. At this point, inexact graph matching methods are proposed to solve this problem.

Two graphs may have a very similar structure except some missing nodes or edges. In this case, a different method can be used to find mappings between graphs instead of using exact matching. The most adopted solution is to make the matching process tolerant in respect to deformations by introducing the concept of matching cost to penalize structural differences (Carletti, 2016). As the structures of two graphs are getting dissimilar, the cost of matching increases. This matching cost can be used later as a similarity parameter for grouping graphs.

3.1.3.3 Graph Edit Distance

Graph edit distance is a flexible graph dissimilarity measure that belongs to the family of inexact graph matching methods. The graph edit distance (GED) between two graphs can be defined as the minimum cost required to transform one of the given graphs into the other (Yan, et al., 2016). In particular, it measures the deformation between two graphs by considering the cost assigned to the sequence of elementary graph edit operations needed to transform the first graph in the second one (Carletti, 2016). A graph operation such as inserting or removing a node is the basic transformation unit performed on a graph, as defined in Definition 1. A transformation is composed of a set of edit operations sequentially applied to a graph, namely an edit path P. The cost of the edit path is defined as the sum of all its elementary operation's costs. An edit path from graph1 to graph2 with minimal cost is called an optimal path.

Definition 1. Graph edit operations:
The elementary edit operations generally include the following operations.

- Node/Edge insertion: Introducing a new node/edge to a graph.
- Node/Edge substitution: Changing a label of a given node/edge in a graph.
- Node/Edge deletion: Removing a node/edge from a graph

Figure 13 depicts an example of graph edit operations. Three edit operations (node addition, edge addition and edge removal) are performed in order to transform the graph 1 into the graph 2.

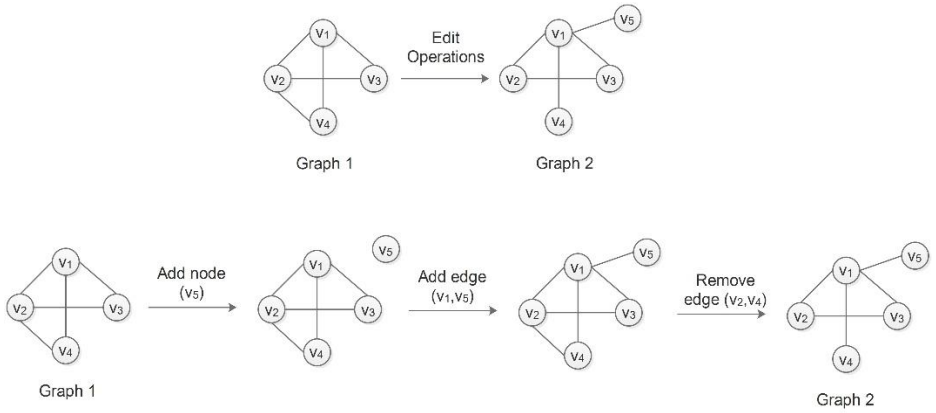


Figure 13: Graph transformation by edit operations

Graph edit distance (GED) is the most employed inexact graph matching method to compute the pairwise similarity of graphs. However, computing the GED is known NP-complete and a classic method is by means of a tree search procedure that basically evaluates all possible node-to-node correspondences (Sanfeliu & Fu, 1983). Even though, the computation complexity of GED is unsuitable for large graphs, several methods have been offered to make the computation of graph edit distance feasible. One is called bipartite GED solved by linear assignment. Bipartite GED and its variants approximate the GED problem by a linear assignment problem, which can be solved efficiently via e.g. the Hungarian method (Munkres, 1957). These methods approximate graph structure by a node-to-node cost matrix that encodes local clique structure (Yan, et al., 2016).

3.1.4 The Dissassembly Tool (IDA Pro)

In a traditional software development model, compilers, assemblers, and linkers are used to create executable programs. To analyze the internal code structure of programs, various tools are used to undo the assembly and compilation processes. The

purpose of disassembly tools is often to facilitate understanding of internal structures of programs when source code is unavailable.

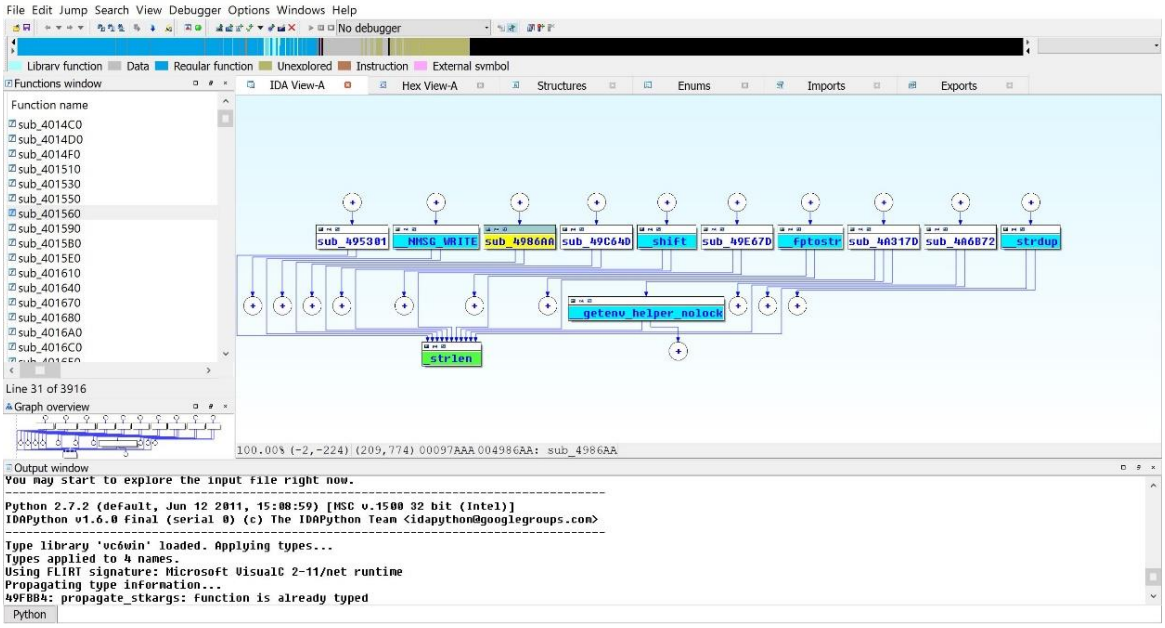


Figure 14: Ida pro tool GUI

We used the Ida pro tool (Hex-rays, 2018) to disassemble malware executables and extract the features necessary for the clustering phase. Since this study aims to deal with large number of malwares, our system needs to automate the disassembly process of malware samples. For this purpose, we use a plugin called IDAPython which allows scripts to run in Ida pro. In addition, Ida pro uses FLIRT signatures to quickly identify local and statically-linked library functions in executables. FLIRT signatures help Ida pro automatically rename functions extracted from malware binaries. A screenshot of the tool's main page is seen in Figure 14.

4. SYSTEM DESIGN AND IMPLEMENTATION

We designed and implemented a malware clustering system which generates clusters based on the internal structure similarity of malware binary codes. Using internal structure of a malware as a comparison parameter provides more accurate clustering results, however, it significantly increases the clustering runtime. To overcome this issue, feature extraction and clustering methods should be carefully chosen and implemented. In this study, we aimed to cover both accuracy and runtime aspects of malware clustering. In this chapter, we present the design of our system, and explain the functions running in the background. We also introduce the graphical user interface of the system.

4.1 Design of The System

We designed a malware clustering system prototype composed of three functions which are binary feature extraction, subspace clustering and graph matching. These functions are implemented based on the algorithms and tools mentioned in Chapter 3. The prototype system has a graphical user interface where an analyst can manage malware clustering process. Figure 15 depicts the system modules and malware clustering process flow through them.

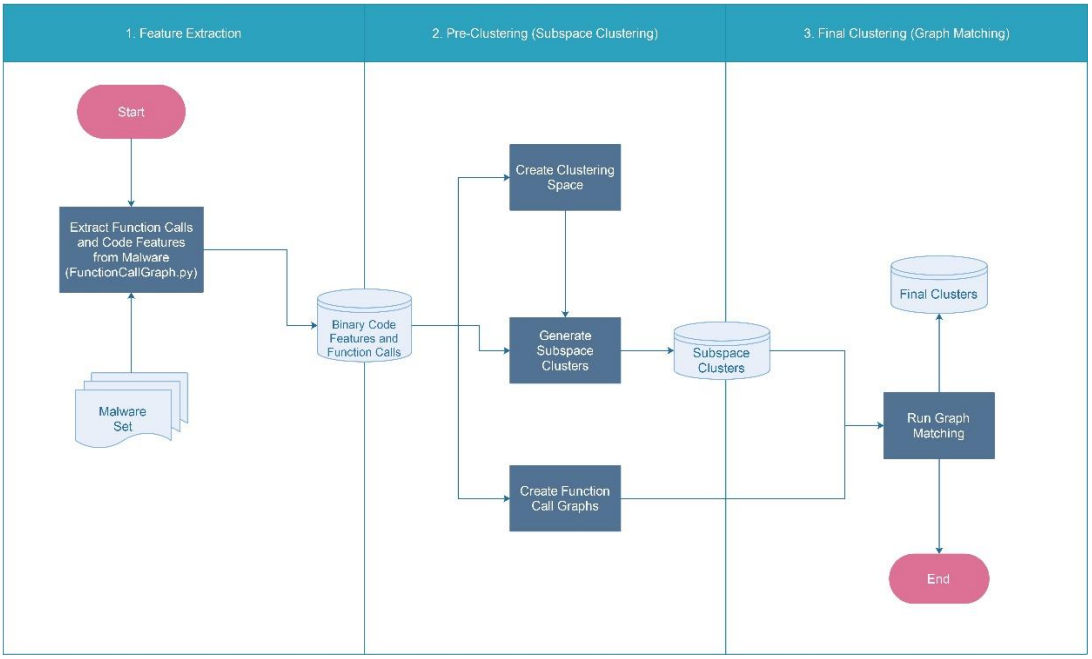


Figure 15: Malware clustering modules

In our study, each malware binary is transformed into a graph which is composed of nodes and edges. Each function of a malware binary is translated into a node while the caller-callee relationships between the functions are the edges. Since the relations between the binary functions have directions, the function call graphs are created as directed graphs in our implementation. Functions of a binary code can be categorized as dynamically linked library functions, statically-linked functions and local functions. Dynamically-linked functions are defined as DLL functions that are linked at runtime while statically-linked functions refer to the library functions that are statically linked into the binary code. Lastly, the local functions are specific functions written by programmers that gives an idea about binaries written with a similar intention. The nodes forming a graph for a malware are composed of these three types of functions.

To create function call graphs, we first need to extract all functions and the relations among them from a binary code by using a suitable disassembler. The disassembler that we use for this study must have some capabilities. First, it should allow us to call it programmatically since we intend to analyze a large malware database. Secondly, we should be able to create and use signatures to identify the functions extracted from binaries. Since we compare the graphs in the clustering process, we also need to determine a naming standard for all type of functions.

To meet these needs, we choose Ida pro (Hex-rays, 2018) disassembler tool which can classify, extract and label the three types of binary functions. There is an Ida pro plugin called IDAPython. Since this plugin allows scripts to run in Ida pro, we use it to automate our malware analysis. Moreover, Ida pro uses Fast Library Identification and Recognition Technology (FLIRT) signatures which provide rapid identification of local and statically linked functions in programs. FLIRT signatures help Ida pro automatically rename functions for the reverse engineering. These features of the Ida tool allow our system to automatize the function extraction process. For this purpose, we wrote a python script which calls the disassembler tool to extract necessary information from malware binaries. Hence, the first module of the system, feature extraction module, can read binary files from a folder, check and unpack packed binaries, extract function features and function calls, and write all the extracted information into a database.

The second module of the system is a pre-clustering tool written to improve the runtime performance of the final grouping which contains expensive graph matching operations. With an efficient indexing method, we can group the malware variants that have similar features before the graph matching process is started. For this reason, a subspace clustering algorithm is employed in this study which can analyze data of high dimension and categorize it based on object relationships.

In data with many attributes, clusters are often hidden in subspaces of the attributes and do not show up across a full attribute space (Assent, Krieger, Müller, & Seidl, 2007). Subspace clustering aims to automatically identify subspaces of the object space where clusters exist by mining all possible attribute combinations in a scalable way. In other words, main objective is to find high quality clusters in different subsets of a dataset in an efficient way. However, clustering a large binary set may generate

excessively large number of clusters that significantly increases the runtime of the analysis. Another issue is that many redundant clusters might be generated in a high dimensional space. To overcome these drawbacks, we use the INSCY (INdexing Subspace Clusters with in-process-removal of redundancY) algorithm, proposed in the article (Assent I. , Krieger, Müller, & Seidl, 2008), that mines subspaces in a depth-first search manner and employs a density-based clustering approach for the actual clustering.

INSCY algorithm has two key advantages: First, it uses depth-first searching with a pruning mechanism for the task of mining sub-regions. In this manner, as the maximal high dimensional projection is evaluated first, immediate pruning of all its redundant low dimensional projections leads to major efficiency gains (Assent I. , Krieger, Müller, & Seidl, 2008). Secondly, potential subspace cluster regions can be indexed by using the algorithm's index structure which supports access to arbitrary subspaces without mining their lower dimensional projections. Thus, costly density-based clustering computations are performed only for maximal high dimensional subspace projections.

Our system extracts several binary code features such as file size, stack size, local function and dll counts, basic block counts and function call counts from binary files. We define these binary code features as dimensions in our approach so that we can create a malware binary code space on which the system can run the subspace clustering process. The subspace clustering algorithm that runs on these dimensions generates clustering results that helps in guiding malware analysis. This clustering approach can provide meaningful information relevant to an analysis, especially in cases where investigators are able to focus only on binaries within a small number of clusters that are based on binary code features.

When the initial clustering is completed, the resulting set of clusters gives a general idea about different classes of binary codes based on binary code attributes. The last piece of our malware clustering system is a graph matching module which is responsible for computing the graph similarity of malware samples. The subspace clustering module generates clusters by comparing and evaluating the specific features of the malware binaries. The graph matching process runs on each of the resulting subspace cluster to obtain the actual clusters. In the graph matching phase, malware samples are compared and clustered based on their function call graphs.

Various algorithms and methods have been implemented for matching different graph types. The common problem of these methods is that the computational cost of matching graphs is very high. We implemented the generic graph edit distance algorithm to compare and verify runtime and grouping performance of the algorithms.

We run the system over a set of malware binaries that we collected from various web resources to verify the contributions of our system. In chapter 5, we present the experiment results including clustering accuracy and runtime performance values.

4.2 Implementation of The System

The components of the system were designed and implemented to perform the processes explained in the system design section based on the algorithms introduced in Chapter 3. The system components and the functions of these components are presented in this section.

4.2.1 The System Components

The system has three components, depicted in Figure 16, to be able to perform the tasks necessary for the clustering of a malware set. The Malware Clustering program is the first and the main component of the system since it provides a graphical user interface for managing the feature extraction and clustering processes. By using this user interface, an analyst can start the clustering on a malware set, and examine the generated cluster results. The INSCY, DBSCAN and GED algotihms, used for generating clusters, are also implemented as parts of this first component. After the clustering process is completed, the system displays the clusters on the graphical user interface to allow further examination of the clustering results.

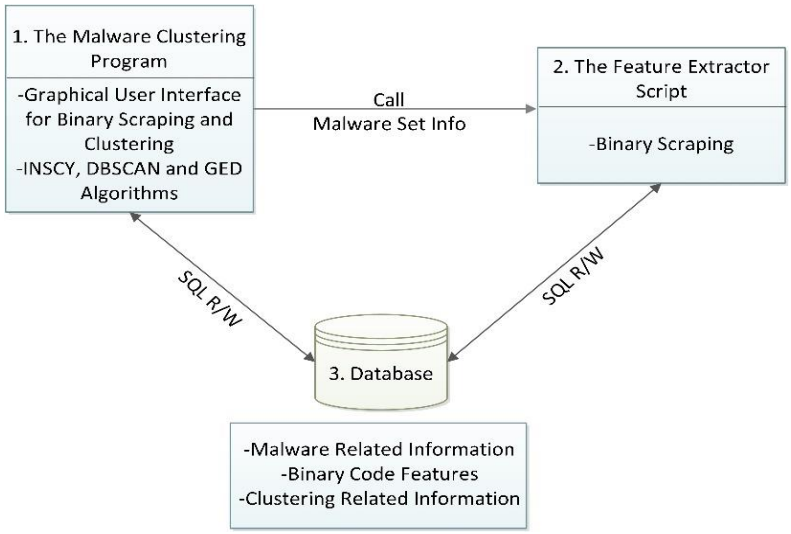


Figure 16: The system components

As we mentioned in Section 4.1, a set of features which are used in the clustering phase should be extracted from malware binary samples prior to the clustering process initialization. The clustering space is formed based on these features. For this purpose, the Malware Clustering Program calls the Feature Extractor script which is the second component written in python. Malware binary paths and the list of features to be extracted from the selected malware set are sent as an argument to the script by the

program. The binary code features extracted by the script are stored in the database. The second task of the Feature Extractor script is to extract all static and dynamic functions and call relations among them. The system creates a function call graph for each malware binary based on its function call relations. The clustering module of the system uses these function call graphs, which are the representations of the malware binaries, to generate malware clusters.

4.2.2 The System Functions

The system generate clusters from a malware set in three steps which are feature extraction, subspace clustering and graph matching.

4.2.2.1 The Feature Extraction Function

To cluster malware samples using a subspace clustering method, we need to construct a clustering space where the malware samples reside. The first information we need to create the space is the dimensionality of the space. The dimensions and the intervals of the dimensions that will form a space should be determined according to the features of the malware binary codes to be clustered. In our system, malware samples are the objects that will be located in the space. Therefore, we use binary code features as the dimensions of the space. These features are binary size, local function count, local function calls count, dll count, dll calls count, basic block count and basic block edge count. In this study, we only deal with the continuous features of the malware binaries.

When we determine the dimensionality of the space, we need to obtain binary code features from the malware set. The features of a binary code gives us the coordinates of that binary in the clustering space. To do this, we use the Feature Extractor script that scrape malware binary codes for their features. The system reads the features of each binary code in the set and stores them into the database. The subspace clustering module of the system then cluster binaries according to their coordinates in the space. The second task of the Feature Extractor is to extract all call relations among local, static and dynamic functions from malware binaries and to turn these relations into graphs which will be used as malware signatures in the graph matching phase.

We implemented the Feature Extraction script in python. As seen in the pseudo code 1 there are two main functions which are the *extract_function_calls* and the *extract_dll_calls*. These two funtions scan all the functions in the given binary code and find function references to build the call graphs. The other four routines were written as helper funtions.

Pseudo Code 1: Feature Extraction

```
Main(Binary File)
        call extract_function_calls(Binary File) function;
        call extract_DLL_calls(Binary File) function;
        write_to_database(allLocalFunctions, functionCalls, functionFeatures,
        allDllFuctionIds, allDllFunctions, importXrefs);

extract_function_calls(Binary File)
        foreach of the function in the Binary file
                foreach of the instruction in the function
                        add instruction to the instruction set
                take_MD5_hash(instruction set)
                take_MD5_hash(function)
                get_basic_block_info (function)
        return allLocalFunctions, functionCalls, functionFeatures;

extract_DLL_calls(Binary File)
        foreach of the dll in the imported dll set in the Binary file
                get all functions that call the dll
                add the functions and the dll into the importXrefs map
        return allDllFuctionIds, allDllFunctions, importXrefs;

take_MD5_hash(string)
        return hash value of the string;

get_stack_size(function)
        return stack size of the function;

get_basic_block_info (function)
        return total basic block number and basic block edge number;

write_to_database(info)
        write info to the database;
```

4.2.2.2 The Subspace Clustering Function

After the dimensions of the space is determined and the database is populated with the binary code coordinates, the Subspace Clustering module steps in and initializes the clustering process. The system employs two clustering techniques, INSCY and DBSCAN, to mine density-based clusters in the sub regions of a clustering space. The

mining process starts with two user inputs: the redundancy index and the epsilon value which are two crucial parameters for the clustering. The redundancy index is used by the INCSY algorithm as the rate of allowed lower dimensional clusters which are similar to higher level clusters. In other words, lower dimensional clusters appear on the result set depending on the redundancy index. With the smaller index values more redundant clusters are generated. The epsilon used by DBSCAN determines the distance between points (in our case between binaries) which form a cluster. So, with the smaller epsilon values, clusters are composed of points that are close to each other. As the epsilon value increases, more dissimilar points will be in the same cluster.

The INSCY algorithm searches for the maximal high dimensional sub regions that contain a certain number of objects. Hence, it determines the sub regions that might include density-based clusters. This algorithm can scan all possible subregions in the space. However, mining a high dimensional space requires excessive effort. To improve the mining performance, the INSCY algorithm uses an in-process redundancy pruning approach that allows the algorithm to prune less informative lower dimensional subspace regions without performing cluster analysis. In this manner, it aims to reduce the runtime for mining clusters.

When the INSCY algorithm finds a subspace region that has an object count above a given threshold, the DBSCAN algorithm is called to find possible density-based clusters in that subregion. A list of clusters is the output of the DBSCAN algorithm. All clusters are written to the database and displayed via the graphical user interface.

4.2.2.3 The Graph Matching Function

The third function of the system is used to create clusters based on function call graphs of malware binary codes. Even though comparing graphs negatively affects the clustering runtime, using internal structures of malware binary codes as the similarity metric improves accuracy of the system's clustering process. For this purpose, a graph similarity comparison mechanism was added to the system to be able to get a better clustering accuracy. The graph matching module of the system employs the graph edit distance method to compare all malware binary pairs reside in a cluster generated by the subspace clustering process.

A basic graph edit distance algorithm is implemented in this study. The pseudo-code displaying the main functions of the algorithm can be seen in Pseudo Code 2.

Pseudo Code 2: GraphEditDistance (Graph1, Graph2)

```
Main()
    call GetDistance()

GetDistance()
```

```
        call CreateCallMatrix function
        int[] indexList = HungarianAlgorithm(costMatrix, 'min')
        foreach index in indexList
                editDistance = editDistance + costMatrix(index)
        return editDistance

CreateCostMatrix()
        n is the node count of G1
        m is the node count of G2
        initialize the costMatrix with a size of [(n+m)x(n+m)]
        call GetInsertCost function
                write results into costMatrix[n+n, m] (Lower-left of the matrix)
        call GetDeleteCost function
                write results into costMatrix[n, m+m] (Upper-right of the matrix)
        call GetSubstituteCost function
                write results into costMatrix[n, m] (Upper-left of the matrix)
        return costMatrix
GetInsertCost()
        Compare nodes and return results
GetDeleteCost()
        Compare nodes and return results
GetSubstituteCost()
        call GetEdgeDistance function
        return results

GetEdgeDistance()
        call CreateEdgeCallMatrix function
        int[] indexList = HungarianAlgorithm(edgeCostMatrix, 'min')
        foreach index in indexList
                edgeEditDistance = edgeEditDistance + edgeCostMatrix(index)
        return edgeEditDistance

CreateEdgeCostMatrix()
        n is the edge count of node1
        m is the edge count of node2
        initialize the edgeCostMatrix with a size of [(n+m)x(n+m)]
        call GetEdgeInsertCost function
                write results into edgeCostMatrix[n+n, m] (Lower-left of the matrix)
        call GetEdgeDeleteCost function
                write results into edgeCostMatrix[n, m+m] (Upper-right of the matrix)
        call GetEdgeSubstituteCost function
                write results into edgeCostMatrix[n, m] (Upper-left of the matrix)
```

return edgeCostMatrix

GetEdgeInsertCost()
Compare edges and return results
GetEdgeDeleteCost()
Compare edges and return results
GetEdgeSubstituteCost()
Compare edges and return results

4.2.3 The System GUI

The system has a graphical user interface that allows analysts to manage malware clustering process. The GUI of the system was written in C# and has two main windows that provide the functions that we explain in this section. In the first user interface, malware feature extraction window, an analyst can select a folder where a malware set located or can select specific malware samples as seen in Figure 17.

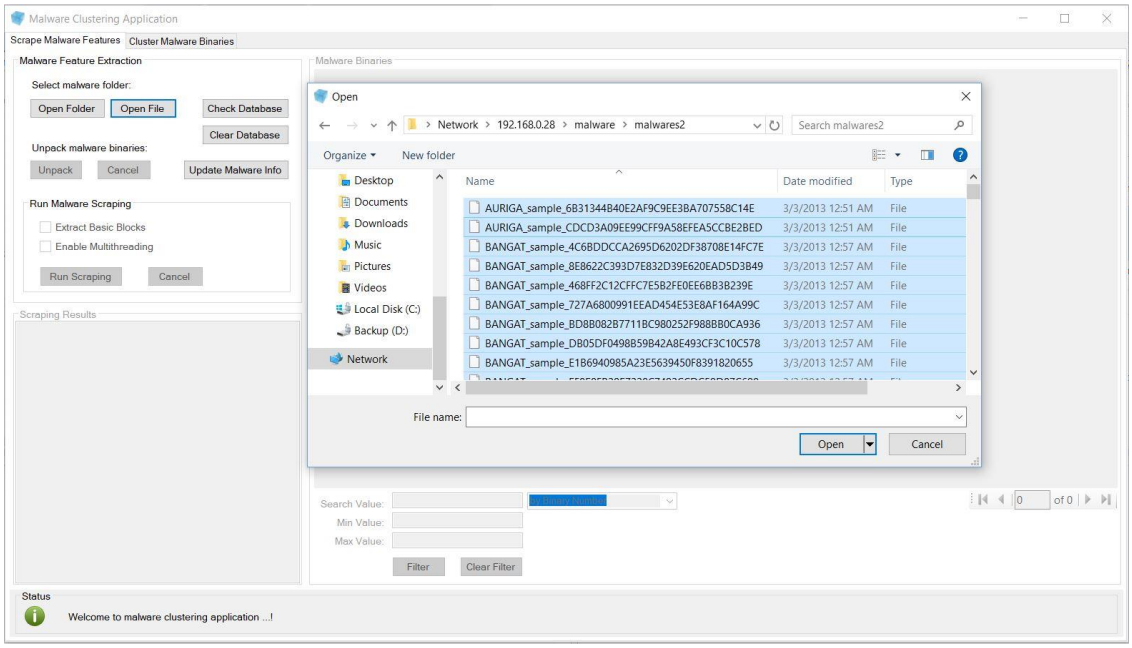


Figure 17: Selecting malware samples

After selecting a malware set, the analyst starts the scraping process which calls the feature extraction script running for each of the malware binary in the selected set. Then, all the features extracted from malware samples are written into database and displayed on the GUI as seen in Figure 18.

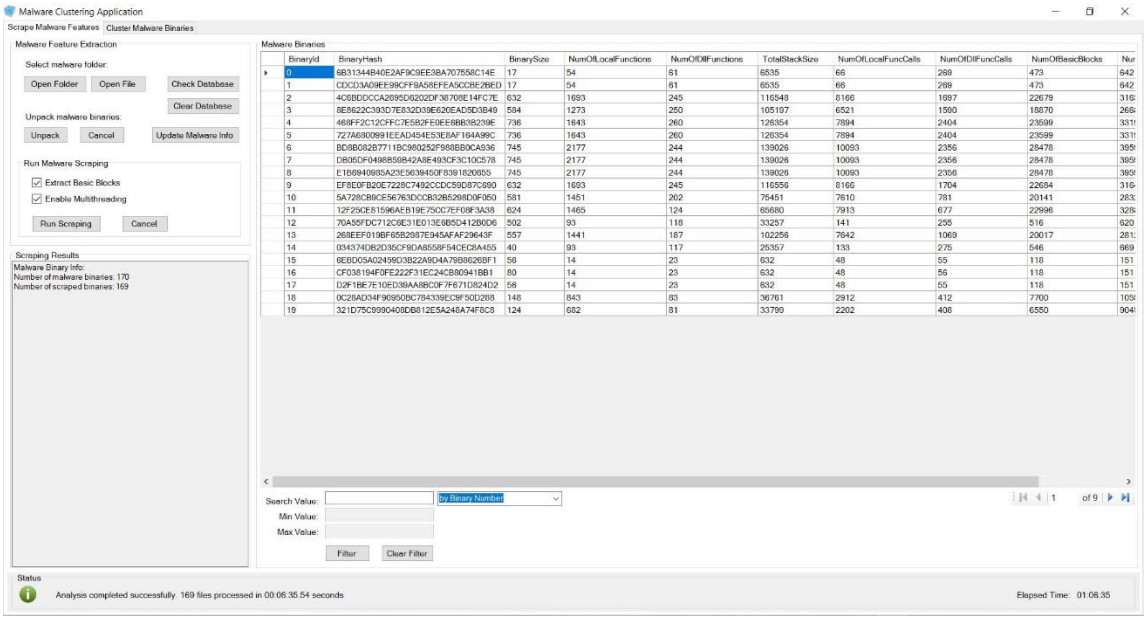


Figure 18: Results of malware feature extraction process

Figure 19 depicts the sample malware binaries and the Ida pro files created for each of these files.

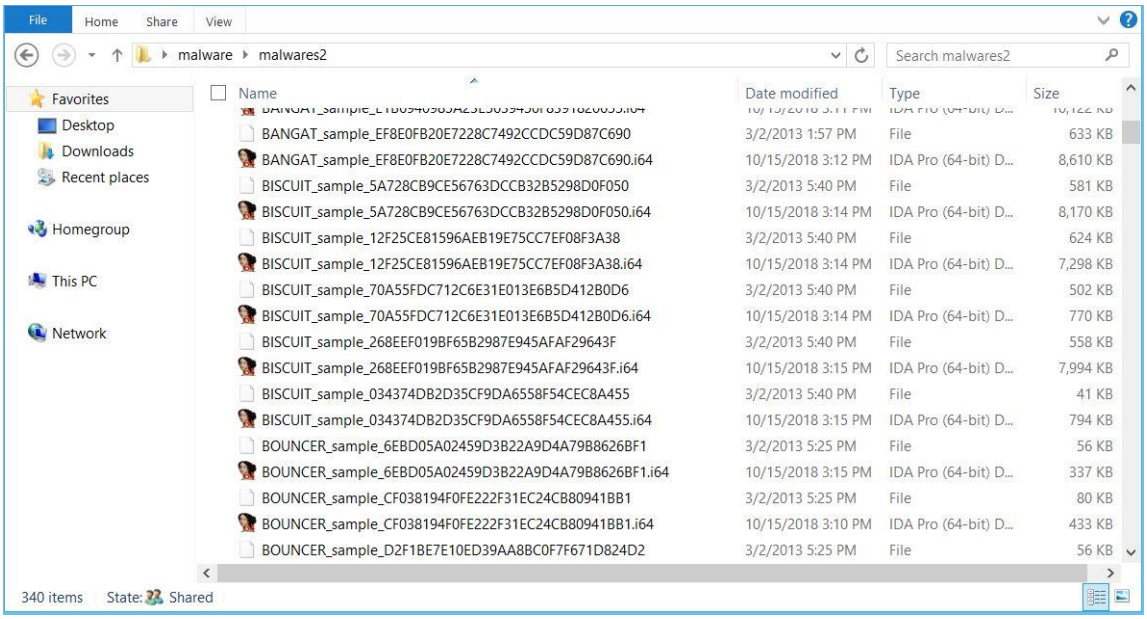


Figure 19: Ida pro files created for each malware

In the second window, malware clustering window, the first task is to select malware samples to be clustered as seen in Figure 20. Then, the input parameters to be given to subspace clustering process are entered to the related fields on the related windows such as dimensions, intervals, redundancy parameter and epsilon value as shown in Figure 21.

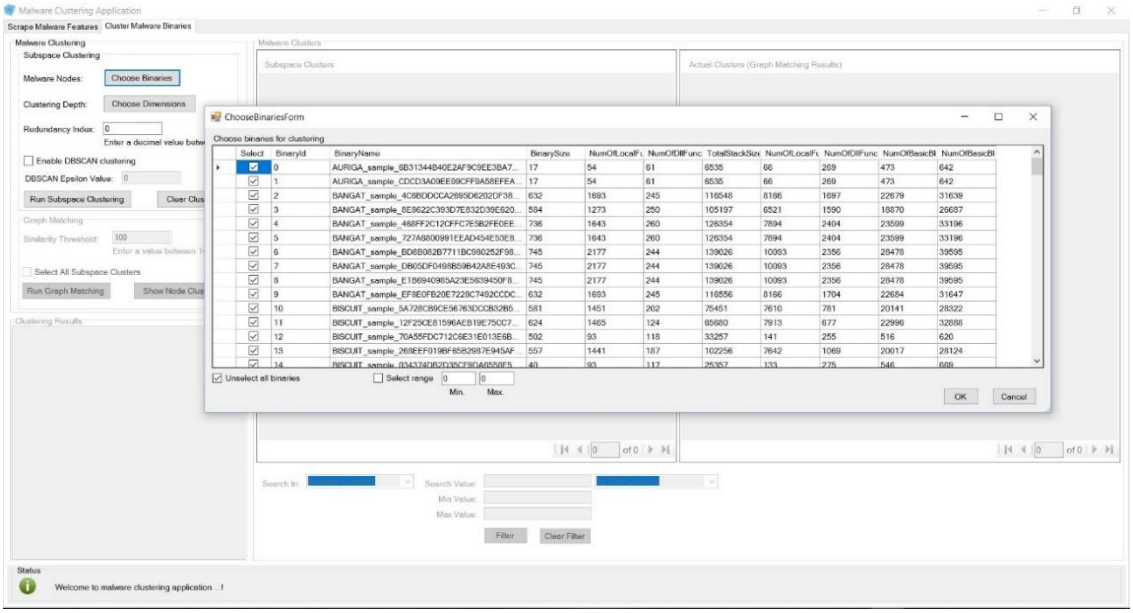


Figure 20: Selecting malware samples for subspace clustering

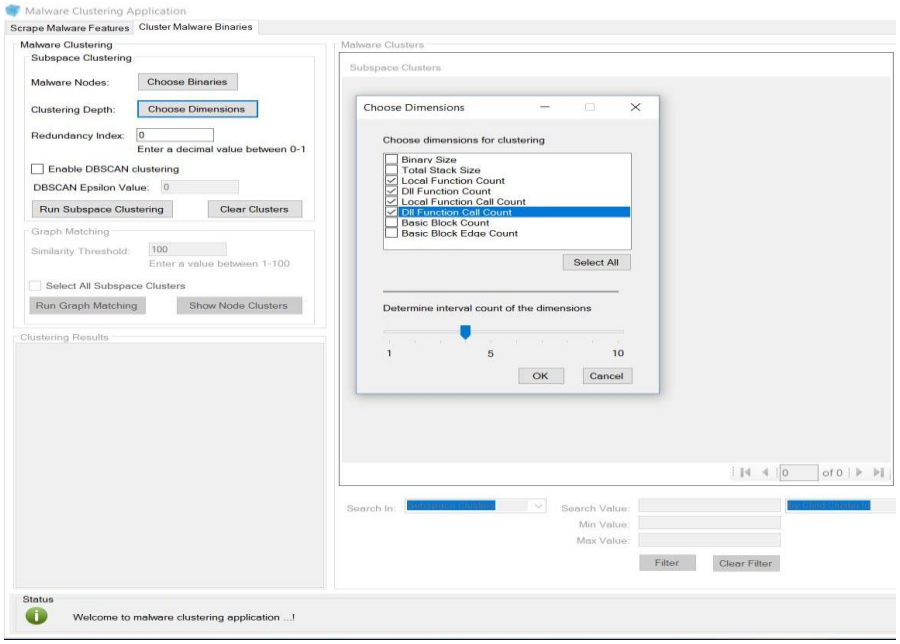


Figure 21: Selecting dimensions for subspace clustering

Generated subspace clusters are listed on the same window. Figure 22 depicts a sample output of resulting subspace clusters. In this example, there are 17 clusters. Contents of a specific cluster can also be examined in detailed by just clicking that cluster as seen in Figure 23. The content window for a cluster shows the malware samples that reside in that cluster.

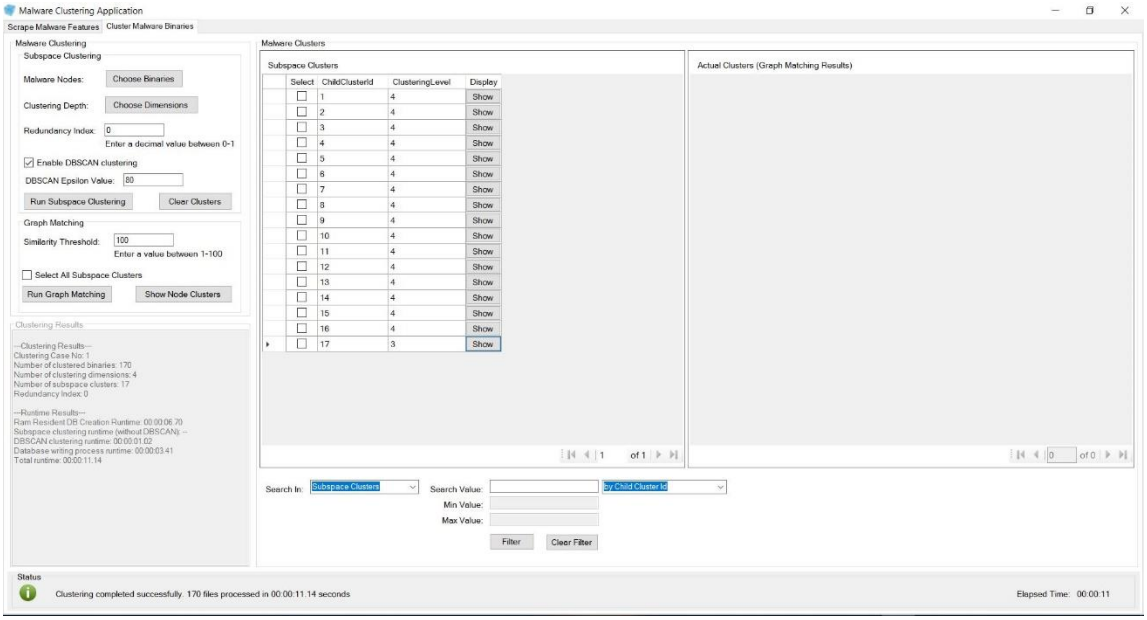


Figure 22: Subspace clustering results

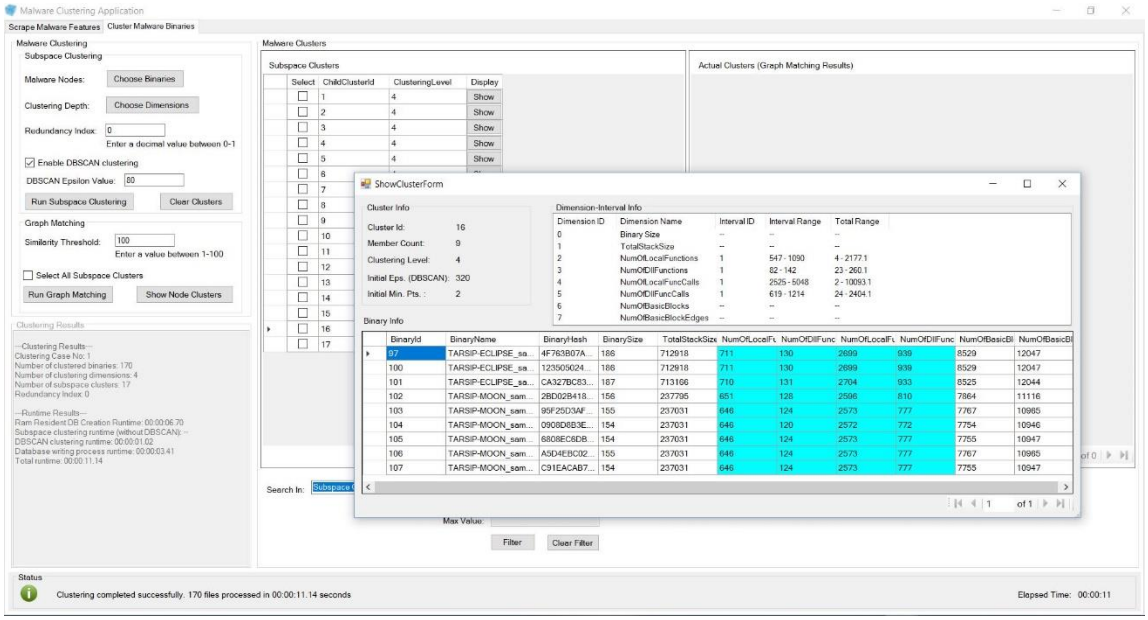


Figure 23: Displaying the content of a cluster

Finally, the graph matching process is run on the selected subspace clusters to generate and display final clustering. Figure 24 shows the screenshot of final clusters. The subspace clusters are turned into final clusters as shown in Figure 25.

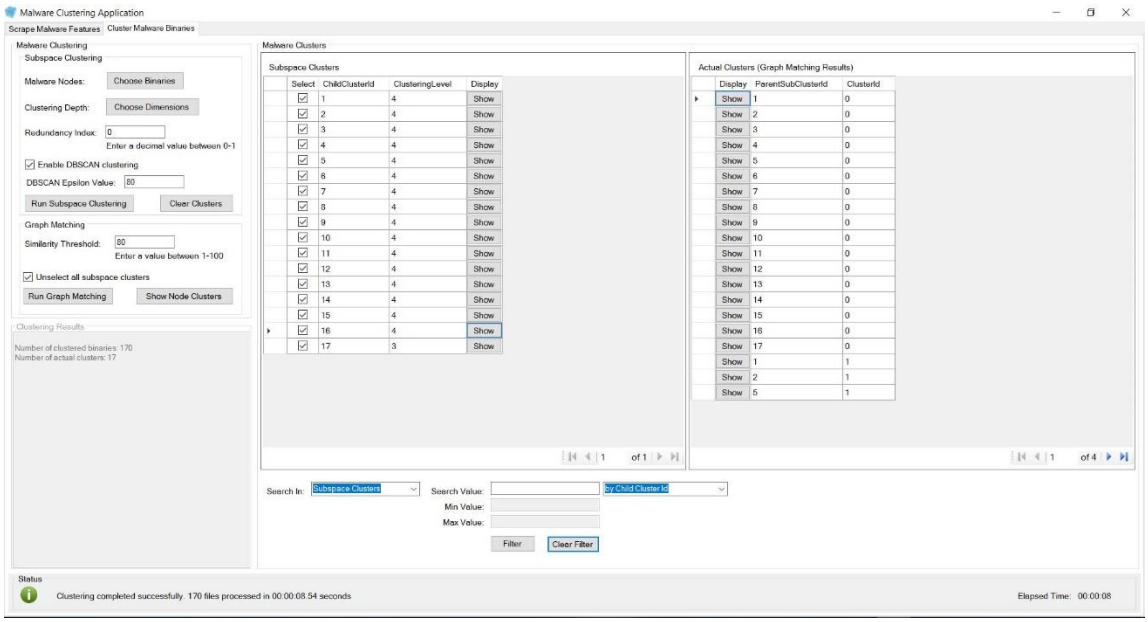


Figure 24: Running graph matching and generating final clustering

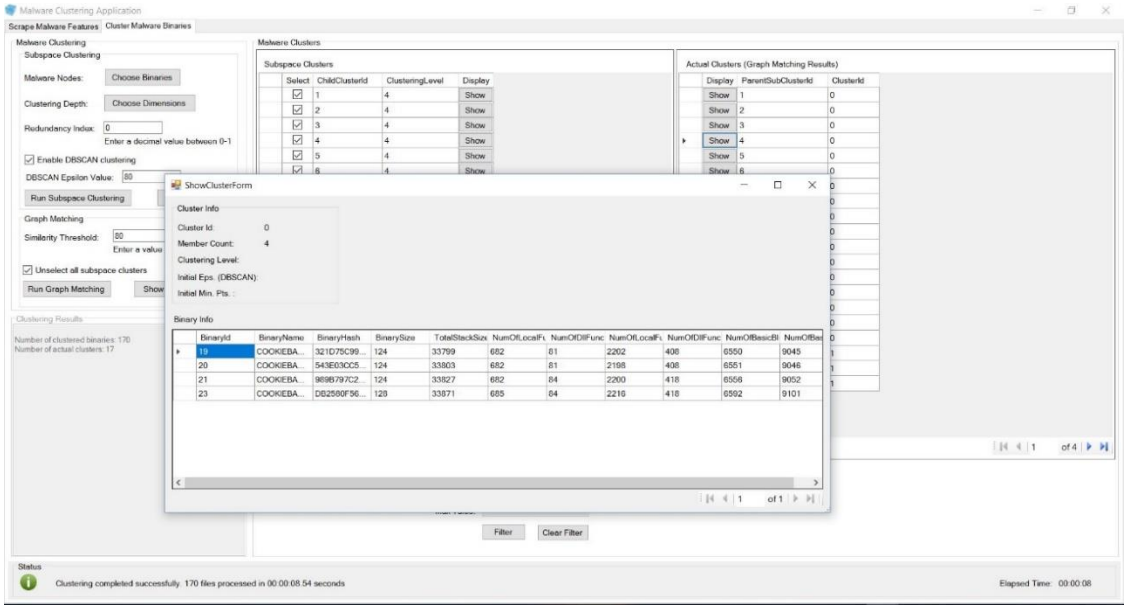


Figure 25: Displaying the content of a final cluster

5. CLUSTERING EXPERIMENT RESULTS

We run our system on two malware sample sets to validate the system's clustering functions, and to measure the clustering accuracy and runtime performance of the system. The first sample set is the APT set, downloaded from contagiodump.blogspot web site (Contagio, 2018), composed of 30 malware families and contains 170 malware samples in total. Mandiant, a cyber security company, grouped the APT malware samples by performing a guided analysis (Mandiant, 2019). Mandiant team carried out a manual analysis, however they utilized some automation tools such as Redline, which is the Mandiant's free tool. The second sample set is the Zeus set, downloaded from virusshare.com web site (Virusshare, 2015), contains 1200 malware samples in total which are manually analyzed and grouped. Table 1 shows the algorithm parameters given to the system as user input. We performed experiments by running the system under different conditions and observed the effects of these parameters on clustering.

Table 1: System inputs

Parameter	Algorithm	Function of the Parameter
Epsilon	DBSCAN	The epsilon determines the distance between points (in our case between binaries) which form a cluster. So, with the smaller epsilon values, clusters are composed of points that are close to each other. As the epsilon value increases, more dissimilar points will be in the same cluster.
Minimum Points	DBSCAN	*Minimum Points* is a metric used for finding core points. A point p is a core point if at least *Minimum Points* count points are within distance ε of it (including p). If p is a core point, then it forms a cluster together with all points (core or non-core) that are reachable from it.
Clustering Space	INSCY	The clustering algorithm creates clusters based on the selected features of the binary codes. Each feature is considered as a dimension in the clustering space. Clusters are formed on the dimensions chosen by the user.
Similarity Threshold	Graph Matching	The user determines the similarity threshold which is between 1-100. For each pair of the binaries in a dataset, the graph matching is run to learn if the two binaries are similar. If the similarity value of a binary pair is bigger than the similarity threshold, then it is said that they are similar.

5.1 Clustering Validation Methodology

We calculated clustering performance measurement metrics such as Rand, Adjusted Rand, Mallow and Jaccard indices in order to evaluate clustering results of the system. These clustering performance evaluations metrics require a knowledge of ground truth. These metrics do not take absolute values of the cluster labels into account, but rather consider if the clustering separate data similar to that of ground truth clusters. As the ground truth, we used the clusters given by the Zeus and APT data set providers.

The definition of these indices are based on number of pairs that are grouped in the same way in both clusterings, i.e. pairs of binaries of a binary set that are in the same cluster under both clusterings (S & D, 2012). Table 2 shows all the types of set of binary pairs. The clustering metrics that we used in this study is calculated based on the pair counts given in Table 2 where C refers to the ground truth true clusterings given with the APT and Zeus malware sets and C' refers to the predicted clusterings generated by our system.

Table 2: Pair sets definitions

Pair Set	Pair Detail
n11	"pairs that are in the same cluster under C and C'"
n00	"pairs that are in different clusters under C and C'"
n10	"pairs that are in the same cluster under C but in different ones under C'"
n01	"pairs that are in different clusters under C but in the same under C'"
C: True clustering, C': Predicted clustering	

We used the indices shown in the Table 3 to measure the clustering performance of the system modules. Particularly, Rand index, Adjusted Rand index and Mallow index are well known and generally used in clustering performance evaluation.

Table 3: Clustering performance measurement metrics

Parameter	Explanation	Formula
Rand Index	"It calculates the fraction of correctly clustered (respectively misclassified) elements to all elements" (Rand, 1971).	R(C,C') = 2(n11 + n00) / n(n-1)
Adjusted Rand Index	"The adjusted Rand index is the corrected-for-chance version of the Rand index." (Hubert & Arabie, 1985)	ARI = (RI - Expected_RI) / (Max(RI) - Expected_RI)
Jaccard Index	"It is very similar to the Rand Index, however it disregards the pairs of elements that are in different clusters for both clusterings" (Jaccard, 1902).	J(C,C') = n11 / (n11 + n10 + n01)
Mallows Index	"It is defined based on the number of points that are common or uncommon in two clusterings" (Fowlkes & Mallows, 1983).	M(C,C') = n11 / $\sqrt{(n11 + n10)(n11 + n01)}$

F-Measure	"The F-Measure is used to evaluate the accuracy of a clustering solution" (Van Rijsbergen, 1979).	F1(C,C') = 2.P.R / (P+R), where Precision P=n11/(n11+n01), Recall R=n11/(n11+n10)

5.2 Clustering Experiment Results

Two distinct malware sets were used to validate the system's clustering operation. The results of the experiments are presented in this section.

5.2.1 The APT Malware Sample Set :

The APT malware binary set was used in the first experiment. This set is composed of 170 malware samples. The provider of the set, Mandiant, grouped the samples into 32 clusters by performing a guided analysis (Mandiant, 2019). Appendix A lists the APT malware families and related cluster identifiers. This grouping is used as ground truth in cluster performance evaluation in this experiment.

Clustering and runtime performances of the system was evaluated by observing the effects of the pre-clustering and the graph matching parameters given in Chapter 5. Experiments were performed with the malware set and the server configuration listed in Table 4. Table 1 shows the algorithm parameters given to the system as user input.

Table 4: Experiment setup configuration

Malware Set	**Test Server Specifications**
APT Malware Set **(True clustering)** **Malware sample count: 170** **Cluster count: 32**	Processor: Intel i7-4700HQ 2.4 GHz Memory: 16GB Operating system: 64-bit Windows 10

5.2.1.1 Experiment 1: Graph Matching Similarity Performance

In experiment 1, the graph matching algorithm was run on the APT malware binary set without running pre-clustering to observe only the graph matching clustering performance. Figure 26 shows the effect of similarity threshold on clustering performance. Similarity threshold is the parameter used for measuring the similarity of binary pairs. When it is set to lower values such as 10 and 20, the system generates a clustering less similar to the true clustering. For example, the similarity threshold of 10 results in lower rand, mallow and jaccard index values, 0.58, 0.3 and 0.09 respectively. The reason of this lower clustering performances is that dissimilar malware binaries are more likely to be grouped in the same cluster due to the small similarity threshold value.

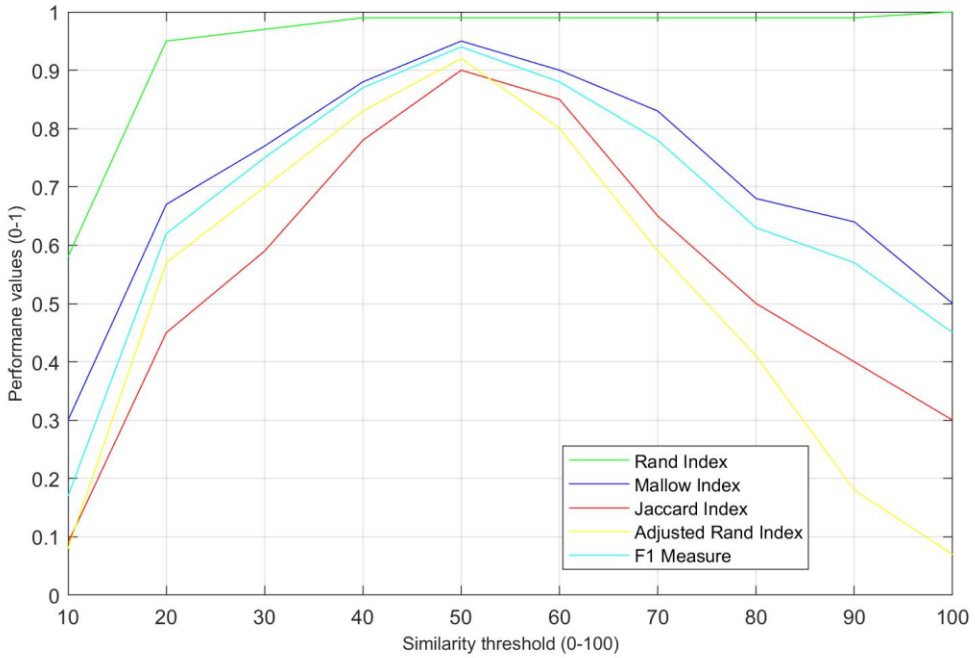


Figure 26: Clustering accuracy

Similarly, high similarity threshold values (90-100) reduce the clustering performance. When it was set to 100, mallow and jaccard indices were calculated as 0.5 and 0.3 respectively. The reason of this result is that the system clustered the binaries which have the exact same function call graph structure. As a result, some clustering information lost during the clustering process. Despite this missing information, if we look at the rand index curves, the true clustering information was correctly generated at high similarity threshold values, contrary to the rand index value obtained at similarity threshold 10. Lastly, if we look at the middle similarity threshold values, we can see that the system shows the best clustering performance (Mallow index: 0.96, Rand index: 0.98) at these values. When we compare the rand index and mallow index, we can see that the rand index value is approaching the 100 percent as the similarity threshold goes high, however, mallow index goes down after a certain threshold value. The reason of this difference is that, rand index shows the only accurately clustered binary percentage, that is, it does not consider the missing information. In mallow index, since the missing information is considered, it gives lower values at the high similarity values.

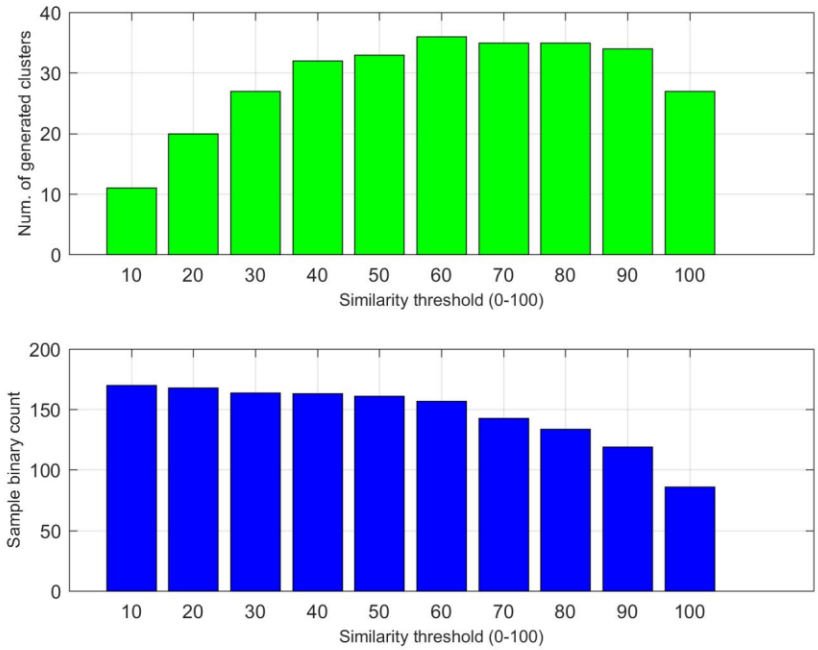


Figure 27: Clustering results

Figure 27 displays the clustering accuracy in terms of cluster and binary counts. The upper half of the graph gives the cluster count generated by the system at various similarity threshold values. When similarity threshold was set to 10, 11 clusters containing 169 binaries in total were generated. But the generated clustering did not reflect the true clustering because the true clustering had 32 clusters. If we look at the similarity threshold 100, the count of the generated clusters is close to the true clustering cluster count. But the lower sub-graph shows that almost half of the clustering information lost at this similarity threshold value. Because in the true clustering, even if some binaries don't have the exact function call structures, they might be grouped into the same cluster. At the middle (40 to 60) similarity threshold values, cluster and binary counts of the generated clustering are almost same with the true clustering values.

The accuracy values can also be observed by looking the confusion matrices displayed on Figure 28. The Figure shows the true clustering versus the generated clustering. Indices of the figure refer to the cluster identifiers on both axes. Malware family class names related to these cluster identifiers are given in the Appendix A. In lower similarity threshold values, the generated clusters by the prototype did not reflect the true clustering. Malware samples that are in the same cluster in the true clustering were placed in different clusters by the system. In middle similarity threshold values, the prototype generated clustering that is similar to the true clustering. Lastly, in the higher similarity values, clustering accuracy went down as seen in the lower similarity values.

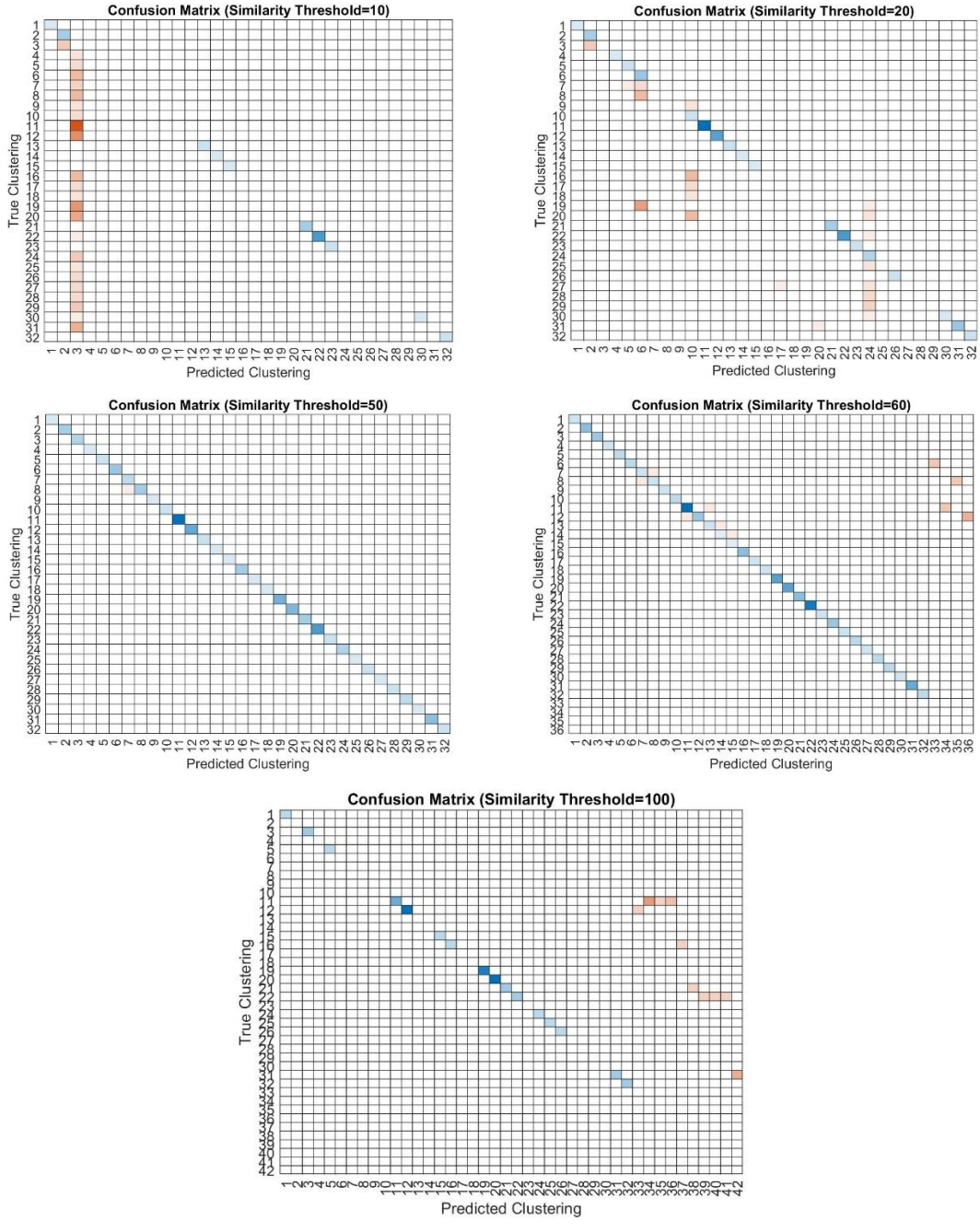


Figure 28: Confusion matrices for different similarity thresholds

Figure 29 shows the function call count distribution for each cluster in the clustering generated by the system against changing similarity threshold values. At the similarity threshold values 10 and 40, some clusters have bigger standard deviations. It means that less similar binaries in terms of function call structure might be found in the same cluster. Clustering processes with the similarity thresholds 80 and 100, on the other hand, generated more flat clusters as expected.

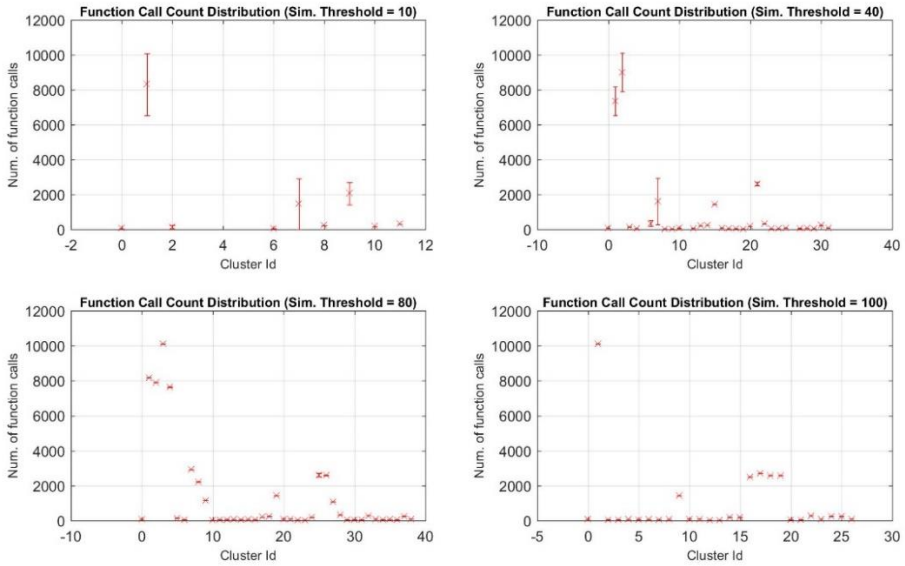


Figure 29: Function call count distributions vs. similarity threshold

Consequently, the graph matching algorithm we used for final clustering in our system generates very good clustering results. Especially at the middle similarity thresholds, more realistic clustering can be obtained for any binary set. However, if exact matching between binaries is wanted in an analysis then higher similarity threshold values can be set.

5.2.1.2 Experiment 2: Pre-clustering Clustering Similarity Performance

We used the INSCY and DBSCAN algorithms together for the pre-clustering process. By performing the pre-clustering, we aim to reduce the runtime of graph matching process without losing clustering information. We choose the DBSCAN algorithm because the clustering space contains distinct nodes in the concept of this

thesis. In other words, there are no hierarchy or such relations between malware binaries. Thus, a density-based clustering algorithm is more suitable for the pre-clustering process. Besides, the INSCY algorithm provides an index structure and reduces runtime of DBSCAN algorithm especially in a high dimensional space.

This experiment measures the effect of pre-clustering on the graph matching performance. Figure 30 shows the DBSCAN algorithm effect on clustering with various epsilon and similarity threshold values. The upper-left graph gives the mallow index values against epsilon value. As epsilon value increases, mallow index increases as well depending on the similarity threshold. Increase in the epsilon value causes higher changes in mallow index when lower similarity values are chosen, because larger epsilon values cause DBSCAN algorithm to cluster distant binaries. For example, the yellow and blue lines (similarity thresholds 10 and 20) show the difference in mallow index against the increasing epsilon value. The difference in mallow index , however, is low for the higher similarity thresholds such as 80 and 100 (green and purple lines). The reason of this is that graph matching running with higher similarity thresholds produce clusters containing binaries closer to each other. Similarly, using smaller epsilon values clusters closer binaries in the pre-clustering process. Since the graph matching runs on the clustering generated by pre-clustering process, the effect of epsilon value on the graph matching results is small. Besides, rand index curves are very satisfactory at any epsilon values as seen in the upper-right graph. The rand index value is also low when the system runs with the similarity threshold of 10 without pre-clustering. Lastly, as seen in the third graph, the F1 measures are satisfactory when the epsilon value is set above 100 for the similarity thresholds above 50.

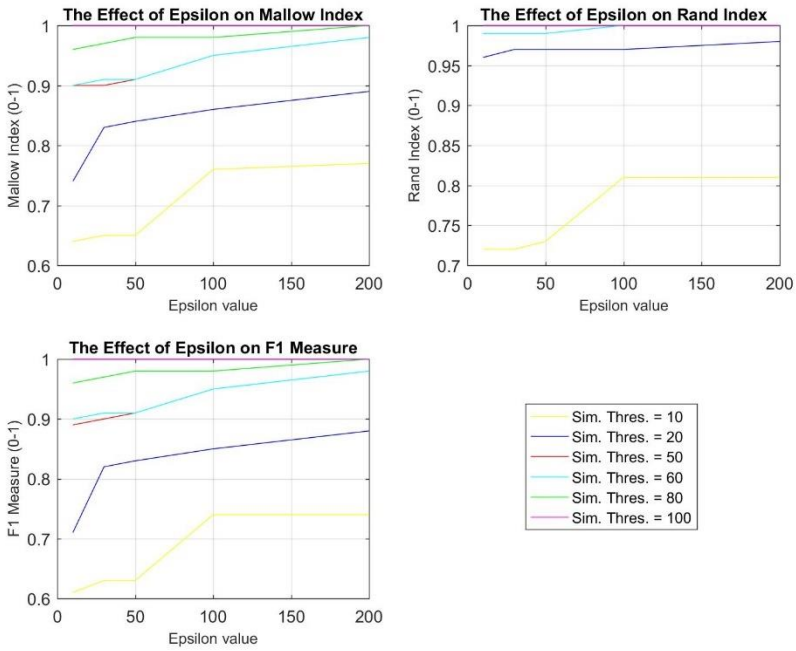


Figure 30: The effect of the epsilon on clustering accuracy

Dimension count is also an important factor in clustering malware binaries. High dimensional data sets require more clustering effort and time. In this experiment, binary sets have 6 features (dimensions) which are listed below. The number of dimensions can also be changed depending on the clustering case and the target set.

Table 5: Features (Dimensions) of a malware binary

Local function count
Dll count
Local function call count
Dll function call count
Basic block count
Basic block edge count

Figure 31 shows the effect of changing dimension count and similarity threshold on clustering. Since each of the dimension which we choose for clustering stores a part of information about binary function call graph structure, we didn't observe drastic changes in the performance values against increasing dimension count. Rand index values are quite good for all the similarity threshold except 10. When similarity

threshold is set above 50, we get satisfactory mallow index values with the increase of dimension count.

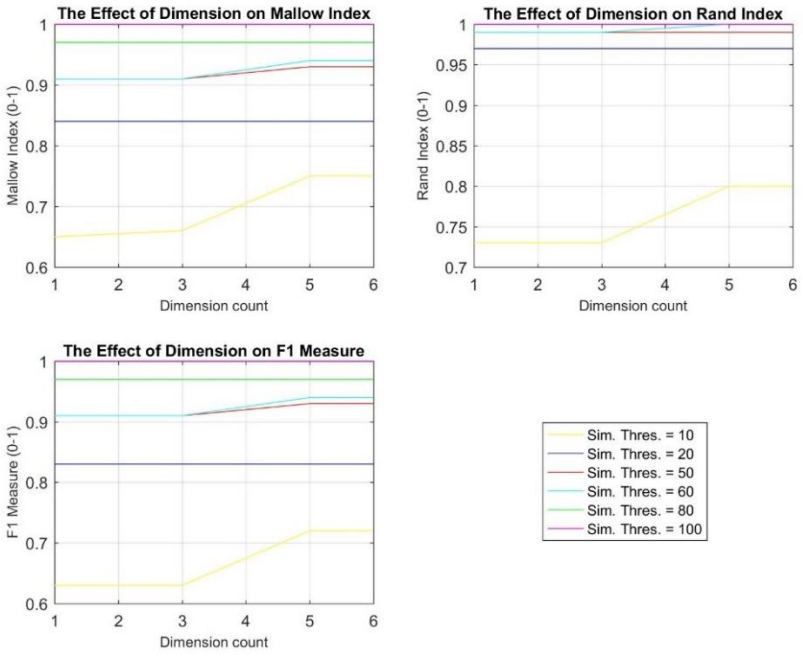


Figure 31: The effect of the dimension on clustering accuracy

5.2.1.3 Experiment 3: Pre-clustering and Graph Matching Runtime Performances

In this experiment, we run the system with and without pre-clustering to observe the runtime remediation. Figure 32 displays the CPU and I/O runtime results of the clustering processes of the system. The upper left graph gives the runtime of graph matching process without pre-clustering. As binary count increases the graph matching runtime increases in exponential. Because, the count of binary pairs that will be compared by the system increases in exponential in parallel with the binary count increase. The complexity of the graph edit distance is $O(n^3)$.

The upper right graph shows the effect of pre-clustering on runtime. The total runtime notably decreased when the two clustering methods were performed sequentially. While the light blue curve represents the pre-clustering process runtime, the blue curve displays the graph matching runtime. The green curve gives the total runtime. With the pre-clustering, the system generated clusters almost four times faster. As seen in the same figure, the pre-clustering makes the runtime curve linear even for the large binary sets by feeding the graph matching process with small binary groups containing more similar binaries. The lower half of the Figure 32, superimposed of the

two upper graphs, displays the difference in runtimes. Lastly, since the system uses a database to store binary and clustering information, the I/O time is much higher than computation time of clustering algorithms.

In this study, our purpose in using pre-clustering is to reduce runtime of graph matching without degrading clustering accuracy. The first two experiment results show that pre-clustering process makes the targeted binary set ready for the graph matching process with preserving true clustering information. The third experiment showed that our implementation of pre-clustering significantly improves the runtime performance of total clustering process.

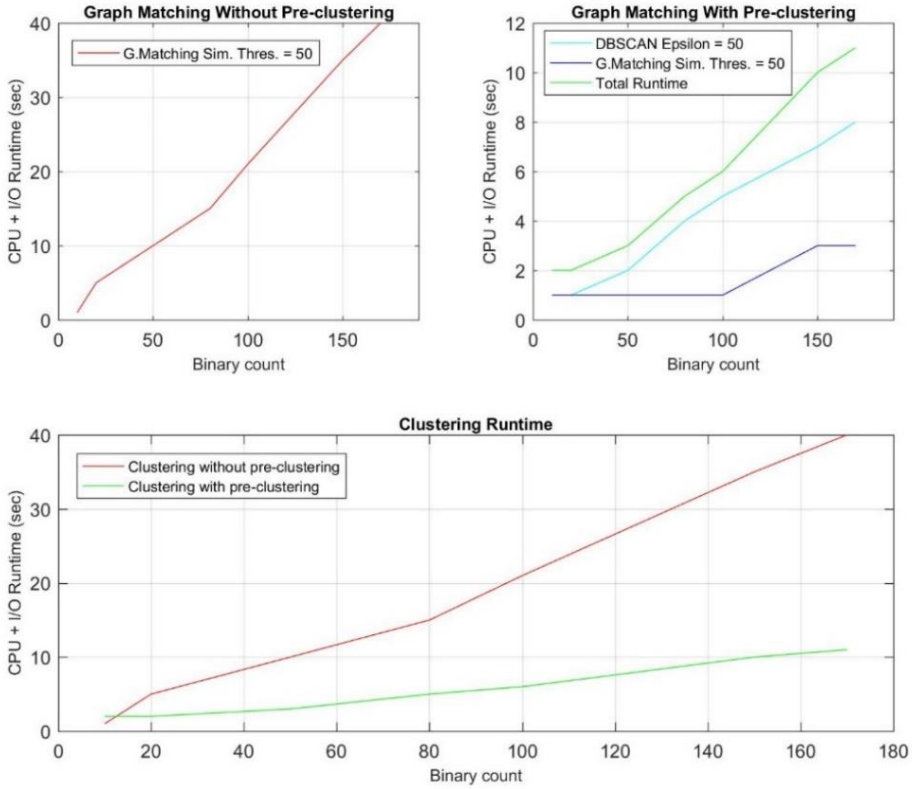


Figure 32: Clustering runtime performance

5.2.2 The Zeus Malware Sample Set :

The second clustering experiment was performed on the Zeus malware binary set. Similar to the previous experiment, three tests were performed with changing clustering parameters in order to measure the system clustering accuracy and runtime performance. The results of this experiment are consistent with the results of the tests

performed on the APT malware set. In other words, our system shows similar clustering behavior for different malware sets.

The Zeus malware binary set contains 1200 sample. The provider, virusshare.com, manually analyzed and grouped this set into 170 clusters. Clustering accuracy and runtime performance of the system was evaluated by observing the effects of the pre-clustering and the graph matching parameters given in Chapter 5. Tests were performed with the following malware set and the server configuration.

Table 6: Experiment setup configuration

Malware Set	**Test Server Specifications**
Zeus Malware Set (True clustering) Malware count: 1200 Cluster count: 170	Processor: Intel i7-4700HQ 2.4 GHz Memory: 16GB Operating system: 64-bit Windows 10

5.2.2.1 Experiment 1: Graph Matching Clustering Similarity Performance

In this experiment, graph matching algorithm was run on the binary set without pre-clustering to observe only the graph matching clustering performance. Figure 33 shows the effect of similarity threshold on clustering performance. When the similarity threshold is set to lower values (i.e. 10 to 20), the system generates the predicted clustering less similar to the true clustering. The reason of this lower clustering performances is that dissimilar malware binaries are more likely to be grouped in the same cluster due to the small similarity threshold. In this experiment, when the similarity threshold is set to 10, the similarity values are higher according to the results of the APT malware set experiment. The reason is that in the true clustering of Zeus set, clusters are formed with malware binaries which have less similar function call graphs.

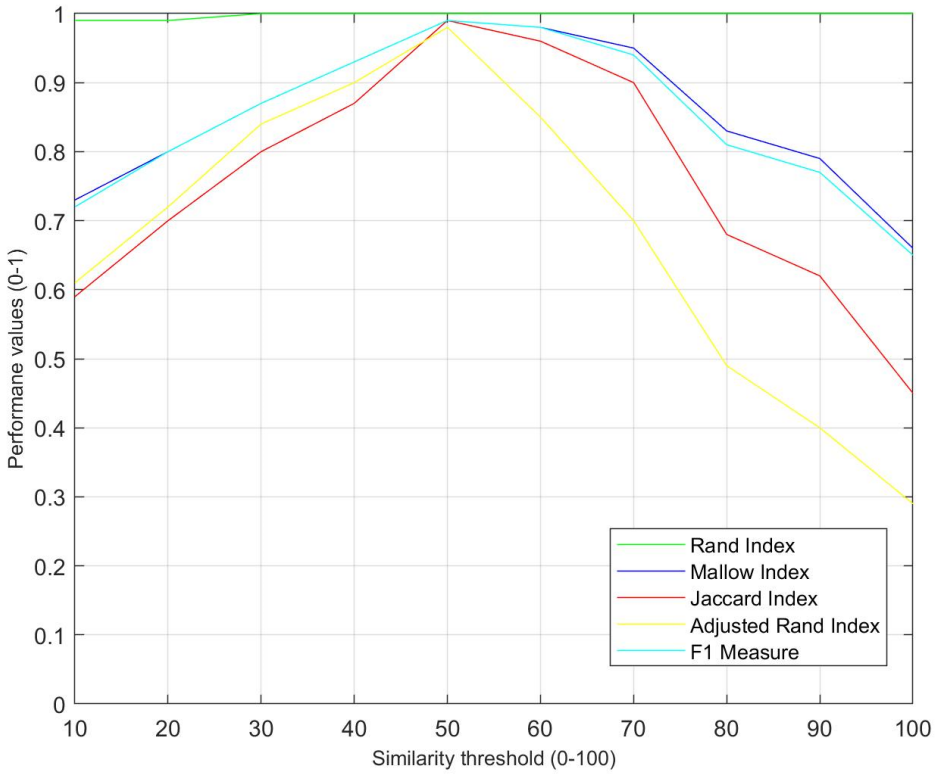


Figure 33: Clustering accuracy

The high similarity threshold values (i.e. 90-100) reduce the rand index values similar to the first experiment. When it was set to 100, mallow index was calculated as 0.66. The reason of this result is that the system clustered the binaries which have the exact same call graph structures. As a result, some clustering information lost during the clustering process. Despite this missing information, if we look at the rand index curve, true clustering information can be obtained at the high similarity threshold values. Lastly, if we look at the middle similarity threshold values, we can see that the system shows the best clustering performance (i.e. mallow index: 0.95, rand index : 0.98) at these values. When we compare the rand index and mallow index, we can see that the rand index value is approaching the 100 percent as the similarity threshold goes high, however, mallow index goes down after a certain threshold value. The reason this difference is that, rand index shows the only accurately clustered binary percentage, that is, it does not consider the missing information. In mallow index, since the missing information is considered, it gives lower values at the high similarity values.

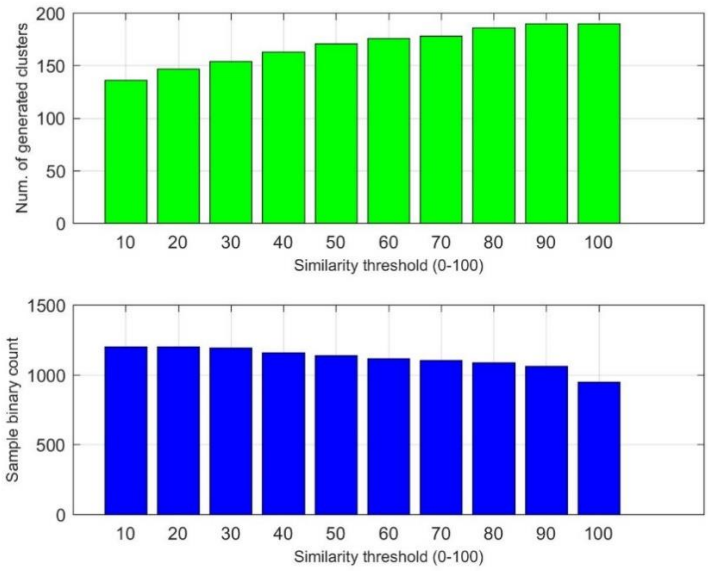


Figure 34: Clustering results

Figure 34 displays the clustering performance in terms of cluster and binary counts. The upper half of the graph gives the cluster count generated by the system at a range of similarity threshold values. When similarity threshold was set to 10, the 140 clusters containing 1200 binaries in total were generated. But the generated clustering did not reflect the true clustering fully because the true clustering has 170 clusters. If we look at the similarity threshold 100, the count of the generated clusters is close to the true clustering cluster count. But the lower sub-graph shows that almost one quarter of the clustering information lost at this similarity threshold value. At the middle (i.e. 40 to 60) similarity threshold values, cluster and binary counts of the generated clustering are almost same with the true clustering values.

Figure 35 shows the function call count distribution for each cluster in the clustering generated by the system against changing similarity threshold values. At the similarity threshold values 10 and 20, some clusters have bigger deviations on the function call counts of the samples in the cluster. It means that less similar binaries might be found in the same cluster. Clustering processes with the similarity thresholds 80 and 100, on the other hand, generated more flat clusters as expected.

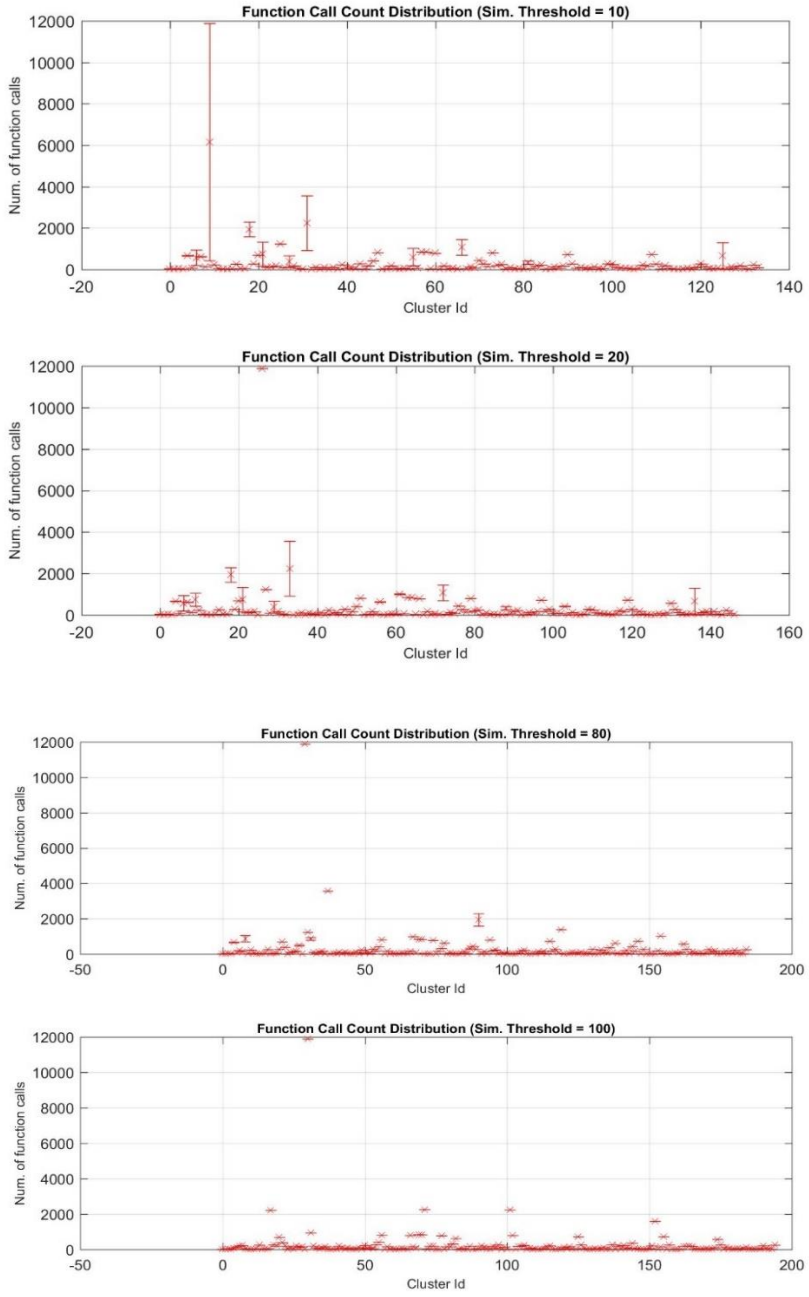


Figure 35: Function call count distributions vs. similarity threshold

Consequently, the graph matching algorithm we used for actual clustering in our system generates good clustering results and verifies the previous experiment. Especially at the middle similarity thresholds, more accurate clusterings are obtained. However, if

an exact match between binaries is wanted in an investigation, higher similarity threshold values can be chosen.

5.2.2.2 Experiment 2: Pre-clustering Clustering Similarity Performance

This experiment measures the effect of pre-clustering on graph matching performance for the Zeus malware set. Figure 36 shows the pre-clustering effect on clustering with changing epsilon and similarity threshold values. The upper-left graph gives the mallow index curves. As epsilon value increases, mallow index increases as well depending on the similarity threshold. Increase in the epsilon value causes higher changes in mallow index when lower similarity values are chosen, because larger epsilon values cause pre-clustering to cluster distant binaries. As an example, the yellow and blue lines (similarity thresholds 10 and 20) show the difference in mallow index against the increasing epsilon value. The difference in mallow index, however, is very low for the higher similarity thresholds such as 80 and 100 (green and purple lines). Besides, rand index curves are very satisfactory at any epsilon values as seen in the upper-right graph. The rand index value was slightly low when the system run with the similarity threshold of 10 without pre-clustering. Lastly, as seen in the third graph, the F1 measure curves are satisfactory when the epsilon value is set above 50 for the similarity thresholds above 50.

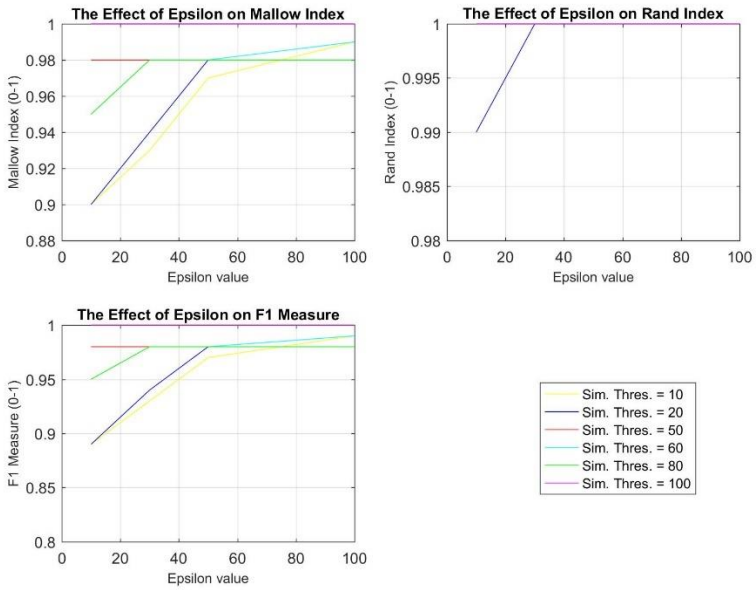


Figure 36: The effect of the epsilon on clustering accuracy

Dimension count is also important factor in clustering the spatial data. High dimensional datasets require more clustering effort and time. In this test, binary sets have 6 features (dimensions) similar to the APT malware set experiment. Figure 37 shows the effect of pre-clustering on total clustering at changing dimension counts and similarity threshold values. Since each of the dimension which we choose for clustering stores a part of information about binary call graph structure, we could not observe drastic changes in the performance values against increasing dimension count. Rand index values are quite good for all the similarity threshold except 10. When similarity threshold is set above 50, we get satisfactory mallow index values with the increase of dimension count. The experiment results prove that pre-clustering process makes the targeted binary set ready for the graph matching process with preserving true clustering information.

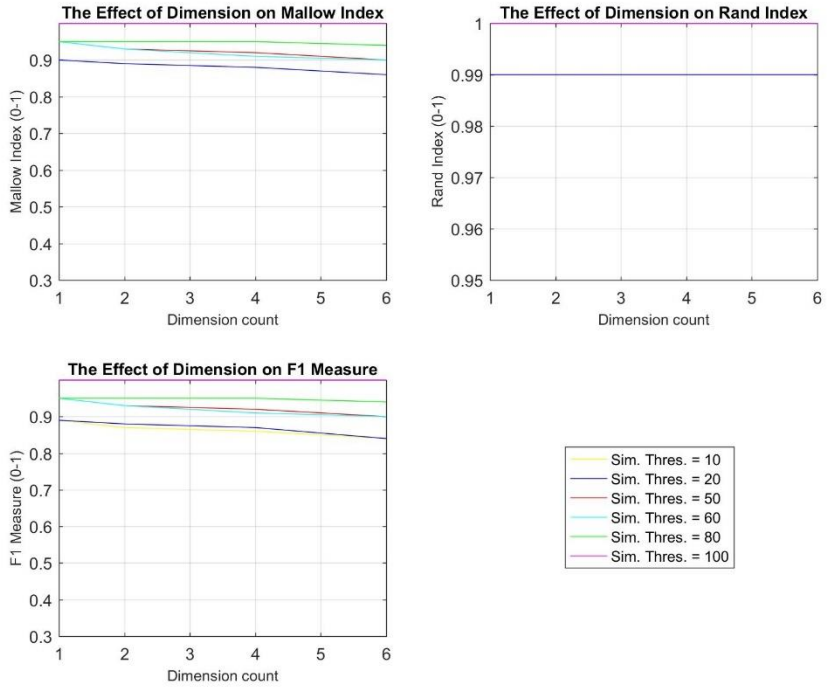


Figure 37: The effect of the dimension on clustering accuracy

5.2.2.3 Experiment 3: Pre-clustering and Graph Matching Runtime Performances

In this experiment, we run the system with and without pre-clustering to observe the runtime remediation. Figure 38 displays the CPU and I/O runtime results of the clustering processes of the system. The upper left graph gives the runtime of graph matching process without pre-clustering. As binary count increases the graph matching

runtime increases in exponential. Because, the count of binary pairs to be compared by the system increases in exponential in parallel with the binary count increase.

The upper right graph shows the effect of pre-clustering on runtime. The total runtime notably decreased when the two clustering methods were performed sequentially. While the light blue curve represents the pre-clustering process runtime, the blue curve displays the graph matching runtime. The green curve gives the total runtime. With the pre-clustering, the system generated clusters almost five times faster. As seen in the same figure, the pre-clustering makes the runtime curve linear even for the large binary sets by feeding the graph matching process with small binary groups containing more similar binaries. The lower half of the Figure 38, superimposed of the two upper graphs, displays the difference in runtimes. Lastly, since the system uses a database to store binary and clustering information, the I/O time is much higher than computation time of clustering algorithms. Consequently, this experiment, carried out on the Zeus malware set, also proves that our implementation of pre-clustering significantly improves the runtime performance of total clustering process.

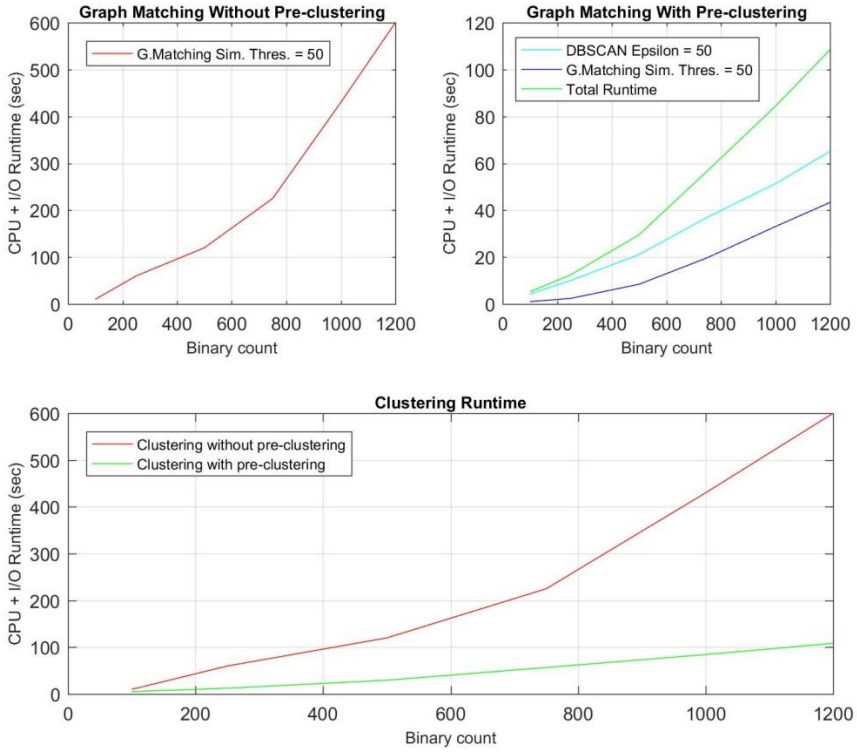


Figure 38: Clustering Runtime Performance

The experiments performed on the malware sets showed that using function call graphs of malware binaries in clustering reveals accurate clustering results. We also observed that our system improves the runtime performance of the time-consuming graph matching process which relies on pairwise function call graph comparisons.

6. CONCLUSION

In recent years, sophisticated mining techniques have been employed in malware analysis in order to handle large amounts of malware variants. Malware clustering approaches are among these analysis techniques. The purpose of malware clustering is to group malware samples based on their features and behaviors. Numerous clustering algorithms have been proposed to improve clustering accuracy and performance in many application domains including malware analysis.

Advances in data collection and sandbox tools have led to large amounts of malware samples being collected, giving rise to malware sets with a large number of attributes. Traditional signature-based clustering methods are not efficient in analyzing large malware sets with a high number of attributes. To deal with this type of datasets, appropriate clustering techniques compatible with the characteristics of malware binaries should be chosen. For this reason, many clustering approaches have been applied to malware clustering. In this study, we proposed a clustering approach based on subspace clustering and graph matching concepts in order to improve clustering accuracy and runtime performance. We designed and implemented a prototype system to observe the effects of our proposed method on malware clustering. Our system provides an interface for finding more specific clusters in large datasets in an efficient way.

Our system uses call relations of local functions and dlls extracted from malware binary samples as the similarity metric in its graph matching process. Hence, it provides a better clustering accuracy than signature-based clustering approaches. Using internal structure of a malware as a comparison parameter provides more accurate clustering results; however, it significantly increases the clustering runtime. To overcome the runtime issue, we employed a subspace clustering method to improve runtime performance of the expensive graph matching algorithms. The subspace clustering module of the system generates clusters based on the static features extracted from malware binary codes such as local function count, local function call count, dll count and dll call count, etc. Hence, malware variants that have similar features can be grouped prior to the graph matching process.

We run our system over a set of malware binaries that we obtained from two different web resources to observe and verify the accuracy and runtime performance of the system. The experiment results show that our method improves the runtime of the clustering process without degrading clustering accuracy. In other words, pre-clustering process, subspace clustering, makes the targeted binary set ready for the graph matching process with preserving true clustering information. We observed the effects of the algorithm parameters on clustering by running the system with different parameter values. Hence, we empirically observed the optimal parameter values which generate best clustering results.

6.1 Limitations

Since we employed supervised clustering algorithms in pre-clustering process, analysts using the proposed system must have information about the input parameters of the algorithms because these parameters have direct effect on clustering accuracy and runtime performance. These parameters are the epsilon and minPoint parameters of the DBSCAN algorithm and the redundancy index, minSize and dimensions of object space parameters of INSCY algorithm.

The proposed clustering system was proposed to process a large number of malware variants. There are different malware types such as viruses, worms, trojans, rootkits, ransomware, etc. Millions of new malware variants have been detected every year. Their count distribution may be different according to their types. Besides, the count of new emerging clusters may also vary depending on the malware type. Despite the large number of malware variants, the number of exploits are limited. The proposed system does not consider the number and types of exploits.

6.2 Future Work

In this study, we focused on clustering of collected malware samples based on their static binary features and internal binary structures. Our system generates clusters based on the features of a malware set given as an input to the system. It does not have a function of comparing a malware sample with an already clustered malware set. A classification module might be added to the system in order to classify a malware sample without running the whole clustering process.

We used the DBSCAN algorithm under the subspace clustering method. Different density-based clustering approaches such as k-means might be also implemented in order to compare clustering and runtime performances of density-based algorithms on malware clustering.

In the current implementation, we use a graph edit distance method to measure file similarity in the graph matching process. However, high runtime complexity of exact graph matching makes it less practical for large datasets. We might integrate inexact graph matching approaches to the system in order to gain additional runtime improvement.

REFERENCES

Abdullah, J., & Chanderan, N. (2017). Hierarchical Density-based Clustering of Malware Behaviour. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 9(2-10), 159-164.

Anderson, B. (2014). Integrating Multiple Data Views for Improved Malware Analysis. (Doctoral dissertation, The University of New Mexico Albuquerque).

Arefkhani, M., & Soryani, M. (2015). Malware clustering using image processing hashes. *In Machine Vision and Image Processing (MVIP), 2015 9th Iranian Conference* (pp. pp. 214-218). IEEE.

Assent, I., Krieger, R., Müller, E., & Seidl, T. (2007). DUSC: Dimensionality unbiased subspace clustering. *In Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference* (pp. pp. 409-414). IEEE.

Assent, I., Krieger, R., Müller, E., & Seidl, T. (2008). INSCY: Indexing subspace clusters with in-process-removal of redundancy. *In Data Mining, 2008. ICDM'08. Eighth IEEE International Conference* (pp. pp. 719-724). IEEE.

Awad, R. A. (2014). Automated clustering of malware variants based on structured control flow. (Doctoral dissertation, Universidad Politecnica Puerto Rico (Puerto Rico)).

Bengoetxea, E. (2002). The graph matching problem. *Doctoral Dissertation*.

Böhne, L. (2008). Pandora's bochs: Automatic unpacking of malware. University of Mannheim, 6.

Carletti, V. (2016). Exact and Inexact Methods for Graph Similarity in Structural Pattern Recognition. *PhD thesis*.

Cesare, S. (2010). Fast automated unpacking and classification of malware. (Doctoral dissertation, Faculty of Arts, Business, Informatics and Education, Central Queensland University).

Cesare, S., Xiang, Y., & Zhou, W. (2013). Malwise—an effective and efficient classification system for packed and polymorphic malware. *EEE Transactions on Computers*, p. 62(6).

Contagio. (2018, May 05). *Contagio Malware Dump*. Retrieved from Contagio Malware Dump: http://contagiodump.blogspot.com/

Dowd, C. (2014). Identifying Malware Using N-gram Clustering Metrics. (Doctoral dissertation, University of Maryland, Baltimore County).

Eagle, C. (2011). *The IDA Pro Book.* No Starch Press.

Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2), 6.

Ester, M., Kriegel, H. P., Sander, J., & Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. *Kdd*, Vol. 96, No. 34, pp. 226-231.

Fowlkes, E. B., & Mallows, C. L. (1983). A method for comparing two hierarchical clusterings. *Journal of the American statistical association*, 78(383), 553-569.

Gandotra, E., Bansal, D., & Sofat, S. (2014). Malware analysis and classification: A survey. *Journal of Information Security*, 5(02), 56.

Han, K. S., Kang, B., & Im, E. G. (2011). Malware classification using instruction frequencies. *In Proceedings of the 2011 ACM Symposium on Research in Applied Computation* (pp. pp. 298-300). ACM.

Hex-rays. (2018). *Hex-rays*. Retrieved from Hex-rays: https://www.hex-rays.com/products/ida/

Hu, X. (2011). Large Scale Malware Analysis, Detection and Signature Generation. (Doctoral dissertation, Computer Science and Engineering in The University of Michigan).

Hu, X., Chiueh, T. C., & Shin, K. G. (2009). Large-scale malware indexing using function-call graphs. *In Proceedings of the 16th ACM conference on Computer and communications security* (pp. pp. 611-620). ACM.

Hubert, L., & Arabie, P. (1985). Comparing partitions. Journal of classification, 2(1), 193-218.

Jaccard, P. (1902). Distribution comparée de la flore alpine dans quelques régions des Alpes occidentales et orientales. *Bulletin de la Murithienne*, (31), 81-92.

Kailing, K., Kriegel, H. P., & Kröger, P. (2007). DUSC: Dimensionality unbiased subspace clustering. *Seventh IEEE International Conference* (pp. pp. 409-414). IEEE.

Kang, M. G., Yin, H., Hanna, S., McCamant, S., & Song, D. (2009). Emulating emulation-resistant malware. *In Proceedings of the 1st ACM workshop on Virtual machine security* (pp. (pp. 11-22)). ACM.

Livi, L., & Rizzi, A. (2013). The graph matching problem. *Pattern Analysis and Applications*, 16(3), 253-283.

Mandiant. (2019, January 1). *Fireeye*. Retrieved from Mandiant Apt Report: https://www.fireeye.com/content/dam/fireeye-www/services/pdfs/mandiant-apt1-report.pdf

Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1), 32-38.

Rad, B. B., Masrom, M., & Ibrahim, S. (2012). Camouflage in malware: from encryption to metamorphism. *International Journal of Computer Science and Network Security*, 12(8), 74-83.

Rand, W. M. (1971). Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336), 846-850.

Ruohonen, K. (2013). *Graph theory*.

S, W., & D, W. (2012). Comparing Clusterings - An Overview . *Karlsruhe: Universität Karlsruhe, Fakultät für Informatik.*, 19.

Sanfeliu, A., & Fu, K. S. (1983). A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13, 353-362.

Sim, K., Gopalkrishnan, V., Zimek, A., & Cong, G. (2013). A survey on enhanced subspace clustering. *Data mining and knowledge discovery*, 26(2), 332-397.

Singh, N., & Khurmi, S. S. (2016). ByteFreq: Malware clustering using byte frequency. *Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), 2016 5th International Conference* (pp. pp. 333-337). IEEE.

Spizler, A. (2018). Clustering Analysis of Malware Binaries Using The Lempel-Ziv Jaccard Distance. (Master Thesis, Faculty of the Graduate School of the University of Maryland).

Tian, R., Batten, L. M., & Versteeg, S. C. (2008). Function length as a tool for malware classification. *In Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference* (pp. pp. 69-76). IEEE.

Tian, R., Batten, L., Islam, R., & Versteeg, S. (2009). An automated classification system based on the strings of trojan and virus families. *In Malicious and Unwanted Software (MALWARE), 2009 4th International Conference* (pp. pp. 23-30). IEEE.

Van Rijsbergen, C. J. (1979). *Information Retrieval.* Butterworth-Heinemann.

Virusshare. (2015, May 10). *Virus Share*. Retrieved from Virus Share: https://virusshare.com/

Yan, J., Yin, X. C., Lin, W., Deng, C., Zha, H., & Yang, X. (2016). A Short Survey of Recent Advances in Graph Matching. 167-174. 10.1145/2911996.2912035.

Zhong, Y., Yamaki, H., & Takakura, H. (2012). A malware classification method based on similarity of function structure. *In Applications and the Internet (SAINT), 2012 IEEE/IPSJ 12th International Symposium* (pp. pp. 256-261). IEEE.

APPENDICES

APPENDIX A

Family and Cluster Identifiers of APT Malware Samples

CLUSTER IDENTIFIER	FAMILY	SAMPLE COUNT
1	AURIGA	2
2	BANGAT	5
3	BISCUIT	5
4	BOUNCER	3
5	COOKIEBAG	7
6	GETMAIL	3
7	GOOGLES	5
8	GREENCAT	19
9	HACKFASE	3
10	MINIASP	3
11	NEWSREELS	7
12	SEASALT	2
13	STARSYPOUND	9
14	TABMSGSQL	6
15	TARSIP-ECLIPSE	7
16	TARSIP-MOON	6
17	AUSOV	2
18	BOLID	4
19	CLOVER	3
20	CSON	9
21	GREENCAT2	6
22	WEBC2-NEWSREELS	4
23	HEAD	10
24	QBP	3
25	RAVE	6
26	UGX	5
27	Y21K	3
28	YAHOO	8

29	WEBC2-GREENCAT	6
30	BANGAT2	3
31	NEWSREELS2	3
32	WEBC2-YAHOO	3

TEZ İZİN FORMU / THESIS PERMISSION FORM

ENSTİTÜ / INSTITUTE

Fen Bilimleri Enstitüsü / Graduate School of Natural and Applied Sciences ☐

Sosyal Bilimler Enstitüsü / Graduate School of Social Sciences ☐

Uygulamalı Matematik Enstitüsü / Graduate School of Applied Mathematics ☐

Enformatik Enstitüsü / Graduate School of Informatics ☐

Deniz Bilimleri Enstitüsü / Graduate School of Marine Sciences ☐

YAZARIN / AUTHOR

Soyadı / Surname : .....
Adı / Name : .....
Bölümü / Department : .....

TEZİN ADI / TITLE OF THE THESIS (**İngilizce** / English) : .....
.....
.....
.....
.....

TEZİN TÜRÜ / DEGREE**:** **Yüksek Lisans** / Master ☐ **Doktora** / PhD ☐

1. **Tezin tamamı dünya çapında erişime açılacaktır. /** Release the entire work immediately for access worldwide. ☐
2. **Tez iki yıl süreyle erişime kapalı olacaktır.** / Secure the entire work for patent and/or proprietary purposes for a period of **two year. *** ☐
3. **Tez altı ay süreyle erişime kapalı olacaktır.** / Secure the entire work for period of **six months**. ***** ☐

***** *Enstitü Yönetim Kurulu Kararının basılı kopyası tezle birlikte kütüphaneye teslim edilecektir.*
A copy of the Decision of the Institute Administrative Committee will be delivered to the library together with the printed thesis.

Yazarın imzası / Signature ..... **Tarih** / Date .....