UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

NordSecMob

Saad Usman Khan

Identification of malicious behavior patterns for software

Master's Thesis (30 ECTS)

Supervisor:	Dr. Dominique Unruh, UT
Co-supervisor:	Dr. Colin Boyd, NTNU
Co-supervisor:	Dr. Felix Leder, Blue Coat Norway

Tartu, 2014

Identification of malicious behavior patterns for software

Abstract:

Over the years malware has increased in number and became increasingly harmful. Traditionally, anti-virus suites are used to protect the computers from various forms of malware. In recent years a new technique called “behavior based malware analysis” has become common which overcomes some shortcomings of traditional anti-virus suites. Just like anti-virus suites require signatures, behavior analysis systems require pattern groups for malware identification. This thesis presents the design and implementation of a Malware Pattern Generator (MPG). MPG is built to automatically generate behavior based pattern groups from a given malicious dataset. MPG uses hierarchical clustering to find similarities between malware and extracts the similarities to generate pattern groups. Three variants of MPG are developed during the work on this thesis and the results of their evaluation against malicious datasets are presented.

Keywords:

MPG, Behavior based malware analysis

Tarkvara pahatahtlike käitumismustrite tuvastamine

Lühikokkuvõte:

Aastate jooksul on pahavara hulk kasvanud ja pahavara on muutunud järjest kahjulikumaks. Traditsioonilised viirusetõrje tööriistad kaitsevad arvuteid erinevate pahavara vormide eest. Viimastel aastatel on aga muutunud populaarseks uus tehnika "käitumispõhine pahavara analüüs", mis ületab mõned traditsiooniliste viirusetõrjeprogrammide puudujäägid. Nagu viirusetõrjed vajavad pahavara signatuure, kasutavad käitumispõhised tööriistad käitumismustrite gruppe, et pahavara tuvastada. Käesolev töö tutvustab tööriista Malware Pattern Generator (MPG ehk pahavara mustrite generaator) tööpõhimõtteid ja implementatsiooni. Tööriist MPG genereerib automaatselt käitumispõhiseid mustrite gruppe kasutades selleks hulka teadaolevat pahavara. MPG kasutab hierarhilist klasterdamist, et leida sarnasusi erinevate pahavarade vahel ja ekstraheerib need sarnasused, et luua mustrite gruppe. Käesolev töö kirjeldab kolme erinevat MPG versiooni ja võrdleb nende tulemusi.

Võtmesõnad:

MPG, käitumispõhine pahavara analüüs

Preface

This thesis is written during spring semester 2014 at Blue Coat Norway AS as part of Erasmus Mundus Joint Masters in Security and Mobile computing (NordSecMob). The thesis is presented at Department of Telematics (ITEM) at Norwegian University of Science and Technology (NTNU) and Institute of Computer Science at University of Tartu (UT) as thesis for Master of Science in Security and Mobile Computing and Master of Science in Engineering (Computer Science) respectively.

The thesis work was conducted by the author in collaboration with Blue Coat Norway under the supervision and guidance of Dr. Colin Boyd (NTNU), Dr. Felix Leder (Blue Coat) and Dr. Dominique Unruh (UT). This thesis is also supported by the Estonian Research Council through the grant IUT2-1.

I would like to take this opportunity to thank my supervisors who helped me in each and every step of the thesis work. Their guidance and support helped in improving the quality of work. I would also like to express my gratitude towards Lukas Rist and Trygve Brox who helped me in finding datasets for evaluation of work and other colleagues at Blue Coat Norway AS who helped and guided me in solving technical problems encountered. Coordinators of NordSecMob Aino Lyytikäinen, Mona Nordaune and Natali Belinska helped in coordinating between universities and helped with other formalities, a big thanks to them. I am also very grateful to Pille Pullonen who helped me in writing a thesis abstract in Estonian.

Finally, I would like to thank my brother Dr. Azhar Khan who reviewed thesis text, my parents and rest of family who supported me during the thesis and throughout the study in Europe.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

API Application Programming Interface.

cmd Command Prompt.

CnC Command and Control.

company Blue Coat Norway AS.

CPU Central Processing Unit.

DLL Dynamic Linked Library.

DNS Domain Name Server.

FN False Negative.

FP False Positive.

HTTP Hypertext Transfer Protocol.

ID Identifier.

IDS Intrusion Detection System.

IP Internet Protocol.

LSH Locality Sensitive Hashing.

MAA Malware Analysis Appliance.

MPG Malware Pattern Generator.

NCD Normalized Compression Distance.

NTNU Norwegian University of Science and Technology.

OS Operating System.

PC Personal Computer.

PDF Portable Document Format.

PE Portable Executable.

SPR Sample to Pattern Ratio.

SVM Support Vector Machines.

TCP Transmission Control Protocol.

UAC User Account Control.

URL Universal Resource Locater.

URN Universal Resource Name.

USB Universal Serial Bus.

USD US Dollar.

UT University of Tartu.

VM Virtual Machine.

Chapter 1 Introduction

Cybercrime and other forms of malicious activities over the Internet are costing the world from 300 billion to over a trillion dollars yearly [MC13]. There are multiple anti-virus software suites available which can be used to fight against malware. However, millions of new malware samples are detected every month by security companies [McA13] which makes the task of keeping the anti-virus signature database up to date a difficult and critical task. Malware now uses polymorphism and data obfuscation to prevent their detection against traditional anti-viruses [YY10]. Due to the limitations of the anti-virus software, a new state of the art technique for analyzing malware called "behavior based malware analysis" has become widely known.

Behavior based malware analysis not only focuses on the contents of the malicious file but also analyzes the behavior of the malware samples under analysis. During this process, a malware sample is executed in a virtualized environment where all the system activities performed by the malware are recorded. These activities can then be analyzed by a human to understand the behavior of malware. This understanding of malware behavior can be used in prevention against the threat. This technique was initially advantageous over the simple static analysis because of deeper insight but after the maturity of this field, scalability of analysis has become the next big challenge.

State of the art dynamic analysis systems such as Blue Coat's MAA [Coa14] provide the capability to analyze thousands of new malware samples every day in virtualized environments. However, it is nearly impossible to manually read and understand this many analysis reports. Some automated system needs to be developed which can analyze the reports and extract important summarized data.

1.1 Problem

One of the uses of behavior based analysis is to provide malware detection for the samples which are difficult to detect using static methods. As mentioned above, obfuscation techniques make the static analysis harder. Furthermore, malware writers sometimes develop slightly different variants of same malware so it is not possible to create a single static signature for them all. However, the behavior of all the variants is the same because it is the same malware with variations only to prevent the detection.

Behavior based analysis can be very useful in both of these cases. Multiple variants of the same malware can be identified because their behavior is still consistent across variants. Furthermore, obfuscated or packed binaries can also be detected as they must unpack or de-obfuscate themselves before execution. In either of these cases the behavior of the malware remains the same.

Dynamic behavior based detection relies on pattern groups: an entity similar to signatures in traditional anti-virus detection. Pattern groups are unique behaviors of malware which can be used to identify it. Manually finding behavior pattern groups which can identify malware and only malware is a tedious task. First, a malware sample is analyzed using a behavior based analysis system to generate an analysis report. This report is then read by a malware analyst to understand the behaviors. Based on knowledge of the analyst and his understanding of malware, such behaviors are found which can identify the malware while keeping the chances of detecting a non-malicious binary low. Because of high rates of new malware development, it became more and more difficult to keep the pattern groups database up to date to detect latest malware.

This thesis addresses this difficulty and studies the possibility of designing and implementing a system which can produce the behavior pattern groups for malware automatically. Successful design of such a system can help greatly in keeping up dynamic analysis with the rate of malware discovery.

1.2 Goals

The central goal of this thesis is to create a software system which takes a set of malicious files as input and produces a set of pattern groups which can be used to identify those malicious files. The system should be completely automated which requires no human intervention or intelligence in reaching from malware samples to the pattern groups. Furthermore, the system is designed to produce pattern groups satisfying the following quality criteria in order of their importance:

1. Zero False Positives: Any behavior pattern group which identifies malicious files as well as clean files is not usable in any practical scenarios. Therefore, the system must produce pattern groups which detect some malicious files but do not falsely detect any clean files.

2. Low False Negatives: In theory, it is very difficult to find a behavior which would help in classifying all the malware however, it seems feasible to find behavior pattern groups which cover a certain set of similar malware. So, the system should be able to produce pattern groups which detect as many samples from dataset as possible.

3. Sample to Pattern Ratio (SPR): Sample to pattern ratio is given by dividing the number of samples by the number of pattern groups used to detect them. SPR represents the approximate number of samples each generated pattern group detects. It is desirable for pattern groups to consist of important behaviors of malware which are shared by other similar malware. Therefore, sample to pattern ratio should be high. If a pattern group detects many samples, it is more desirable over pattern groups which detect only a few.

The quality criteria mentioned above are mentioned in the order of their importance. Avoiding the generation of pattern groups which cause False Positives (FPs) has the first priority followed by coverage of the dataset. And finally, the number of pattern groups created is kept to the minimum. The system should then be evaluated on a dataset of real malware samples.

1.3 Preview of results

A software was developed to generate malicious behavior pattern groups from any dataset of malware. The software was capable of producing behavior pattern groups with detection rate of almost 100% for the dataset that was used to generate pattern groups. Furthermore, the behavior pattern groups were able to detect approximately 60% of a three times bigger dataset which did not participate in pattern group generation. All the pattern groups were false positive free on the basis of a reasonable dataset of non-malicious files. Depending on the dataset, SPR in the range of 2.7 to 128 was observed.

1.4 Structure of Thesis

After this introduction chapter, chapter 2 summarizes the previous work done related to behavior based malware pattern generation. The work done related to static and dynamic automated signature generation is presented. Furthermore, some work on clustering the malware based on dynamic events is cited because of its importance to the solution presented in this thesis.

Chapter 3 provides the background information on malware and explains the behaviors of malicious software in detail. The definition of malware and some examples of real world malware are presented. Furthermore, a detailed work-flow of behavior based malware detection is provided. In section 3.4 detailed description of behaviors of the software in Microsoft Windows is provided. This section provides an explanation of each behavior along with its properties and its importance towards detection of malware.

The system which is designed to generate automatic behavior pattern groups relies on clustering the malware. Therefore, a separate chapter: chapter 4 explains the clustering of malware based on dynamic events. Chapter 4 explains hierarchical clustering and partition based clustering and concludes that hierarchical clustering is a more usable alternative for the problem of pattern group generation.

Chapter 2, chapter 3 and chapter 4 provide the background information required to design the system while chapter 5 presents the contribution by the author.

Chapter 5 presents Malware Pattern Generator (MPG) which is developed to generate the pattern groups for the malware. Architecture of the system along with different components is explained in detail. Implementation details of some components are also presented where required.

Chapter 6 presents the evaluation of the system against datasets consisting of real world malware samples. Evaluation results are also explained and discussed in chapter 6. Chapter 7 contains the work that can be done in future to improve MPG. Finally, chapter 8 concludes the thesis.

Examples of malware events and a pattern group are included in appendices. A sample set of events produced by a malware are presented in appendix A and a sample pattern group generated by MPG is presented in appendix B.

Chapter 2
Related Previous Work

Variety of malware existed in world for many decades now. It all started when two brothers from Lahore a provincial capital of Pakistan created the first Personal Computer (PC) virus. That virus was not intended to cause any harm, rather it focused on proving that computers are not secure [Mil13]. From that time onwards the number of viruses kept increasing and many new variants kept arising. Also, many more viruses were created afterwards which were intended to cause harm to people for financial gains.

With the increase in number and sophistication of malware, many anti-malware studies have taken place. And more importantly, many commercial anti-malware software suites were developed. The research on viruses started even before the first PC virus was developed. Fred Cohen wrote an academic paper on computer viruses in 1987 [Coh87]. In that paper, he discussed the possible scenario of a virus attack and proposed different guidelines when designing systems to make them resistant to virus attacks. He also demonstrated how easy it is to create viruses by providing the source code of an example virus.

In late 1980s and early 1990s commercial anti-virus software such as Norton Antivirus, McAfee and many others came on surface. In the beginning, all the anti-virus software used signature generation as a method of malware detection.

2.1 Signature Generation

Traditionally, signatures usable to detect malware were generated by human malware analysts after reverse engineering malware. Typically this is done by using disassembling software such as OllyDbg [Oll14] to convert the executable binary into assembly language instructions. These instructions can then be executed line by line to understand the behavior of the malware. Typically, malware is analyzed in an isolated physical or virtual machine to protect the machine and the network from the malicious effects of malware. By utilizing the understanding gained by reverse engineering, a binary string corresponding to some critical functionality of malware is identified. This binary string then acts as a signature to identify the malware.

Since its a tedious task and the number of new malware is always increasing, this strategy has proven to be very slow. Naturally many people researched into generating the malware signatures automatically.

2.1.1 Automated Static Signature Generation

In 1990s several patents were filed which contained automatic virus signature generation in one way or another. Some of them were specifically focused on generating signatures for anti-virus softwares while others generated signatures for network anomaly detection.

In 1995 a patent was published which described a method to automatically extract signatures of computer viruses [Kep95]. Another patent explains an automatic immune system for computer networks. This work consists of automatically finding anomalies over the network and then finding their signatures [ACKW95].

Karin Ask in her thesis [Ask06] proposed a system to generate automatic malware signatures. The strategy used there was to cluster the malware into families. A clustering algorithm was not invented by her, rather she used predefined clusters. After receiving a cluster she compared the binary executable of the malware samples within the same cluster to find out the common parts of the binaries. And then she generated signatures out of the strings shared by malware. She stated in her thesis that it is indeed possible to generate signatures which can identify malware. Moreover she received good detection rate on her signatures.

Symantec used to use a hash based signature scheme for identifying malware. They used the technique of generating hashes of the malware executables. These hashes were then distributed as a signature with their anti-virus software. On the client side, user files were scanned, their hashes were computed and those hashes were then compared with hashes in malware hash database. If a match occurred, the file was considered malicious. The biggest advantage of this scheme is that if a malware hash is added to the database malware gets detected with 100 percent certainty. And since hash functions are collision resistant, chances of FPs are quite less. The shortcoming of this approach was that they needed to create a hash of each file and the database of the hashes was too large which required too much space and processing to match. However later [GSHC09] Symantec research laboratories published a technical report explaining a system which replaces their hash based static signature system with a different system where signatures consist of byte strings from the malware. This system also generates automated signatures based on bytes of the executables.

2.1.2 Dynamic Behavior based signature Generation

Apart from generating signatures for the executable, several works have been done to generate signatures based on the network traffic generated by malware. Since the traffic is one of the behavior of the malware, it can be considered a behavior based signature.

[SEVS04] discusses on how to generate signatures for worms. This research is conducted based on analyzing the network traffic generated by the worms. Similar other works also exist [KK04], [NKS05], [TC05], [SK03].

A similar work [KC04] proposed a system to generate automatic signatures from network traffic of malware for Intrusion Detection Systems (IDSs). In another research paper a honeypot based worm detection solution is presented. A honeypot is deployed which analyzes the network traffic to capture suspicious traffic. This network traffic is then used to generate the worm signatures [PB05].

Li et al. in [LWLR08] worked on generating automatic signatures for malware based on their dynamic behavior. Their system is based on a security policy which mentions which activities are suspicious and which activities are benign. The security policy consists of things such as "hooking the keyboard is forbidden". If a process performs activities which violate the security policy, it is considered malicious and signatures are generated for instances of such activities. So, although it uses dynamic events, still a static database of security policy is required to find and make signatures of malware.

2.2 Clustering

Clustering of malware helps in understanding what family the malware belongs to. Generally, when a malware is released, it's many variants are also released to make it difficult for security companies to produce detection. These malware variants are generally performing similar activities while changing those properties of malware which are used for static detection. Clustering also helps in identifying the malware which are similar to each other but may or may not have been written by same authors.

Before the popularity of behavior based analysis, malware clustering was also performed using static properties of malware. Some examples of these properties are:

- contents of malicious executable;
- function calls;
- dependencies on external components.

In recent years much work has been done in clustering malware based on its behavior. Different studies have used different methods for calculating distance between samples followed by using standard clustering algorithms to cluster malware.

[BOA^{+}07] is a basic research paper in this field. In this research the author uses Normalized Compression Distance (NCD) to find the distance between behaviors of two malware samples. This distance is then used to identify the samples which are closest to each other from the whole sample set. [BCH^{+}09] shows that the previous clustering scheme can be improved considerably by using a different distance function. Authors in [BCH^{+}09] used Locality Sensitive Hashing (LSH) along with Jaccard's index to measure the distance between samples.

Another behavior based malware clustering approach used only Hypertext Transfer Protocol (HTTP) traffic generated by malware to cluster them. The HTTP traffic is decomposed into subcomponents such as Universal Resource Locater (URL) and Universal Resource Name (URN) etc and then the distance between samples is measured. To measure the distance, a combination of various distance functions including Levenstein and Euclidean distance is used [PLF10].

The basic building block of any successful clustering algorithm is to identify the similarities between samples. Since similarities between malware samples are already calculated, clustering can be extended to generate signatures based on similarities between samples of same cluster.

This thesis will therefore re-use techniques explained in this section to cluster malware and then identify the signatures from them.

2.3 Signature Extraction

Machine learning is also a well studied research area. It has been used to solve many problems. The problem of malware signature generation can also be approached using machine learning.

[RHW^{+}08] explains a Support Vector Machines (SVM) based algorithm to cluster malware based on the dynamic properties. The work not only differentiates the benign files from malicious files but also clusters the malicious files into different clusters.

Generally, it is difficult to understand how any machine learning algorithm classifies a certain malware instance. The reason for this is that in a very high dimensional space the rules which classify a sample are too complex to be written in a formal way. However, some work has been done in this field to find out the exact rules which classify a file into a certain group.

[BD04] explains a strategy which can be used to extract learned rules out of SVMs based system.

2.4 Summary

This chapter gave a brief overview of other work which is related to this thesis. Most anti-virus software suites use signatures for malware detection. Traditionally, signatures were generated by human malware analysts based on static properties of the malware. However, with increase in the number of new malware samples much research work took place to automate the manual signature generation. Section 2.1.1 explains the research done in the field of automatically generating malware signatures. In recent years more focus has been put on detecting malware based on its behavior. Section 2.1.2 explains some research work in this field.

Malware clustering is useful for seeing relationship between different malware samples. In this thesis, clustering is used as one of intermediate steps to generate behavior based pattern groups as discussed in chapter 4. Much work has been done to cluster the malware based on its dynamic behavior. This was discussed in section 2.2.

Finally, SVM can be used to differentiate malware from benign files. Some work in this field is discussed in section 2.3.

Chapter 3
Malware and Software Behavior

A computer consists of physical hardware and software components which are installed on top of hardware. The most important software on computers is an Operating System (OS). On top of the OS further software can be installed by users. The software installed on a computer can be both benign and malicious.

This chapter will start with a formal definition of software behavior followed by in-depth explanation of malware in section 3.1. Section 3.1 will explain different types of behaviors malware may produce and how these behaviors can be used to detect the malware. After giving the brief description of malware behaviors, section 3.2 provides the example of some real malware found in the wild. Rest of the chapter is focused on detection of malware based on behaviors. A typical work-flow of behavior based malware detection is provided in section 3.3 followed by a detailed description of different events a software may produce. Different properties of the events based on their relevance for malware detection are also discussed.

Suppose there is a software *soft1* installed on a computer but not yet executed. This state of the system is named *State1*. After execution of *soft1* the state of the system will change to *State2*. The difference between *State1* and *State2* represents the changes which are done in the system by *soft1*. This difference is named *Diff1* as shown in Equation 3.1.

$$\mathit{Diff1} = \mathit{State2} - \mathit{State1} \tag{3.1}$$

Apart from the difference in the state there are many small changes that happen after *State1* but they disappear before system reaches *State2*. These changes are usually temporary system changes such as creation of a file followed by its deletion or a process launch followed by its termination etc. All such changes are named *Temp* for scope of this thesis. This leads us to Equation 3.2.

$$\mathit{Changes} = \mathit{Diff1} + \mathit{Temp} \tag{3.2}$$

Later throughout this chapter and rest of the text, words such as behaviors, events and system changes will be used to refer to *Changes*. The main topic which will be discussed in this chapter is what features in *Changes* can help us identify whether *soft1* was malicious or benign.

All the software when run on an OS exhibits the behavior mentioned above. However, for the scope of this thesis more focus will be put on system changes caused by malware.

3.1 Malware

The term malware stands for malicious software. Malicious software is the software which causes some intentional harm to the computer where it is run. Additionally, a malware may cause harm to the owner of computer by stealing some valuable information belonging to him or, it may use the resources of computer without consent of the owner of the computer. The harm is caused mostly without involvement of the user in the process. However, sometimes users are tricked into performing malicious activities on their system. This technique is called social engineering: where users are led to believe that they are performing something which is useful for them but in reality they are helping the malware to execute on their system.

Malware is a generic term which encompasses a wide range of malicious software. Some malicious software is targeting the computers of the people with the goal to recruit them in a botnet while others are targeted at specific corporations to extract their intellectual property. The rest of this section will explain different malicious behaviors found in the malware. There are many labels associated with malware such as viruses, worms and trojan horses etc. However, such classifications are not well defined. A malware may belong to several categories and one category may have very diverse malware in it.

The initial malware samples were called viruses. The term virus is more widely known compared to malware. A significant portion of malware self replicates. The basic property of self replicating malware is that when its executed it normally performs two activities. First, it replicates itself to other infect-able locations and then it performs the malicious activities it is supposed to perform. The part of the code which performs the malicious activities is called payload. However, not all replicating malware samples have a payload. Some malware samples may just replicate to consume the system resources without specifically performing any other harm. Common locations where such samples stores their copies are: programs running on the system, other locations of file system, boot sectors and removable drives. A category of malware when run injects itself into other running programs to perform its malicious activities. A more formal definition of viruses can be found in [Coh].

For many years virus replication through removable hard drives was very common. It was largely driven by the fact that users were using removable drives to transfer data between computers. Using external storage medium propagation malware can infect off-line computers.

Some malware samples replicate themselves by sending their copies over the network. Such malware not only causes usual harm but also consumes network bandwidth which the user is paying for.

Additionally it has been observed that some malware samples hide behind other legitimate software to infiltrate computers. That software looks either useful or interesting and users willingly install it on their system. After the software is installed, the malware performs it's malicious activities.

Majority of malware do not have administrator access to the system. However some malware samples manage to acquire the administrator or root access to the system. Because of this access, they can perform many activities which a standard malware sample cannot perform. This includes all the functions which requires root access such as installing new administrator software, modifying system files, interrupting and monitoring standard computer processes such as network communication.

Some malware has ability to circumvent the detection. Since the anti-virus software runs with root access, it is difficult for malware which does not have root access to interfere with it but it is not impossible. However, malware which has root access can stop the antivirus altogether or tamper with it so it doesn't identify the malware. Malware samples which run with root access are called rootkits. In recent years much work has been done on rootkit detection. More information about this can be found in this survey [KPL+12].

The text above explained different ways malware spreads, survives and infects its victims. After successfully infecting a system, it may perform many different malicious acts which are harmful to the owner of the computer. Some of such activities are explained below.

A malware may keep a log of the keys users press on their computers. Primary target of such malware is to steal the credit card numbers and passwords of different high value on-line accounts such as bank accounts. However, keylogging malware can also be used to intercept everything a user types on a computer. Keyloggers are a typical example of how a very secure system can be breached by focusing on the weakest link. All the important communication over the internet takes place over encrypted connections. As encryption is a very well studied topic and it is very difficult to break, so instead of breaking that, keyloggers focus on stealing the information before its encrypted.

A malware sample may be targeted at a specific user to spy on him, or can be used to spy on a number of computers. Such spying malware collects the list of programs people use and the websites they visit. The collected information is then sent to Command and Control (CnC) servers. Those servers can then use that information differently based on the victim. For most people it is used to serve ads to the users based on their internet browsing habits. Some high profile people may be blackmailed based on their system activities.

Some malware samples take some system resource hostage and promise the user that it will release the hostage if a “ransom” is paid. This type of malware is referred to as ransomware. One such type of ransomware is an encrypter. When it manages to run on a system it encrypts the user files on that system with a strong cipher. After encryption is completed, the user is told that all its files have been encrypted and the user must pay the attacker some money to get the decryption key for decryption. Once the user pays he gets the decryption key to decrypt the data. Apart from the ransomware which relies on encryption, there is other ransomware which forces the users to pay ransom through other methods. One such method is to display pornographic images on the user’s screen and tell them that if they pay some money the pornographic images will be stopped from displaying.

Some malware samples do not perform any malicious activity themselves, rather they just download new malware and execute them on the system.

3.2 Malware Examples

The behaviors of malware which have been explained above are in no way complete. There are many more types of malware in the wild. And the trend keeps changing. Every now and then new techniques for creating new types of malware are discovered. The reason why the field of security is so challenging is that malware writers are very innovative and new techniques are developed very often. In this section two examples of real malware will be given to give a feel of what capabilities real advanced malware has. The two examples explained below are chosen based on their importance. Flame described in section 3.2.1 is one of most sophisticated malware ever found and went undetected for many years. ZeuS explained in section 3.2.2 is a famous banking malware which is responsible for loss of millions of dollars.

3.2.1 Flame

Flame is one of very famous Microsoft Windows malware from 2012. It was first analyzed in this report [BBFP12]. The reason for fame of this malware was that it went undetected for many years. Some people believe it first appeared in 2007 and went on undetected for 5 years. One of the reasons behind the fact that it remained hidden for so long was that it was very targeted and it was not widely spread. In 2012 when it was detected only 1000 computers were infected worldwide. It was specifically targeted at government and educational institutions in Middle East.

Flame was technologically very sophisticated. It was able be spread to other computers using Universal Serial Bus (USB) flash drives as well as using the local network. It could also trick the system to believe that it is Microsoft Windows update software which resulted in system installing it with administrator access.

It had a very advanced CnC server infrastructure. The servers were located geographically in many different countries. Malware used to send all gathered information to one of the CnC servers and was capable of receiving instructions from its CnC servers and act accordingly. One of the instructions it could receive from the server was “kill command”. If the malware received this command it could delete itself from the infected system without leaving a trace. After it was detected in 2012, a kill command was sent to all infected machines to remove the malware.

Apart from all the infrastructure it had in place for survival, the main payload of Flame was used to gather intelligence. It included taking screenshots of the system periodically, keylogging the keyboard activity and storing and monitoring the network traffic. It was also capable of reading all the user files, recording audio from microphone and communicating over both wired and wireless network. One of the techniques it used was to make the infected computer a bluetooth hotspot to connect to the bluetooth enabled devices. The user contacts were then fetched from the bluetooth connected devices such as smart-phones and sent to CnC servers. Another interesting capability was the ability to record Skype calls which took place on the infected machine [BBFP12].

To conclude, the capability of the malware is as big as the imagination of the malware programmer. If anything is possible it is likely exploited in some malware already or is likely to be exploited in future.

3.2.2 ZeuS

ZeuS is a Trojan Horse with root-kit functionality. It is mainly used to steal credentials of the important user accounts such as to steal the banking credentials. Other form of credentials could also be stolen using ZeuS such as credentials of Facebook and Twitter. After stealing the credentials, ZeuS sends them to one of its CnC servers.

ZeuS mainly relies on having multiple slightly modified copies of itself. There are many variants of ZeuS in the wild and not all anti-viruses detect all variations of

ZeuS. It is thought that Zeus has infected over 3 million machines and has stolen over 70 million US Dollars (USDs).

More information on ZeuS can be found in [Inc10]

3.3 Behavior based Malware Detection

This section provides brief description of typical workflow of behavior based malware detection.

A typical behavior based malware system consists of two main components, a malware execution engine and a pattern matching engine. In this section, the word pattern will be used to refer to dynamic behavior based patterns rather than a static string based signature.

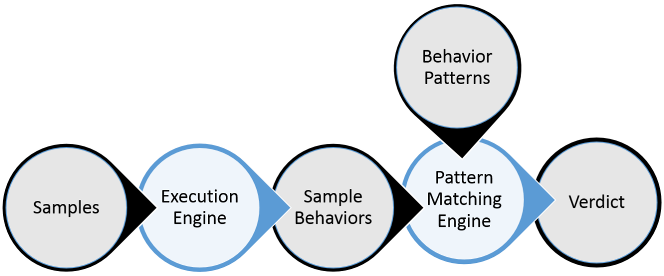

Figure 3.1: Work flow of malware detection based on it's dynamic behavior

In Figure 3.1 dark circles represent the data while light circles represent the processing modules.

Initially a set of files are obtained which need to be analyzed. These files can be collected in a number of different ways. A typical example is to scan the whole computer of a user and thus all the files present in the computer will be considered as samples and will be submitted to the execution engine. Execution engine then processes the samples to generate the behaviors of that sample in the form of events. These behaviors are then matched against patterns to detect whether the sample was malicious or benign. The final verdict is traditionally boolean (malicious or benign). However, more detailed verdict may also be issued such as the sample is x% likely to be malicious.

3.3.1 Sample Execution Module

The sample execution engine runs the given sample in a virtualized environment. Traditionally, it is a Virtual Machine (VM) with some OS installed on it. The Engine executes the malware for a predefined period of time in the OS. In the meanwhile every activity that sample performs in the OS is noted down. After the sample has finished execution or a set amount of time has elapsed, the execution of malware is stopped. Different types of events which can take place and their importance for malware detection is discussed in detail in Section 3.4.

The events which occurred in the system are the output of this module and they are then passed on to the Pattern Matching Engine.

3.3.2 Pattern Matching Module

The pattern matching module is responsible for matching the patterns in the pattern database to the events produced by the sample. The pattern database contains the behavior based patterns of malware. Such patterns are also called indicators of compromise. If a pattern matches a sample, the sample is declared malicious. A format of events and the patterns has to be agreed upon to make sure that patterns are consistent with the behaviors generated with sample execution module.

All the components shown in Figure 3.1 except the patterns database are taken as is. The objective of this thesis is to generate a pattern database automatically which would result in declaring malicious samples as malicious without causing any FPs.

3.4 Software Behaviors

The behavior of software can be categorized in different sections based on underlying OSs. Since the architecture of different OSs is quite different, the behaviors generated are also different and are usually not comparable to each other. Traditionally, majority of malware has been designed for Microsoft Windows and most of the research data sets contain the malware which only executes on Microsoft Windows. So, for the scope of this thesis only the malware for Microsoft Windows will be discussed.

There are many software applications both commercially and freely available which can record the system changes produced by a certain software. Cuckoo [Cuc14] is one such software. A web based application called Anubis [Anu14] also exists where samples can be submitted to retrieve their behaviors.

Depending on the software used to generate the behaviors, they might differ slightly. The differences between different behavior generating systems is not important for the scope of this thesis and will not be discussed here. Only the most important type of events which are common among most such systems will be discussed.

Malware comes in many forms and shapes. The most common form is Microsoft Windows Portable Executable (PE) files. However certain other forms of malware can also be found in wild such as:

- Portable Document Format (PDF) Files
- Microsoft Office files
- Various websites which infect the users upon visiting them using different methods which may work on only some browsers

However, the common thing between all these forms of malware is that they produce certain events when they are run inside a system. Since, we are only considering Microsoft Windows so such files when run in Microsoft Windows produce Windows events such as file creation, file deletion etc. Most important events which can be useful for malware detection are listed below:

1. File Events
2. Network Events
3. Service Events
4. Process Events
5. Registry Events

In this section, these events will be explained in more detail and the specific subsections of these events which can be used for pattern generation will be explained.

Each of different types of events mentioned above can have multiple specific events. Such as file system events may be divided into file create events, file delete events and so on. And each of the specific event can have multiple properties which are basic unit of a malware behavior. An example of such property would be the name of the file which was created. So, the knowledge that some file has been created (boolean) is useful but more importantly the exact name of the file which has been created is more valuable.

3.4.1 File Events

Files have a central position in a Windows system. All the information stored on the hard drive is in the form of files which includes all the code which is used to run Windows along with the user generated information. There are many file events which are important for malware detection. Most important of these are explained below.

File Create

This event tells if a file has been created. When a file is created in Windows, a unique handle is assigned to it [Mic14, aa363874]. The unique handle is very useful for operating system to keep track of the file creation however it gets generated dynamically so every time the same malware is run on the system, system may return a new handle therefore it is not useful for pattern generation. Along with knowledge that a file has been created, generally the name of the file, location of the file, mode and attributes make up this event. The mode of the file can be for example read only. The attributes can be, such as whether it should be compressed or not etc. Apart from these properties the file name itself is consistent for some malware while random for others. The random name is effective for avoiding the detection based on name and some malware samples use it. However, many malware samples still use consistent file names which can be used for detection.

File Delete

This event contains the information about file deletion. The name of the deleted file is useful from this section.

File Open

Since most of critical information stored in a computer is stored in the form of files. So, opening a file is an important event which can be helpful in pattern generation. However, some malware samples open many files when they are scanning directories like drivers directory etc. In that case the noise in file open is more than useful information. None the less, file open events are important. Typically the path of the file opened and the mode it was opened in are important characteristics of the event. The list of different modes and their explanation can be found at [Mic14, aa363874].

File Read

A file open event takes place before a file can be read but it is possible for a file to be opened but not actually read. So, this event can help in differentiating the files which are opened but not read. A file read event consists of path of the file, the offset at which the file was read and the size of the data which was read [Mic14, aa365467].

File Seek

In Windows Application Programming Interface (API) a seek function [Mic14, aa365227] is present which can be used to skip some bytes of a file to a certain offset. Then the next bytes can be read from that location. These events are not very common but can identify behaviors of malware who want to seek a certain amount of bytes of a certain file to read some specific information.

File Write

File writes are very common by both legitimate and malicious software. Most malware store their intermediate data in files with some specific names and in specific paths. This information can be used to identify malware. A call to a WriteFile function [Mic14, aa365747] provides us with data that was written, name of the output file, size of written data and the offset at which the data was written. The file name specifically is important because some viruses create files having consistent names such as *virusname.dat* which can be very useful for their detection. Other viruses use well known names such as *skype.exe* [Vir12]. But even in this case the path of the file is different than the legitimate file. So, that change in path is useful for detection based on this. The third case is the malware samples which generate a random string for a filename. This makes the file name property of this event useless for detection because if same malware is run twice it will just generate a different file name. However, file path, size and contents of the file may still be the same.

3.4.2 Network Events

In the present age of connectivity, most machines are connected to Internet or other local area networks. So, naturally malware takes advantage of the network connectivity and uses it for various reasons such as uploading stolen information to the CnC servers. Spreading the copies of same malware to other machines for infection and so on. Most important of network events are explained in depth below.

Process to network binding

To start communication over the network, typically a process has to create a socket and then communicate through that socket. This is achieved by calling the bind function [Mic14, ms737550]. In this event a socket handle, local address of the machine along with the port used are the properties. However, all of these values are not reproducible because socket handle is generated by operating system dynamically. The host Internet Protocol (IP) address depends on the host machine and its local port is also generated dynamically. Rest of the properties such as remote IP address, remote port and protocol are consistent.

Network socket close

Socket close function [Mic14, ms737582] provides the functionality to close the sockets. Since the socket name itself is not consistent across executions, this API call does not have much value for detection.

Network connection

According to a study [LCB10] majority of Internet traffic consists of Transmission Control Protocol (TCP) traffic. And TCP protocol requires that a connection is established between hosts [ISI81]. This event explains the network connections generated. Typically a connection consists of at least 5 elements. Source IP address, Destination IP address, source port, destination port and protocol. From these properties, the source IP and source port are not interesting due to the reasons mentioned above. However, destination IP, port and protocol can be very useful in identifying CnC servers.

Domain Name Server (DNS) query attempt

DNS queries are used to convert a URL to an IP address. Internet works internally on IP addresses, however, it is difficult to remember IP addresses. DNS servers provides the functionality of converting easy to remember string based domain names to an IP address. The DNS query events contains the domain string which was requested for resolution and it can be used in detection.

DNS query response

DNS query is usually responded by a DNS response. The response contains the name of the URL for which query was made, the IP address it resolves to, and any aliases this IP address has (CNAME) [Bar96]. If a known malicious domain name has been resolved, it usually indicates strong malicious behavior.

Since DNS is not static and a single domain can resolve to many IPs and multiple domains can resolve to a single IP [Bar96] the DNS request, response pair is not a very strong measure of malware behavior. However only DNS request is very useful as it is more likely that a malware will always request the resolution of same domain. But depending on the DNS the same query may result in different responses.

Network data transfer

Data is transferred over the Internet in the form of flows or sessions. A flow is defined by a unique tuple of 6 elements:

1. Source IP

2. Destination IP
3. Source port
4. Destination port
5. Protocol
6. Direction of data flow

A session consists of same elements above except the direction. So, in other words a flow is a set of data transferred between two machines in a specific direction on specific ports, while session represents the data transferred in both directions between two machines on predefined ports. This data transfer is very valuable because the malware communicates with its CnC servers by receiving and sending data to them. However, this can also cause FPs because many other applications also automatically send and receive data. So, special care must be taken when using this information to not classify standard traffic as malicious. Some common data transfer that takes places automatically on a Windows computer consists of:

- Windows Update check and installations
- Checking Internet connectivity by contacting msftncsi.com [HHW+10]
- Third party installed software which checks for updates

Important properties of this event are the remote IP, remote domain, remote port and transferred data.

URL Open

A URL acts as an address for Internet websites. URLs are generally descriptive and human readable. Because of these reasons, it is very helpful to monitor the URLs malware communicates with. This can lead to detection of malicious websites. The properties of this event generally consists of the URL and URN which is the path of the page which is accessed on the accessed domain.

3.4.3 Service Events

Windows service is just a process which runs in the background. It is very suitable for running long processes which need to execute continuously in the background [Mic14, d56de412]. Since services run non-stop in the background, they are very lucrative targets for malware. If a malware can manage to run as a service, it can perform continuous monitoring of the user activities, logging its key strokes or downloading new malware to the system.

Create Service

Creation of service event usually consists of name of the service, its display name and the process which it executes as the service [Mic14, ms682450]. There are also some other properties of this API call but the above mentioned are most important.

Delete Service

Apart from running as a service, a malware may be interested in deleting existing services such as Windows Firewall which prevents certain attacks on Windows systems. A delete service function [Mic14, ms682562] can be called for this purpose.

3.4.4 Process Events

Whenever a program is run on a Windows computer it runs in the form a process. A program may have more than one processes and a process may have additional threads [Mic14, ms684841]. Most of times malware samples also run as processes and some times as a thread of another process. Furthermore, it sometimes tries to interact with existing processes by either terminating them or stealing the information they have in memory. Therefore, the process events are very important for malware detection.

Create Process

When any executable is run, it results in creation of a process for that executable. Some arguments can also be passed to the executable before execution which act as user input for the process [Mic14, ms682425]. Both the arguments and the name of the process are useful for identifying malware. Some malware samples execute standard non-malicious processes such as Internet Explorer or Command Prompt (cmd). However, they pass certain arguments to these legitimate processes to perform malicious activities such as passing a URL to Internet explorer for execution or passing some batch script path to the cmd to execute that batch script.

Read Process Memory

It is possible to read the memory of another process by calling ReadProcessMemory function [Mic14, ms680553]. The properties of this event include a handle to the process, size and location of the memory being read. The handle is generally not useful as its operating system generated but by using that handle it is possible to get the name of the process which is usually more reliable.

Terminate Process

A process can be terminated by calling Terminate Process [Mic14, ms686714] Function. Only the name of the process is important from this event.

Write Process Memory

WriteProcessMemory [Mic14, ms681674] is similar to ReadProcessMemory except that it writes to the process memory rather than reading it. The name of the process which is writing and the name of process which is being written onto are important properties of this event.

Create Thread

A process can have multiple threads. The major benefit of a thread over a process is that multiple threads of a single process can run in parallel but the code which is part of a single-threaded process runs sequentially. Therefore, threads are used extensively in parallel applications where the execution of different parts of program in parallel are more beneficial over sequential execution [Mic14, 6kac2kdh]. For the purpose of malware detection, the name of the process which creates the thread can be useful.

3.4.5 Registry Events

Registry is a database used to store and retrieve the configuration settings of software installed on a Microsoft Windows system. Registry, not only contains the configuration of third party installed software but it also contains the configuration of different components of Windows and its core software itself [Mic14, ms724871]. This makes registry very critical because changing some values in registry can change how core components of Windows behave. Similarly, registry changes can change the behavior of other installed software. Malware leverages this power and uses the cover of legitimate processes to hide its activities by changing their configuration and making them perform the malicious activities for them.

Registry database is structured in the form of a tree. Each node in the tree is called a registry key and each node may have child nodes [Mic14, ms724871]. Apart from having children which are called subkeys, a node may also have value fields which contain the configuration information.

Create Registry Key

CreateRegistryKey function [Mic14, ms724844] creates a key in a given location. The location where registry gets created is important information which can help in detection. An example of a suspicious registry key creation is creation of following registry.

```
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_RASMAN\0000\
    Control
```

ControlSet001 key contains configuration for device drivers and services. Malware can create keys in this location with the name of their malicious services to store the malware's configuration information.

Delete Registry Key

Delete registry function [Mic14, ms724847] deletes the registry key specified as parameter along with all the values that registry contains.

Read Registry Key

A registry key can be opened using RegOpenKeyEx function [Mic14, ms724897]. This function returns a handle which can be used to read the specific attributes such as values and sub-keys of the given key. Malware may read current registry values to detect the current state of the system and perform malicious activities accordingly.

For some software, both malicious and benign it may be enough to just create or detect the presence of a key to indicate certain configuration. The boolean configuration values can be stored in registry this way. However, for some advanced configuration the registry value fields are used to store the explicit strings representing the configuration. The remaining registry events in this section are about registry values.

Delete Registry Value

RegDeleteKeyValue API lets the caller remove a specific value from a given key [Mic14, ms724848]. Malware may use this to remove existing legitimate key values to change behavior of system software.

Query Registry Value

Query registry value just returns the value of the registry queried [Mic14, ms724909]. The value and the name of registry are properties of this event.

Save Registry Value

This event saves a value in an already existing registry key. The value and the path of registry are important for this event [Mic14, ms724921].

Modify Registry value

This key is same as save registry key value except that it replaces an already existing registry key which already has a value.

3.4.6 Miscellaneous Events

There are some further events which can be useful for malware detection. They are explained below.

Create Event

An event [Mic14, ms682655] is a synchronization object which can be used for certain synchronization functions. A function call can be used to set the state of the event to signaled and other threads can check the current state of the event. One of the use cases is to tell the waiting threads that a certain event has occurred so they can continue their execution. They also have some other uses mentioned in [Mic14, ms686915]. The name of the event can be useful for identifying malware as they sometimes produce events with reproducible name.

Create Mutex

A Mutex is also a synchronization object [Mic14, ms684266]. It is used to give safe access of a shared resource to multiple threads. One of the threads can acquire the mutex and use the shared resource. After its done using the resource, it can release its lock on mutex. The other threads can then acquire the mutex and perform their activities. This prevents the cases where multiple threads are trying to use a resource simultaneously and the end result of resource becomes something which was not planned. Mutex can also be used to synchronize processes. Some malware samples launch multiple copies of themselves at the same time and then use mutexes to synchronize between multiple running instances. The name of the mutex is sometimes random but sometimes the creating process sets it to a specific string. Such consistent mutex name strings are useful for malware detection.

Create Semaphore

A semaphore is a synchronization object similar to mutex. A mutex can only be released by a thread which acquires it but semaphore provides a more sophisticated system where other threads and process can also signal the semaphores. A semaphore provides a count value which is decremented every time a thread acquires the lock of the resource. The same count value is incremented every time a thread releases the lock [Mic14, ms685129]. If the value of the count variable reaches zero, no further locks can be acquired on it. However, for the purpose of malware detection only the name of semaphore provides a behavior specific to a program. Similar malware samples may produce semaphores with same or similar names which can be used for their detection.

3.5 Summary

This chapter explained what malware is and gave some examples of malicious activities they perform. Some examples of real malware samples were also given to show what malware is capable of. In the second half of the chapter behavior based malware analysis was explained i.e. a malware sample is executed in a virtualized environment to get all the activities it performs in the form of events. These events are then matched against behavioral patterns to identify malware. In section 3.4, a detailed description of important events generated by software in Microsoft Windows was given. The importance of the events with respect to malware analysis was also discussed. In the coming chapters these events will be used to cluster malware and later generate behavior patterns based on them.

Chapter 4 Clustering

Clustering is a technique which can be used to group a set of objects into clusters. The clusters should be such that the elements belonging to the same cluster should be similar to each other. An element belonging to one cluster should be more similar to other elements of its cluster compared to elements of other clusters. Figure 4.1 shows an example where data in 2 dimensions is clustered. Each color represents a different cluster.

Since clustering helps in finding the elements which are similar to each other. It can also help us identify the similarities between elements. This idea can be extended to behavior based pattern generation of malware and similarities between elements can be used as patterns to identify all the elements of that cluster.

In start of this chapter, clustering will be explained briefly with a simple example. Followed by section 4.1 on hierarchical clustering which will explain hierarchical clustering in detail with an example. Different clustering schemes and clustering metrics will be explained. Towards the end of chapter in section 4.2 partition based clustering algorithms will be described briefly and will be compared with hierarchical clustering.

Consider a tourist visits a forest in another continent. He sees many new species of animals and birds there which he has never seen before. He quickly gets overwhelmed with so much diversity, so he decides to divide all the animals and birds into a set of groups based on their properties. He starts with noting down all the characteristics he can observe of each new specie he observes. For the sake of simplicity he starts with only 4 properties and makes a table similar to Table 4.1.

From this table it is very easy to divide these species into groups based on their properties. For example three species 2, 4 and 6, all have 4 legs and no wings and their weight is considerably large (50 to 2000 kg). He initially makes one group out of these. He then proceeds onto making a different group out of 1 and 3. Both of

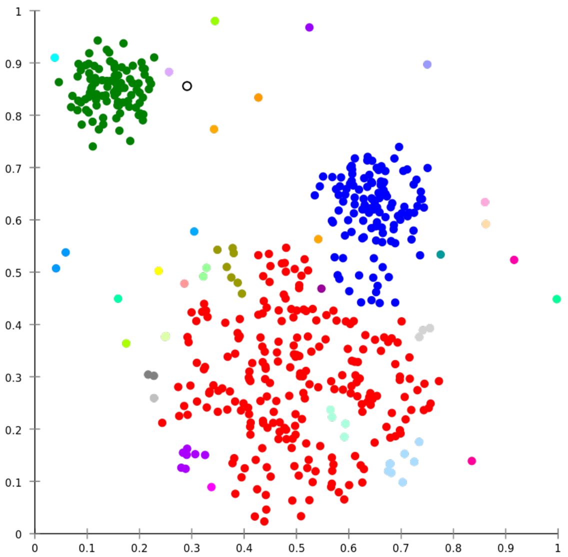

Figure 4.1: An example of clustering. Different colours represent different clusters [CC11]

Table 4.1: Species and their properties

Serial No	No of Legs	Has Wings?	Colour	Weight (kg)
1	2	yes	blue	2
2	4	no	brown	100
3	2	yes	white	20
4	4	no	gray	2000
5	0	no	black	1
6	4	no	brown	50

these species have 2 legs, they have wings and their weight is comparatively small (2 to 20 kg). And finally he makes a third group of one remaining specie which has no legs, no wings and doesn't weight much.

So, after performing this classification the results can be seen in Table 4.2.

Table 4.2: Species, their properties and the groups they belong to

Serial No	Group No	No of Legs	Has Wings?	Colour	Weight (kg)
1	2	2	yes	blue	2
2	1	4	no	brown	100
3	2	2	yes	white	20
4	1	4	no	gray	2000
5	3	0	no	black	1
6	1	4	no	brown	50

The benefit of performing this computation is that now the tourist can explain the species better by just explaining the three groups or clusters he generated rather than explaining each of the species separately. He can understand each individual specie better by observing the species which are similar to him. He might be able to study the species belonging to one group to find some other characteristics of the group which were not considered when clustering the data such as whether a group is dangerous or not. This information can then be useful when another specie is discovered and added to current clustering. If it gets classified into a group which contains dangerous species, chances are that newly discovered specie is also dangerous.

The process done above is a simple example of clustering. In the example above a set of data was gathered and it was clustered into three groups. Elements in each group were similar to each other and had more in common with elements of their own cluster compared to elements of other clusters. This idea can be extended to cluster much different and complex data. Clustering helps in understanding and visualizing the data which is otherwise too big to be understood by a human being. However, practical clustering problems are much more complex. The number of elements which are to be clustered may be much larger. Millions of elements are quite possible in certain situations. Secondly, in the example we saw only four properties of each element. In practical problems, number of properties may be much larger and type of properties may be complex such as a floating point number with no defined range or text strings.

There are many systematic approaches to clustering data into groups. Usually, first pair-wise distance of all the data points is calculated using some similarity metric. After the distances of all points to all points are obtained then the points which are closest to each other can be found and put into same cluster.

Majority of clustering schemes can be classified into two categories: hierarchical clustering and partition based clustering [KR09]. Hierarchical clustering is a bottom up clustering algorithm where elements are assigned to small clusters and then clusters are joined together to make larger clusters, whereas partition based clustering is top down clustering algorithm where elements are divided into predefined number of clusters and then each cluster is subdivided into more clusters if needed. In the sections below hierarchical and partition based clustering will be discussed and how they can help in clustering malware.

4.1 Hierarchical Clustering

Hierarchical clustering is an algorithm which divides data into clusters and produces a structure which can be displayed as binary tree or a dendogram among other ways. This tree can be traversed to find elements belonging to a specific cluster. Hierarchical clustering algorithm is a very generic algorithm which performs clustering. It can be used to achieve very different results by configuring the different parameters it depends on. In this section the major skeleton of hierarchical clustering algorithm will be explained. In the sections below the parameters which can be changed will be explained further.

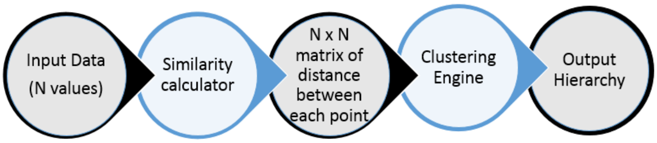

Figure 4.2: Hierarchical Clustering. Dark circles represent the data while light circles represent the processes.

Given a dataset first the features of the data are selected based on the type of problem. The selection of features is an important step and choice affects the results heavily. However, there is no known correct method to choose features, the selection depends heavily on the data and final goal of clustering. After feature selection, distance between features of each point to features of each other point is calculated using some clustering metric. Clustering metrics are further explained in section 4.1.2.

The step of measuring distances between points is a quadratic time and quadratic space algorithm. If the number of elements to be clustered are n, the algorithm will take $O(n^2)$ space and $O(n^2)$ operations to complete [Mül13]. The result of this step is an $n \times n$ matrix.

To save memory, the distance matrix is usually stored in the form of upper triangular matrix. Since the distance of an element to itself is 0, diagonals of the distance matrix can be omitted. And lower triangle of the distance matrix is a mirror copy of upper triangle so it can also be omitted. An example of distance matrix can be seen in Table 4.4.

After calculating the matrix the smallest value in that matrix is selected to get the elements which are closest to each other. Those elements are then put into a cluster. In the next step again the next smallest value is chosen to find the next two closest elements and they are put together into a cluster. At the end, all the elements are clustered together in the form of a tree.

This can be better explained using a small example.

4.1.1 Example

Four data points are taken in a 2D space. Each point contains two dimensions in a 2D plane, an x coordinate and a y coordinate. They are to be clustered based on the distance between the points. So, the points which are close to each other go into same cluster. Points are shown in Table 4.3 and their plot can be seen in Figure 4.3.

Table 4.3: Input data for hierarchical clustering algorithm

Serial No	Element
1	(0.7, 0.8)
2	(0.2, 0.95)
3	(0.8, 0.8)
4	(0.87, 0.629)

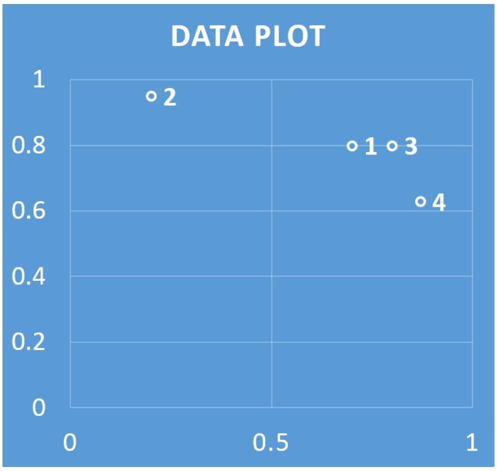

Figure 4.3: Plot of data to be clustered

Depending on the type of data, an appropriate distance function should be chosen. There are many possibilities, some of them are discussed in section 4.1.2. For this particular case data consists of 2D points. Each co-ordinate of the point is selected as a feature. Then euclidean distance is the appropriate measure of distance between points. For points (x_1, y_1) and (x_2, y_2) euclidean distance is given by Equation 4.1.

$$\text{Euclidean distance} = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2} \tag{4.1}$$

As a first step of calculations, distance of each point with each other point has to be measured and stored in the form of a matrix. The resulting matrix is shown in Table 4.4.

Table 4.4: Distance Matrix

	1	2	3	4
1		0.52	0.10	0.24
2			0.62	0.74
3				0.18
4				

After getting the distance matrix, the value which is smallest in whole matrix has to be found. In this example that value is 0.10 which is the distance between point 1 and point 3. At this point, point 1 and point 3 are known to be closest to each other so they can be put in one cluster. After deciding a cluster the rows and columns representing point 1 and 3 can be merged to form a combined column. The next step is to find the distance from that cluster to all other points. There are many ways to calculate the distance between a cluster and a point or between multiple clusters. They will be explained in depth in section 4.1.3. For now the distance between a point a and a cluster B will be taken as distance between point a and the point closest to a which belongs to B.

After merging column 1 and column 3, the distance between cluster (1,3) and all other points in the table have to be calculated. After making these computations the Table 4.4 shrinks down to Table 4.5.

Table 4.5: Distance Matrix

	(1,3)	2	4
(1,3)		0.52	0.18
2			0.74
4			

Now using the same system the smallest value in the matrix is to be found which is 0.18 in this table. 0.18 is the distance between cluster (1,3) and point 4. These 2 entities (a cluster and a point) can be combined to make a bigger cluster named ((1,3), 4). Similarly the Table 4.5 can now be further shrunk down to Table 4.6.

Table 4.6: Distance Matrix

	((1,3),4)	2
((1,3),4)		0.52
2		

Now the smallest entry in the table is 0.52 which is also the only entry. This completes the clustering process to give the following result:

(((1,3),4),2)

The final result can be seen in Figure 4.4.

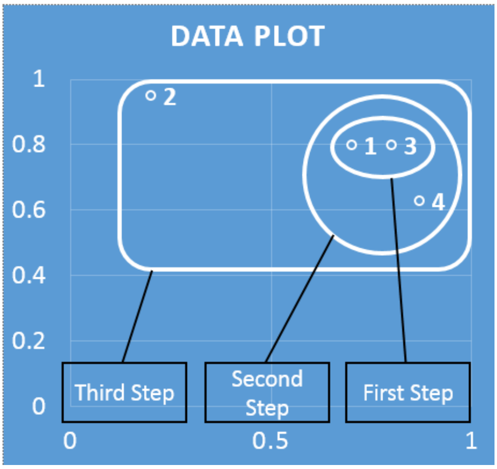

Figure 4.4: Hierarchical Clusters

.

This can be plotted in the form of a dendogram as shown in Figure 4.5.

The end result of hierarchical clustering is a dendogram which shows hierarchy of relationship between elements. This dendogram can be cut down at a particular point to retrieve the individual clusters. For example this dendogram can be cut at root node to get 2 clusters. One containing elements 4, 1 and 3 while other containing element 2.

The height of the dendogram represents how further apart the underlying clusters are. In Figure 4.5 it can be seen that point 2 is much farther from cluster ((1, 3), 4).

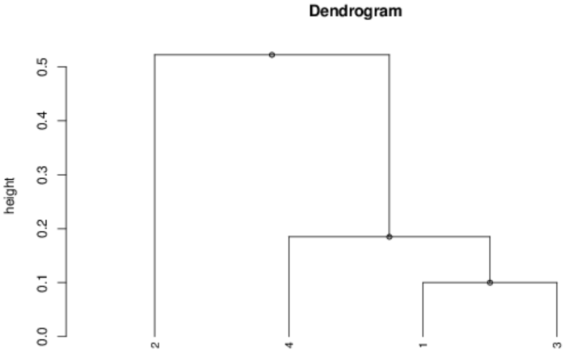

Figure 4.5: Dendogram of clustering. It can be seen that point 1 and 3 are closest to each other followed by point 4 and finally point 2. The diagram was drawn using the software [Wes12].

In the following sections different parameters which can be used to customize hierarchical clustering will be explained.

4.1.2 Clustering Metrics

Finding the distance between two points is an essential component of hierarchical clustering. In the example mentioned in section 4.1.1 euclidean distance was used to compute the distance between points. Euclidean distance is a very good distance measure for multidimensional numerical data where only the location of the points contribute towards the distance. However for non-numerical data such as text, euclidean distance is not always suitable.

As described in section 3.4 software generates many events of many different types. Each event may have different number and different type of sub-events. This makes distance computation between events of two software applications or malware samples a complex task.

In the sections below different methods of computing the distance between software events will be explained.

NCD

NCD is a distance metric which uses data compression algorithms to compute the distance between two strings. Data is compressed by finding repetitions so multiple copies of same data are stored only once with some extra information to reduce the space required to save the data. This feature of data compression is used to find similarities or repetitions between two different strings.

If x and y are two strings and Z is an good compression function. Their NCD is given by Equation 4.2.

$$NCD(x,y) = \frac{Z(x \mid y) - min\{(Z(x), Z(y)\}}{max\{(Z(x), Z(y)\}} \tag{4.2}$$

Strings x and y are concatenated and compressed to compute $Z(x \mid y)$ [CV05].

If all the events of a software are concatenated together in the form of a string then NCD can be used to compute the distance between the events. NCD is useful for general clustering as its fast and focuses more on similarities than distances but its not very effective for pattern generation as the similarities are not easily extractable without going in depth of compression algorithms. One other drawback of using NCD for pattern generation is that, it considers all the events as a single string so if two events are similar but they are of different types such as:

```
{FileCreate: {"path": "c:\Program Files\Common Files\
   Microsoft Shared\Web Folders\Microsoft Office 2003
   Crack.exe"}}

{ProcessStart: {"path": "c:\Program Files\Common Files\
   Microsoft Shared\Web Folders\Microsoft Office 2003
   Crack.exe"}}
```

The contents of these events are exactly the same but their types are different. Since NCD does not take into consideration any sub parts of the string, they will be considered very similar events but as their type is different they should be considered completely different.

Bailey et al. used NCD to cluster data based on dynamic behavior in their research paper [BOA$^+$07].

Jaccard Index

Jaccard Index [Jac01] is a measure of similarity between two sets. If A and B are two sets then their Jaccard index is defined by Equation 4.3.

$$J(A,B) = \frac{\mid A \cap B \mid}{\mid A \cup B \mid} \tag{4.3}$$

The value of Jaccard index is between 0 and 1. Since Jaccard index is a measure of how similar two sets are, it can be easily modified to find the distance or dissimilarity between two sets. The distance based on Jaccard index is called Jaccard distance and is given by Equation 4.4.

$$d_J(A, B) = 1 - J(A, B) \tag{4.4}$$

Because Jaccard distance is computed on sets, it can be used in many different ways depending on how the data is converted to sets. In the problem of finding distance between software events, each event can be be considered as one element of a set. The distance of such sets will depend on the events which are completely identical in both sets, while ignoring the events which are anything less than 100% identical. In a different setting each property of event such as path of a FileCreate event can be considered as an element of set. Then the set will consist of all such properties and the resulting distance will be influenced by common properties created by both software samples. Yet in another setting each character of the both events can be considered an element of set and then similarity will be measured at character level.

Ssdeep

Ssdeep is a fuzzy hash based similarity detection tool. The tool first computes the hashes of both the files or strings which need to be compared. Then, rather than matching the files with each other which are way bigger, only hashes of the files are compared with each other. Ssdeep has a hash comparison tool included which can calculate how similar two files were based on their hash values. Ssdeep can be downloaded from [She14].

4.1.3 Linkage Criteria

When performing hierarchical clustering the first cluster always contains two data points. The rest of clusters may consist of a point and a cluster or a cluster and a cluster. This leads to the problem on how to compute the distance between two clusters. There are many ways this can be achieved. Some of these criteria will be discussed below.

Complete linkage clustering

Complete linkage method finds the distance between two clusters as the maximum distance between any two points of clusters.

$$max\{d(a,b) : a \in A, b \in B\} \tag{4.5}$$

where A and B are two clusters and a and b are points belonging to A and B respectively. This method is also called maximum linkage clustering. The best hierarchical clustering algorithm based on complete linkage criteria has complexity of $O(n^2)$.

Single linkage clustering

Single linkage method is opposite to complete linkage and it finds the distance between two clusters as the minimum distance between any two points of clusters.

$$min\{d(a,b) : a \in A, b \in B\} \tag{4.6}$$

where A and B are two clusters and a and b are points belonging to A and B respectively. This method is also called minimum linkage clustering and its computational complexity is also $O(n^2)$.

Average linkage clustering

Average linkage clustering computes the distance between clusters as the average of distance between all points of clusters [RC58].

$$\frac{1}{\mid A \mid . \mid B \mid} \sum_{a \in A} \sum_{b \in B} d(a,b) \tag{4.7}$$

$O(n^2)$ algorithm also exists for computing hierarchical clustering using this method.

There are many other methods which can be used, however only most commonly used methods have been explained above.

More on hierarchical clustering can be read in this survey [XW05].

4.2 Partition based clustering

Hierarchical clustering is a bottom up clustering mechanism which is started by putting two elements into a cluster and adding more and more elements to make a hierarchy. This hierarchy provides a relationship between smaller clusters. And each cluster can be further broken down in smaller clusters. All this information is inherently present in hierarchical clustering output.

Partition based clustering is exactly opposite to that. It's a top down approach where the number of clusters are decided up front. Then, each element is mapped to one of the clusters. There are many schemes that fall under the category of partition based clustering. One such famous scheme is called k-means clustering [JD88].

This technique is very useful for certain clustering problems. One advantage it has over hierarchical clustering is that it can be very easily parallelized and elements can be broken down into any number of predefined clusters. However, the ability to specify the clusters upfront is also one of biggest disadvantages of such schemes because sometimes its not easy to determine how many clusters are reasonable for a certain problem.

As will be explained in chapter 5 the tree structure of clustering is very helpful in finding the patterns which detect maximum number of samples. The tree structure of hierarchical clustering proves to be very critical for that. Since partition based clustering does not have tree like structure, pattern extraction algorithm presented in chapter 5 can not be applied to it.

4.3 Summary

This chapter explained the clustering of software based on its behaviors. General clustering was introduced using some examples. The notion of hierarchical clustering was explained in depth with examples. There are many different parameters which can affect the result of hierarchical clustering. Those parameters were explained and their importance with respect to software clustering based on events was discussed. Towards the end of the chapter, partition based clustering was introduced and it was mentioned that hierarchical clustering is more suitable for pattern extraction algorithm developed in this thesis compared to partition based clustering.

The ideas from this chapter will be taken forward to chapter 5 where a scheme will be presented which will extract patterns from malware samples after clustering them using hierarchical clustering.

Chapter 5
Malware Pattern Generator

This chapter presents Malware Pattern Generator (MPG). A tool which can automatically generate patterns for malware detection from a given dataset of malware samples. This chapter will explain the design and implementation of MPG. Three similar but slightly different systems were built and evaluated which will be explained in the text below. In section 5.1 overall architecture of all three approaches will be explained. From section 5.2 onwards, the components which constitute these systems will be explained in depth.

5.1 Architecture

The software which is written to generate automatic patterns for malware is named MPG. As explained in chapter 3, when software is run in a virtualized environment the events it generates in the system can be recorded. These events act as features of the malware which will be used in clustering the malware. Details of which features are chosen and how important they are for malware clustering and pattern generation was explained in section 3.4. After the system events generated by malware are collected, they can be fed to MPG to get behavior pattern groups. A malware sample may produce many events of many different types. Each event can have one or more event properties. An example of events created by a malware sample is given in appendix A.

After processing the events of the samples from dataset, MPG returns a set of pattern groups which attempt to detect the samples present in the dataset. A pattern group is a collection of patterns with some relationship between patterns. The realtionship can be either “and” (intersection) or “or” (union) meaning all patterns have to match for pattern group to match or one of patterns has to match for pattern group to match. A pattern consists of one or more sub-patterns with similar “and”, “or” relationship. Appendix B contains an example of a malware pattern group.

Figure 5.1 shows this in the form of a diagram. If MPG is considered a blockbox, it takes in events and produces pattern groups as output.

Events
MPG
Pattern Groups

Figure 5.1: Input and Output of MPG.

MPG can be decomposed into different components. These components will be explained in detail in the section 5.2 onwards. During the work done for this thesis three slightly different versions of MPG were developed. They are named as

1. MPG Single-Events
2. MPG Multiple-Events
3. MPG Fine

This section will explain the architecture of all three approaches.

5.1.1 MPG Single-Events

The architecture of this approach can be seen in Figure 5.2. This approach is named Single-Events because the resulting pattern groups consists of events of only a single type.

Events from all the samples in the dataset are collected by running them in Blue Coat's Malware Analysis Appliance (MAA). They are then filtered to remove un-necessary events. As can be seen in Appendix A, different events belonging to a sample can be of different types. After filtering, the events are sliced into different buckets based on type of event. After this step each bucket will contain events from all the samples of a specific type. Section 5.2 explains the filtering and slicing in more detail. Subsequently, each bucket is clustered hierarchically independent of other buckets. The clustering will create a hierarchy of each bucket of events based on their similarity with other events in the bucket. Details of clustering are explained in section 5.3 and section 5.4. After finding which samples are close to each other, events which are shared by multiple samples will be extracted and stored in a tree in *common events extractor and FP remover* block (section 5.5). The events which can cause false positives will also be removed in this block. Then, out of all common events, the patterns which have maximum coverage will be selected in *patterns extractor* block. This block will output a set of patterns for only its specific category. Finally all patterns are collected from each category.

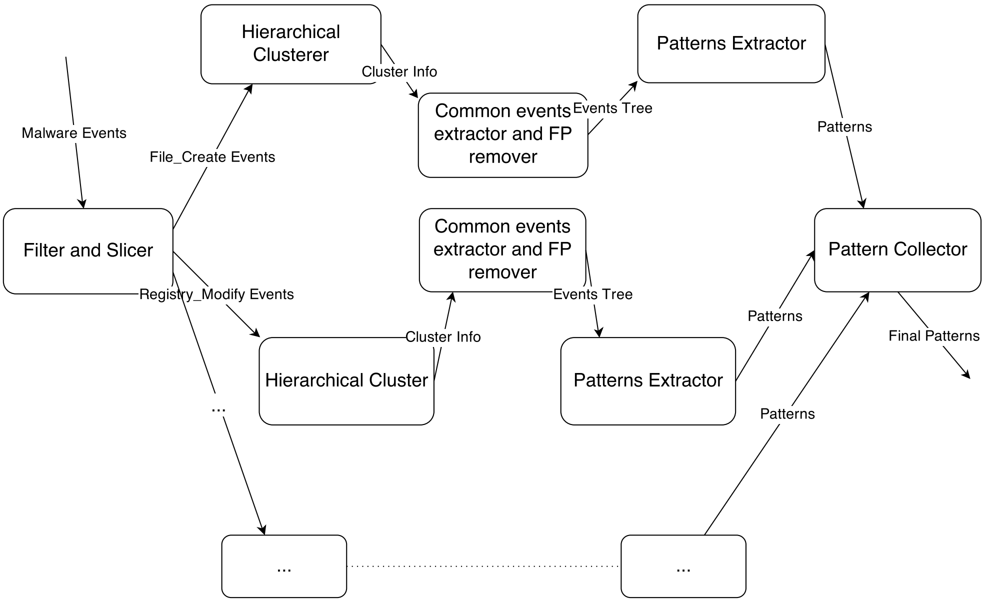

Figure 5.2: Architecture of MPG Single-Events.

All the components of this approach will be explained in detail in section 5.2 onwards.

5.1.2 MPG Multiple-Events

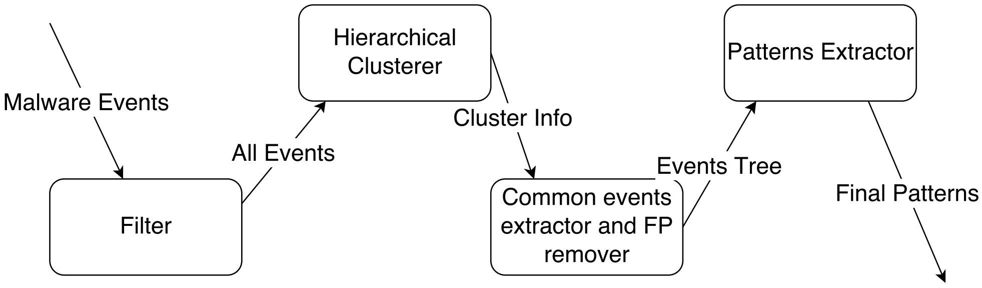

Figure 5.3: Architecture of MPG Multiple-Events.

One of the shortcomings of the approach of Single-Events is that the resulting pattern group consists of only one type of events. However, some malware samples may share more than one events of different types and using multiple events of different types to create a pattern group has potential of generating patterns which are stronger than Single-Event approach and are less false positive prone. Therefore, a separate system was written to take into consideration multiple events. The architecture of multiple events is very similar to Single-Events except the component of slicing is removed. In this approach all the events are clustered hierarchically together regardless of their type. The simplified architecture can be seen in Figure 5.3.

The disadvantage of this approach is that it requires more processing resources than Single-Events approach as all events are clustered together.

Let m be the number of events in sample 1, while n be the number of events in sample 2. In approach of Multiple-Events the number of comparisons required are $n \times m$. While, in Single-Events approach each bucket is clustered separately therefore the required operations are less than $n \times m$.

5.1.3 MPG Fine

This approach is an extension of MPG Multiple-Events. The architecture is same as MPG Multiple-Events but clusterer and pattern extractor components are changed to incorporate finer details of events when clustering and extracting patterns. Exact differences will follow in the text.

This approach requires even more processing power than both previous approaches but it incorporates not only complete events but also event properties. Hence, promising more accurate pattern groups.

5.2 Filter and Slicer

A malware sample may produce many events. Some complete events are known to cause false positives while only some parts of some other events are not useful for the purpose of pattern generation. These events are filtered out in this section. Since filtering requires traversing all the events, they are also sliced in this step based on their types. Sub-sections below will explain this in more detail.

5.2.1 Inconsistent event properties removal

When malware is run in a virtualized environment there are many events which can be recorded. As explained in section 3.4 some properties of the events are not consistent across multiple executions of same malware sample. The biggest portion of such event properties are OS generated handles. File handles, process Identifiers (IDs) and network sockets are examples of such properties.

This section removes all the OS generated inconsistent event properties but keeps the rest of properties intact. For example, if an event of type *file_create* has two properties: *file_path* and *file_handle*, *file_path* will be kept while *file_handle* will be removed because *file_handle* is generated by OS dynamically and does not remain static across multiple executions of same malware sample. This module removes following properties because of the same reason.

- thread handle
- file handle
- process ID
- socket
- source port
- source address

After removal of these properties, only the reproducible events remain in the system.

5.2.2 Event white-list

There are certain events which are not suitable for malware pattern creation. An example of such an event is network communication with “windowsupdate.com”. Such events are of a considerably large number and removing them at the earliest stage helps in efficiency of the system because it does not have to process the events which are known to be useless from the perspective of malware pattern generation. Furthermore, because of early removal, they do not influence the clustering which will be explained in section 5.3 and section 5.4.

This section does not only remove the property of the event which is in white-list but it removes the complete event which contains a white-listed property.

Most common domains which are accessed by windows system itself or by non-malicious software are white-listed. Such as “windowsupdate.com”, “google.com” “akamai.net” and much more. The commonly accessed IP address 8.8.8.8 is white-listed as this address is the address of google’s DNS service. Apart from this, the private IP address ranges are white-listed. These ranges include 192.168.0.0/16, 172.16.0.0/12 and 10.0.0.0/8.

5.2.3 Slicer

Slicing is only performed for MPG Single-Events. In this step, events belonging to each malware are grouped together based on their type and then each type of events are stored in a separate file on the hard-drive.

For MPG Multiple-Event and MPG Fine, slicing doesn't take place. After performing the filtering, filtered events are stored on hard-drive together without slicing.

5.3 Distance Matrix generation

The objective of MPG is to generate detection patterns for malware. And events which are common in multiple malware samples are strong candidates for those patterns. To achieve this goal, malware samples are clustered to find the samples which have common events amongst them. Chapter 4 presented two clustering schemes and stated that hierarchical clustering is more suitable for the problem of pattern generation. This will become more obvious in section 5.5 when the fact that hierarchical clustering provides a tree structure as output will be used to extract patterns from samples.

After pre-processing the input data, hierarchical clustering has to take place. In this step the malware samples will be clustered together based on the similarities they have in their events. The first step in clustering is generation of an $n \times n$ distance matrix from the n malware samples. The theoretical details of hierarchical clustering were explained in section 4.1. This and next section will only contain the details of hierarchical clustering from implementation point of view.

As explained in section 4.1 a distance matrix can be stored in upper triangular form to save space. However, still a table containing rows and columns has to be created which has memory overhead. To get rid of this overhead the distance matrix can be flattened by putting all the values next to each other in a list rather than in a table. If this optimization is applied to Table 4.4 the output will be as below:

```
[0.52, 0.10, 0.24, 0.62, 0.74, 0.18]
```

This format of storing distance matrices is called condensed distance matrix and it can be given as input to many clustering libraries including fastcluster [Mül13] and scipy [JOP01]. Condensed distance matrix requires less memory compared to the standard table. Secondly, this list provides all the functionality of a table and location of distance between any two points can be calculated easily. During the implementation of MPG the distance matrix was stored in condensed distance matrix format. Algorithm 5.1 shows how to calculate the distance matrix in Python.

Algorithm 5.1 Distance matrix generation algorithm.

```
m = []
for i in xrange(len(files)):
    for j in xrange(i + 1, len(files)):
        m.append(jaccard_index(files[i], files[j]))
return m
```

Since generation of distance matrix is an $O(n^2)$ task, its output is saved on hard disk and is read again from hard disk for clustering.

5.3.1 Jaccard Distance

MPG uses Jaccard distance as measure of distance between malware samples. All clustering schemes classify the elements into clusters based on their similarity. As a simple example if two elements are exactly the same they should be in same cluster. In general clustering problems, it is not that important how extractable the similarity is, but in our particular problem the objective is to find detection patterns. So, it is important how similar two samples are but what exactly is similar in them is more important. Section 4.1.2 presented three different clustering metrics. Based on this criteria out of the three approaches presented, Jaccard distance was chosen because it is calculated based on intersection and union. And the intersection precisely provides the events which are common in two samples. This property is not present in other techniques where extracting the common events out of samples is not trivial. Furthermore, malware events can be easily converted to sets using $set()$ or $frozenset()$ functions of Python. Different granularity can be used depending on the problem.

MPG Single-Events and Multiple-Events

To calculate the Jaccard distance the data must first be converted into sets. In case of MPG Single-Events and Multiple-Events, each complete event is considered one element of set. Then the union and intersection of these sets is calculated. The values of union and intersection are then plugged in equation 5.1 to calculate the Jaccard distance. This algorithm can also be seen in Algorithm 5.2.

$$J(A, B) = 1 - \frac{\mid A \cap B \mid}{\mid A \cup B \mid} \tag{5.1}$$

Algorithm 5.2 Jaccard Index Algorithm

```
#data1, data2 : A list containing events in text format.

set1 = frozenset(data1)
set2 = frozenset(data2)

intersection = len(set1 & set2)
union = len(set1 | set2)

jaccard_coefficient = intersection/float(union)
jaccard_distance = 1 - jaccard_coefficient
```

MPG Fine

MPG Fine uses a slightly modified Jaccard distance to also take properties of events into account. In the approach mentioned in Algorithm 5.2 if two events are 80% similar they will not influence the Jaccard distance at all. Anything less than 100% similarity has no effect on the distance. Due to this limitation the results are not as good as they theoretically can be.

To overcome this problem a separate algorithm was designed to take into consideration not only events but also their properties.

As a first step the normal Jaccard index is calculated using Algorithm 5.2. After calculating this Jaccard index the final result is adjusted based on similarities between other events which are not 100% identical.

Let J_f (Jaccard full) be normal Jaccard index calculated by Equation 5.2.

$$J_f(A,B) = \frac{\mid A \cap B \mid}{\mid A \cup B \mid} \tag{5.2}$$

Where A and B are sets of events from two malware samples. Each element of set A and set B is a complete event with all its properties. So, $A \cap B$ will contain only the events which are completely identical in both A and B.

Let J_a (Jaccard adjustment) be adjustment to the J_f based on event properties then J (Final Jaccard Index) will be

$$J(A,B) = J_f(A,B) + J_a(A,B) \tag{5.3}$$

And Final Jaccard distance will be

$$J_d(A, B) = 1 - J(A, B) \tag{5.4}$$

Method of computing J_a is explained below.

1. A' and B' are calculated using:

   $$A' = A - (A \cap B) \tag{5.5}$$

   $$B' = B - (A \cap B) \tag{5.6}$$

   At this point $A' \cap B'$ should be $\{\varnothing\}$ because all the common elements have already been subtracted from A and B to get A' and B'.

2. A' and B' are sliced based on event type. Let a, b, c ... be event types, then A'_a, A'_b, A'_c ... are sets containing only events of type a, b, c ... respectively. Similarly slices of B' are obtained.

3. Pair wise Jaccard index of elements belonging to each slice is calculated. For this step Jaccard index is calculated on properties rather than the complete events. Standard Jaccard index formula formalized in equation 5.2 is used for measuring the index. This can be better explained using an example. Let:

   A'_a = { {“path”:”file.txt”, “attributes”:534238}, {“path”:”file2.txt”, “attributes”:534238}}

   B'_a = { {“path”:”file.txt”, “attributes”:233482}}

   A'_a contains two events while B'_a contains only one event containing two properties each. So, Jaccard index between first event of A'_a and first event of B'_a will be 1/3 because length of union is 1 and length of intersection is 3. Similarly by taking Jaccard index of remaining pairs a table such as Table 5.1 can be obtained.

   Table 5.1: Jaccard index matrix

$B'_a \backslash A'_a$	event1	event2
event1	1/3	0

4. After calculating Jaccard index, matrix events from B'_a and A'_a can be linked together based on how similar they are to each other. This is done by finding the largest value in the table and linking the events it belongs to together. In the example of Table 5.1, it can be seen that event1 of B'_a is closer to event1 of

A'_a compared to the other options as they have largest value (1/3) in the table. So, both these elements are eliminated from the table and their similarity is noted down which is 1/3.

After removing the most similar events, the next most similar events are found in the table by finding the maximum in the table. In this particular example there are no elements left in B'_a set so the process will stop. But if there would be more, entries procedure would be kept repeating until all entries belonging to one of the sets in the table get removed.

At this point, all the similarity scores retrieved from the table are added together to get the overall similarity of slice.

Let S_a be similarity of the slice a then,

$$S_a = \sum \text{Jaccard index of most similar elements} \tag{5.7}$$

The final adjustment of slice a is given by:

$$Adj_a = \frac{\mid A'_a \cup B'_a \mid}{\mid A \cup B \mid} \times \frac{S_a}{(\mid A'_a \cup B'_a \mid) - S_a} \tag{5.8}$$

In equation 5.8, S_a is normalized by plugging it into Jaccard index formula. That normalizes the similarity and brings it back in the range of 0 to 1. After normalization of the similarity S_a, the normalized similarity is further normalized according to $A \cup B$ so that sum of Adjs (equation 5.9) of all slices will be within the range of 0 and 1.

Using the same process adjustment of all slices is obtained.

5. All adjustments are added together to get the final adjustment J_a

$$J_a = \sum_{x \in \{a,b,c,\ldots\}} Adj_x \tag{5.9}$$

Final Jaccard distance is used for computing the clustering. This Jaccard distance provides more accuracy than the distance calculated in previous step. However, this process takes much more computing resources. Furthermore, this improved Jaccard distance is still between 0 and 1 and can be used instead of the previous Jaccard index without any other architectural changes (for clustering).

5.4 Hierarchical clustering

Hierarchical clustering is an $O(n^2)$ operation which has been studied extensively and therefore many libraries exist which can be used to compute hierarchical clustering. Fastcluster [Mül13] is a library implemented in C++ and supports most common hierarchical clustering algorithms. The library provides interface to Python and R. So, the speed of C++ can be achieved without loosing the simplicity of programming in Python.

All variations of MPG use fastcluster library with average linking as linkage criterion (section 4.1.3). When calculating the distance between clusters, average linkage criterion calculates the pair-wise distance between all points of both clusters and then takes an average. It takes into consideration all the points of both clusters which provides better accuracy than single or complete linkage criteria. Fastcluster provides an algorithm to compute the hierarchy using average linkage criterion with $O(n^2)$ complexity.

Fastcluster takes condensed distance matrix and returns the hierarchy tree. The method of calculation of condensed distance matrix was explained in section 5.3.

The format of the output tree is different than a traditional binary tree and understanding it is useful for following next sections. In a hierarchy, it is not required to know if a child is right child or left child. Because of this relaxation it is possible to store this tree in less memory than a traditional binary tree. The storage system is further explained using an example.

If there are 4 points which need to be clustered.

$$points = [1, 2, 3, 4] \tag{5.10}$$

And their distance matrix is given by:
condensed distance matrix = [0.52, 0.10, 0.24, 0.62, 0.74, 0.18]

Then, the fastcluster and other clustering libraries such as scipy will return the following output

```
[[ 0.          2.          0.1         2.        ]
 [ 3.          4.          0.21        3.        ]
 [ 1.          5.          0.62666667  4.        ]]
```

The output should be read and executed line by line. First and second column of the output represent the index of the elements which should be clustered at this step. The result of the merge always creates a cluster labeled as number of last cluster + 1. The third column represents the distance between the two clusters while the last column represents the number of elements in that cluster. This output can be seen in the form of a dendogram in Figure 5.4.

In the first row of this example, elements at location 0 and location 2 in points array are clustered together to get the cluster no 4. The cluster is numbered four because there are four input points which have the label of 0, 1, 2 and 3. So, the first cluster will be labeled the next value in the sequence which is 4. Point at location 0 of array in expression 5.10 is 1, and point at location 2 is 3. As can be seen in dendogram points 1 and 3 are joined first. Distance between points 1 and 3 is 0.1 and there are two elements in newly created cluster which is labeled 4.

In the second row, element at location 3 and cluster 4 (created in last step) are merged together. Distance between them is 0.21 and the newly created cluster (5) has 3 elements in it. And finally in the last row element at location 1 is joined with cluster 5 to create cluster 6. Cluster 6 has all 4 elements in it.

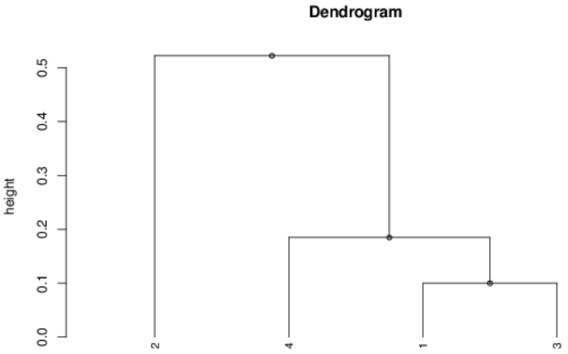

Figure 5.4: Dendogram of clustering. Image was drawn using the software [Wes12].

This format of hierarchy output is called “linkage matrix” and can be easily stored in a text file or database. It requires much less space as no references or pointers to children need to be maintained. And using the linkage matrix and the input data, dendogram can be easily created as explained above.

In MPG, linkage matrix is obtained using fastcluster library and it is stored in the hard disk to be used in next step.

5.5 Pattern Extraction

At this stage a hierarchy has already been obtained. Now, in this step, hierarchy will be followed as it is built to generate a binary tree. The parent of every two nodes of the tree will contain the events which are common between those two nodes. The process will be repeated until all the tree will be built. The process of generating the tree will be explained in section 5.5.1. Section 5.5.4 will explain the procedure of extracting the patterns out of the tree. From here onwards, this tree will be referred to as pattern tree.

5.5.1 Pattern tree generation

The sample events used in MPG are stored in files. The dendogram constructed in clustering step contains name of the event files as leaf nodes. The pattern tree generation module reads the linkage matrix generated by fastcluster line by line. Each line provides a connection between two existing event files or clusters. After reading the files which are linked, their intersection is taken to extract the events which are common between them. These common events are stored in a separate file and the name of the newly generated file is appended in a new column of linkage matrix. At the end of this process all the lines in linkage matrix contain not only the name of files which are linked but also the filename which contains the common events between them. This step converts the hierarchy to a binary tree with data in all nodes rather from just linkage information.

At the end of this step the linkage matrix for the example mentioned would look like this:

```
[[ 0.        2.        0.1         2.        node4.txt]
 [ 3.        4.        0.21        3.        node5.txt]
 [ 1.        5.        0.62666667  4.        node6.txt]]
```

And this can be shown in the form of a binary tree as shown in Figure 5.5. The structure of the pattern tree is same for all approaches but how parent of two nodes is populated differs from approach to approach.

MPG Single-Events and MPG Multiple-Events

For the case of MPG Single-Events and MPG Multiple-Events, simply an intersection of two nodes is taken to populate the parent node. That means if two nodes have identical events, they will move to parent nodes from child nodes. After the elements in intersection are copied to the parent node, they are removed from the child nodes.

MPG Fine

Since the procedure of clustering is different for MPG Fine (explained in section 5.3.1), the pattern tree generation is also slightly different.

1. Intersection of the nodes is taken to extract the identical events. The common events are put in parent node and removed from both child nodes.

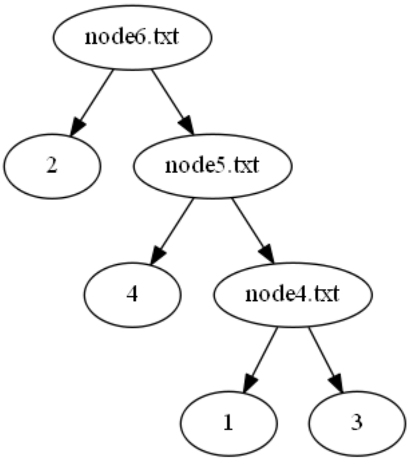

Figure 5.5: Pattern tree

2. The remaining events are sliced and aligned to each other using the algorithm mentioned in steps no 2, 3 and 4 of section 5.3.1. After aligning the events with each other, the intersection of each pair is taken to extract the properties which are common in both nodes. An event is then created out of only those common properties and is stored in the parent node. The complete events still exist in the child nodes.

Properties of pattern tree

For each variant, following the steps mentioned above, the pattern tree can be built. A pattern tree is a binary tree with each non-leaf node having two children. The leaf nodes of the tree represent the malware samples while all the nodes above leaf nodes are combination of events from multiple malware samples. A node contains events which are common in all of its descendant leaf nodes. The position of subtrees of any node (left and right) can be swapped without any loss of information. In fact, the tree does not store the location of its subtrees (left, right) at all in linkage matrix. In MPG Single-Events and MPG Multiple-Events, any event which is present in a node is not present in any of its descendant nodes. It saves the space by removing duplicating. In MPG Fine, if a node has a partial event, the complete event is present in the descendant nodes. If a parent node contains a complete event it is not present in the child nodes.

Furthermore, as shown above, the tree can be stored as a $5 \times (n-1)$ matrix. So, it can be easily stored in hard drive and re-read again without the need of object serialization. An example of pattern tree can be seen in Figure 5.5.

5.5.2 Pattern tree traversal

Since a pattern tree is stored in the form of a $5 \times (n - 1)$ matrix, a special algorithm is required to traverse it. To traverse a pattern tree following items are needed:

1. Number of leaf nodes
2. List of leaf nodes in the same order in which distance matrix was generated from them
3. Pattern tree

The last row of pattern tree refers to the root node of the pattern tree. To simplify the understanding of traversal, text below shows the list of leaf nodes and the pattern tree written together.

```
leafnodes:
      0:   [leaf1.txt]
      1:   [leaf2.txt]
      2:   [leaf3.txt]
      3:   [leaf4.txt]
tree:
      0:   [ 0.        2.        0.1         2.        node4.txt]
      1:   [ 3.        4.        0.21        3.        node5.txt]
      2:   [ 1.        5.        0.62666667  4.        node6.txt]
```

Algorithm 5.3 presents a depth first traversal algorithm for traversing the tree in above mentioned format. Similarly, a breadth first algorithm can be written. The algorithm starts at the root node and keeps traversing until all nodes have been visited.

5.5.3 False positive removal

After creation, the pattern tree contains all events from malware samples. Some of those events can also occur in the clean data set. To decrease the probability of false positives a step of false positive removal is performed.

A relatively small dataset of known benign files was created. Their software events were generated using Blue Coat's MAA product [Coa14]. The pattern tree is then traversed node by node and if all events present in any node are also present in the clean files, all events from that node are removed as a pattern consisting of

Algorithm 5.3 Pattern tree traversal

```
def traverse(node):
    if(node < 0):
        visit(leafnodes[node + len(leafnodes)])
    else:
        first_child = tree[node][0] - len(leafnodes)
        second_child = tree[node][1] - len(leafnodes)
        visit(tree[node][4])
        traverse(first_child)
        traverse(second_child)

traverse(len(tree) - 1)
```

those events will cause false positives. Section 5.5.4 will explain in detail that when a pattern group is extracted from a node, the relationship between different patterns in it is “and”. So, the nodes which have some events which cause FP but also some other events which do not cause FP are kept because altogether they will not cause an FP.

FPs are not removed before clustering because then some of the events which are present in clean files alone may not be present in any clean files when they are grouped with other such events, or when grouped with malicious events. Therefore, to increase the strength of pattern groups, FPs are removed at last possible position.

5.5.4 Pattern group extraction

To extract the pattern groups from the pattern tree, the tree is traversed breadth first. Usually the root node of the tree is empty and many top nodes also do not have any data in them. It is because, its very less likely that a malware dataset will contain some event which is common among all malware samples but it is not present in any of benign files. Due to this reason the tree is continued to be traversed from root in breadth first manner until a node is found which contains data.

A pattern group can have theoretically many patterns, but for MPG a limit of five patterns per pattern group was chosen. If the number of patterns in a pattern group are kept to a small number their strength decreases and the probability of FPs increases. However, if the number is kept too high then it takes relatively long time to match that pattern against malware which affects the system efficiency. So, a trade-off number of maximum five patterns per pattern group was chosen. Some further research can be done to fine tune the maximum number of patterns. A research topic covering this is proposed in the chapter of future work, section 7.3.

During breadth-first traversal, if a node is encountered which has more than five events, five events are chosen randomly to generate a pattern group. If five or less events are present, then all of them are used for pattern group generation. After generating a pattern group out of a node, the subtree of that node is ignored because the pattern group from that node will cover all its descendants. For the scope of this thesis no intelligence was added in choosing which of the five events should be chosen if there are more possibilities. Further research on this topic can be done as proposed in chapter of future work, section 7.3.

This process is repeated until all tree has been traversed and pattern groups are extracted.

5.6 Minimum patterns selector

All the pattern generated by MPG are FP free and useful for malware detection. However, for a specific dataset some pattern groups may remain un-used. This section presents a tool which can identify the minimum number of pattern groups needed to detect a specific dataset. This tool will be used in chapter 6 to present the minimum number of patterns needed to detect each dataset.

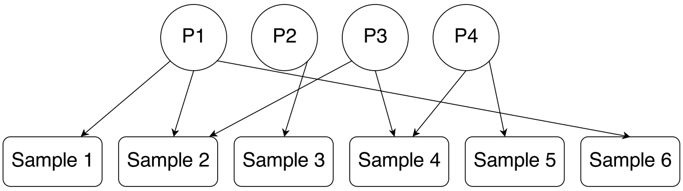

Figure 5.6: Example of a redundant pattern. P<x> represents a pattern, an arrow from a pattern to a sample represents that the pattern detects the sample.

For a given dataset of malware samples and a set of pattern groups, a sample may be detected by more than one pattern groups while a pattern group may detect more than one samples. Figure 5.6 shows an example of this many to many relationship between patterns and samples. In some cases it is possible to eliminate some of the pattern groups without loosing any detection. In order to select the minimum number of patterns which are needed, all generated patterns are matched against all samples in dataset to get a pattern to sample mapping as shown in Figure 5.6. Then, Algorithm 5.4 is used to select the minimum number of pattern groups needed to maintain the same detection.

In the example of Figure 5.6, in the first iteration P1 will get selected as it detects three samples (Sample 1, Sample 2 and Sample 6) while the other patterns detect

Algorithm 5.4 Minimum patterns selector algorithm

```
Create a sample to pattern map which can return
the number of patterns which detect a sample

Create a pattern to sample map which can return
the number of samples a pattern detects

Initialize all samples to unchecked state
Initialize all patterns to unchecked state

while(there are unchecked samples):
    best_pattern = From the pattern map, find an unchecked pattern
                   which detects maximum unchecked samples

    Check all the samples which are covered by best_pattern
    Check the best_pattern

Choose only checked patterns to get minimum\
patterns needed to maintain same detection rate
```

only two or one. And then Sample 1, Sample 2 and Sample 6 along with P1 will be checked. This can be seen in Figure 5.7.

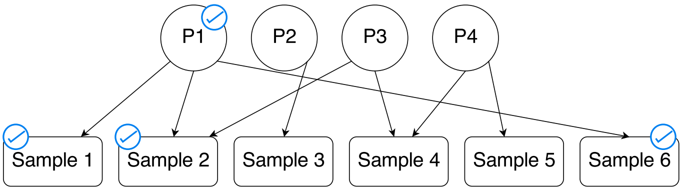

Figure 5.7: After first iteration of minimum patterns selector.

During second iteration P4 will be selected as it detects two unchecked samples while the other patterns detect only one. This iteration will lead to checking of P4, Sample 4 and Sample 5.

In the third iteration, P2 will be selected as it detects the last remaining Sample 3. This marks the end of while loop because all the samples have been checked and all patterns have also been checked except P3. P3 is left unchecked which means even without P3 same detection rate can be achieved.

5.7 Final FP Testing

After collecting all pattern groups a final FP test takes place. The step of FP removal (section 5.5.3) is very processing intensive. All clean events are matched against all malicious events. Therefore, the dataset that is used in that step is kept very small. However, at the end of all the above steps a larger dataset of clean files is used to perform a final FP test.

During final FP test if a pattern group detects a clean sample, it is removed from the patterns list. Furthermore, the clean sample which was detected by the pattern group is added to the initial FP set to remove that FP in initial step for future executions of MPG.

5.8 Summary

This chapter presented MPG: a tool to generate malware detection patterns automatically. There are three variants of MPG : MPG Single-Events, MPG Multiple-Events and MPG Fine. Design and implementation of all approaches is discussed in detail.

All malware samples are run in Blue Coat's MAA to generate their events. These events are then processed by MPG to generate pattern groups. MPG Single-Events slices the malware events based on their type and processes each of the slice independently. Each slice is hierarchically clustered. After the clustering, a pattern tree is generated and finally patterns are extracted from the pattern tree. At the end patterns from all slices are gathered together.

One of the limitations of MPG Single-Events was that each pattern group consisted of patterns of only a single type. MPG Multiple-Events removes this limitation by clustering all the events of all types together and generates pattern groups which can have multiple patterns of different types.

MPG Fine improves on MPG Multiple-Events further by not only taking into consideration identical events, but also the events which are partially identical. MPG Fine is most resource intensive followed by MPG Multiple-Events and then MPG Single-Events. After creation of the patterns FP testing takes place where the patterns which cause any false positives are removed.

Chapter 6 Evaluation

This chapter presents the evaluation results of the all variants of MPG. In section 6.1 the criteria which will be used to evaluate the results will be presented. Section 6.2 will explain the datasets which were used in experiments. Section 6.3 onwards will enlist and explain the results of each variant of MPG.

6.1 Criteria of Evaluation

The final goal of MPG is to automatically create the behavior patterns of malware which can be used in malware detection. The pattern groups generated by system will be evaluated based on criteria enlisted below in the order of their priority.

6.1.1 False Positives

Just like other signatures, behavior patterns are also prone to false positives. A pattern which can identify a malware may also identify a file which is non-malicious. The objective of MPG is to generate patterns for malware which can be used to identify it. The most basic use-case is: if a set of benign and malicious files are given to a system which uses MPG generated patterns as distinguisher, the malware should get detected as malware whereas the benign files should not get detected. So, a good pattern set should not contain any patterns which detect benign files or in other words, cause false positives. This requirement has the highest priority among all requirements.

6.1.2 Coverage

Coverage of dataset means what percentage of dataset samples were detected using the patterns generated. The dataset used to generate the patterns should be able to detect maximum number of samples from the dataset. The more the coverage, the better are the results. This requirement ensures that any malicious dataset can be given to MPG and the patterns generated will be enough to detect maximum number of samples from the dataset. In other words, the system should have least number of false negatives. This requirement has second priority after the false positives.

6.1.3 Sample to pattern ratio

Sample to pattern ratio (SPR) is given by Equation 6.1.

$$SPR = \frac{\text{No of Samples detected}}{\text{No of Patterns groups used to detect them}} \tag{6.1}$$

The process of matching a pattern group against datasets is an expensive process and affects the performance of a system which processes high number of malware. So, a situation where a small number of patterns can be used to identify a large number of malware is desirable. Therefore a measure of SPR is introduced here. The larger the value of SPR, the better the results are. This requirement has lower priority compared to both false positives and false negatives.

6.1.4 Matching speed

The time it takes to match the generated pattern groups against malicious samples has effect on efficiency of the system. The patterns which are faster to match will result in an efficient system. This will be measured by noting down the matching time when matching all MPG generated pattern groups against a dataset. Within the scope of this thesis, this criterion has lowest priority.

6.2 Datasets

MPG requires that all the samples in malicious dataset should be known malware. If some of the files in the dataset will be non-malicious, they will result in producing pattern groups which detect those non-malicious files. Such patterns will cause false positives. Secondly, some of the malware samples do not perform any activities in virtual environments. There can be many reasons for that, such as: the malware detects that it is being run in a virtual environment for detection and it does not execute. There are some malware samples which require a certain condition to be true before they can execute. Examples of such malware are: malware samples that run only on a specific date, or a malware sample that executes only on a Windows XP machine furthermore, only if a specific version of Adobe Reader is installed on it. If such malware is executed in a virtual machine, it does not perform any activities hence its detection using pattern matching is not possible. And detection of such malware is out of scope of this thesis. Therefore, the datasets chosen only consist of the malware which produces events. Based on this criteria, the datasets have following properties:

1. All samples are malicious, it is confirmed using Blue Coat's malware database.
2. All samples are unique. This is confirmed by making sure no two samples have same md5 hash.
3. All samples were executed in Blue Coat's MAA [Coa14]. Samples were executed on Microsoft Windows XP Service Pack 3 profile of MAA for 1 minute each to generate events from samples. Microsoft Windows XP was chosen because XP lacks User Account Control (UAC) [Rus07] which makes it less secure than later version of Windows.

Three datasets following the properties mentioned above were selected. Dataset 1 and Dataset 2 were chosen to generate the pattern groups. Dataset 3 was chosen specifically to see how the patterns generated by previous datasets help in detecting larger set of malware. These datasets are analogous to training and evaluation datasets used in machine learning. Dataset 1 and Dataset 2 are training datasets while Dataset 3 is evaluation dataset.

Apart form this, a separate dataset named Clean Dataset of non-malicious files was used to remove any patterns which cause false positives.

6.2.1 Dataset 1

Dataset 1 consists of 10000 samples chosen according to criteria mentioned above. The dataset contains 8609 samples which have events in them while remaining samples have no events which made their detection through patterns impossible. Therefore, the samples with no events are discarded and tests are performed using only 8609 samples.

6.2.2 Dataset 2

Dataset 2 consists of 1000 malware with an already known pattern. The patterns for that dataset were created manually by human analysts. This dataset helps in comparing how MPG competes with manually generated patterns. To detect this Dataset 27 human generated pattern groups were required. All samples in this dataset have events and there are no silent samples.

6.2.3 Dataset 3

For selection of Dataset 3, 30000 malware samples were chosen according to criteria mentioned above. Out of the 30,000 samples 3746 were silent which were discarded. Remaining 26254 samples were used for experiments. This dataset is not used for creating pattern groups but rather evaluating the pattern groups generated by other datasets to see how effective they are for a different dataset.

6.2.4 Clean Dataset

Clean Dataset consists of 74530 samples, all files are known to be clean and do not contain any malicious samples. This dataset is used to make sure that any of the malware patterns do not detect any of non-malicious files. The number of samples in this dataset are chosen to be much larger than other datasets because avoiding false positives is highest priority of MPG.

In the sections below each variant of MPG will be evaluated first against Dataset 1. During this evaluation, Dataset 1 will be used to generate a set of behavior patterns. These behavior patterns will then be matched against Dataset 1 to get the SPR and detection rate. Finally, the patterns generated by Dataset 1 will be matched against Dataset 3 to see how these patterns perform against a dataset which is different from the one used to generate patterns.

The second experiment for each variant of MPG consists of evaluating the variant against Dataset 2. Since, Dataset 2 has samples with known patterns generated by human analysts, it is used to compare the performance of MPG patterns with manual patterns. First a set of pattern groups will be generated using Dataset 2, then those pattern groups will be evaluated against Dataset 2 to measure the detection rate and finally, the pattern groups will be evaluated against Dataset 3. A brief comparison of MPG pattern groups versus human generated pattern groups will also be given.

6.3 MPG Multiple-Events

MPG Multiple-Events produces pattern groups which can have multiple patterns in each group. And all patterns can be of different types.

6.3.1 Experiment 1

During this experiment, pattern groups were generated by MPG Multiple-Events using Dataset 1. Table 6.1 shows that 3643 pattern groups were generated from this malicious data set.

Table 6.1: Experiment 1: Pattern groups generated by MPG Multiple-Events using Dataset 1

Total samples	8609
Patterns generated	3643

Each of these pattern groups contain up to five patterns. Figure 6.1 shows that 1290 (35.4%) of the pattern groups had only one pattern in them while 1719 (47.2%) had five patterns in them. Rest of the pattern groups have either 2, 3 or 4 patterns which make up 17% of the whole pattern set. The pattern groups with higher number of patterns are stronger and are less FP prone.

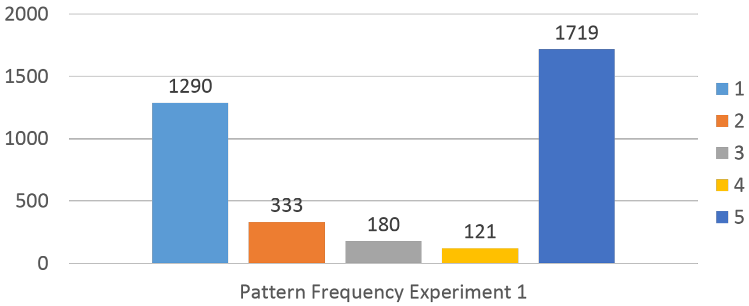

Figure 6.1: Experiment 1: Frequency of number of patterns in pattern groups

In the second part of the experiment, the patterns generated in Table 6.1 were matched on the same dataset (Dataset 1). Results of this experiment are presented in Table 6.2. It can be seen that out of 3643 patterns generated only 3146 patterns were used for detection. Even if the rest of them (497) were removed, detection rate would still be the same. These 3146 patterns detected 8606 out of 8609 files giving a detection rate of 99.96%. The 3 samples which were not detected did not have any events left after events causing false positives were removed. From the results, it can be seen that it happens very rarely that all the events in a malicious sample are causing false positives. No false positives were encountered when these patterns were matched against Clean Dataset.

Table 6.2: Experiment 1: Evaluation of MPG Multiple-Events on Dataset 1

Total samples	8609
Total patterns	3643
Patterns used in detection of this dataset	3146
Samples detected	8606
Sample to pattern ratio (SPR)	2.735
Detection Rate	99.96%
False Positives	0
FP Rate	0%

A pattern set of 3146 pattern groups detecting 8606 samples gives a sample to pattern ratio of 2.73 which means on average each pattern detects 2.73 malware samples. However, the average does not give the complete picture of relationship between patterns and samples. Figure 6.2 shows the number of patterns needed to detect the number of samples. Curve rises slowly for the first half of sample set. To detect first 2000 samples only 8 patterns are needed. To detect first 4000 samples the patterns needed are around 50. And around 50% of the sample set can be covered by approximately 72 (2.3%) of the patterns. It can also be seen in the curve that after around 6000 samples the curve becomes more or less linear and each pattern detects only a single sample. For the last 30% (after 6000 mark) of samples approximately 84% of pattern groups are needed.

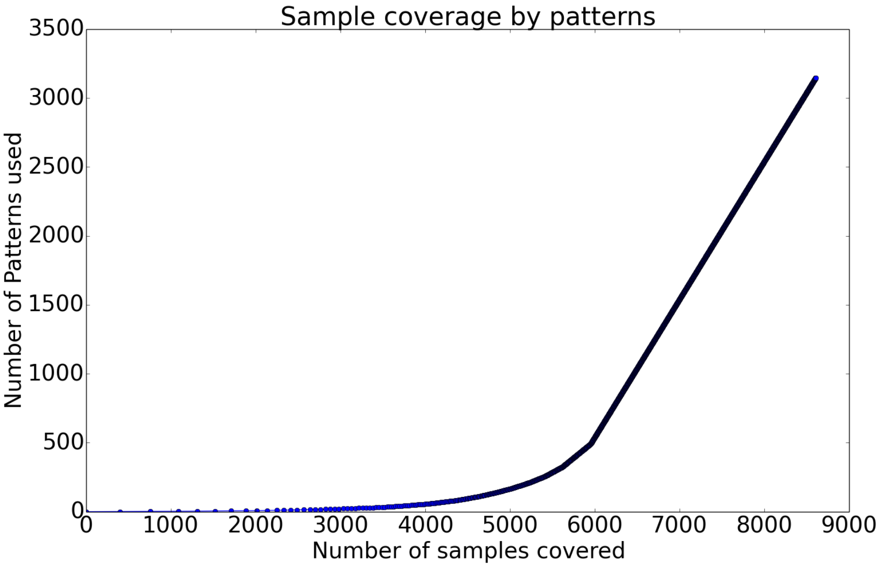

Figure 6.2: Experiment 1: Dataset coverage by number of patterns

The trend explained in Figure 6.2 shows that majority of samples in a randomly selected dataset have events in common. These events can be used to generate patterns for them. However, a portion of dataset does not have any events in common therefore, to detect those samples a separate pattern group must be generated for each of those samples. And finally these results show that if a set of malware samples are used to generate malicious patterns, it is indeed possible to generate false positive free pattern groups which detect almost all the samples.

Table 6.3 shows the matching results of the pattern groups generated in Experiment 1 against Dataset 3. Dataset 3 consists of 30,000 samples chosen randomly to check the quality of generated pattern groups against a dataset which was not used in pattern generation. From a total of 3643 pattern groups 3135 pattern groups were used to detect 16596 (63.21%) of dataset. Since, only 8,609 samples were used to generate the pattern groups, detection of 16596 samples shows that generated

Table 6.3: Evaluation of patterns generated during Experiment 1 on Dataset 3

Total samples	26254
Total patterns	3643
Patterns used in detection of this dataset	3135
Samples detected	16596
Sample to pattern ratio (SPR)	5.294
Detection Rate	63.21%
False Positives	0
FP Rate	0%

pattern groups indeed detect not only the samples in training set but also other samples.

6.3.2 Experiment 2

Experiment 2 is replica of Experiment 1 with a different dataset. During this experiment Dataset 2 was used to evaluate MPG Multiple-Events. All the samples chosen in this dataset had at-least one event. Therefore, there are no silent events. Furthermore, malware analysts at Blue Coat have already created patterns for these samples. During this experiment, the pattern groups generated by MPG Multiple-Events will be compared with analyst generated behavior pattern groups.

Table 6.4 shows that Dataset 2 consisted of 1000 samples and 302 pattern groups were generated from this dataset.

Table 6.4: Experiment 2: Pattern groups generated by MPG Multiple-Events using Dataset 2

Total samples	1000
Patterns generated	302

Figure 6.3 shows that majority (89.5%) of the pattern groups for this dataset had 5 patterns in them. All other pattern groups had 1, 2, 3 or 4 pattern groups. It shows that most of pattern groups generated during this test are strong pattern groups because of having 5 patterns each. Therefore, the probability of them causing FPs is lower. The reason for this is the nature of Dataset 2. As opposed to Dataset 1 which consists of random samples, Dataset 2 consists of samples which already have human generated pattern groups. Because the selection of samples was not random, the selected samples have more common events than general dataset such as Dataset 1. Therefore, the pattern groups generated also have more patterns as can be seen in Figure 6.3.

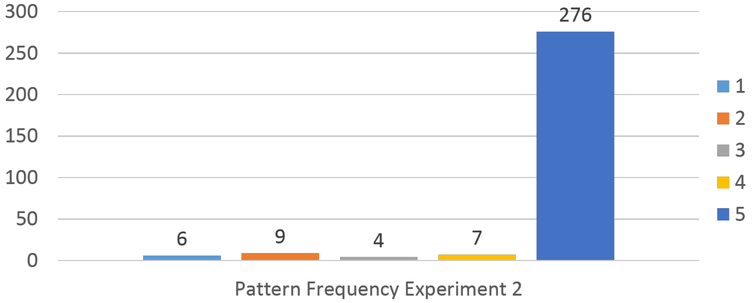

Figure 6.3: Experiment 2: Frequency of number of patterns in pattern groups

Similar to Experiment 1, patterns generated from Dataset 2 which are shown in Table 6.4 are matched against Dataset 2. The matching results are shown in Table 6.5. For this particular dataset only 253 pattern groups were used and detection rate was 100% with 0% FP rate.

Table 6.5: Experiment 2: Evaluation of MPG Multiple-Events on Dataset 2

Total samples	1000
Total patterns	302
Patterns used in detection of this dataset	253
Samples detected	1000
Sample to pattern ratio (SPR)	3.952
Detection Rate	100%
False Positives	0
FP Rate	0%

Similar to Experiment 1, the SPR of 3.952 in this case is also not smooth. The relationship between number of pattern groups and number of samples it covers for this experiment is shown in Figure 6.4. The curve is again non-linear with 19 (5.82%) pattern groups detecting 50% of sample set and curve converts into a straight line at 772 samples mark. The last 228 samples (22.8%) require 228 (70%) of pattern groups which means one pattern group for each sample.

In the final part of this experiment the patterns generated during this experiment are matched against Dataset 3. Table 6.6 shows the results.

It can be seen in Table 6.6 that out of 302 patterns, 251 were used to detect 4506 samples. The detection rate for the complete dataset is 17.16%. However, considering that only 1000 samples were used to generate these pattern groups and 4506 samples

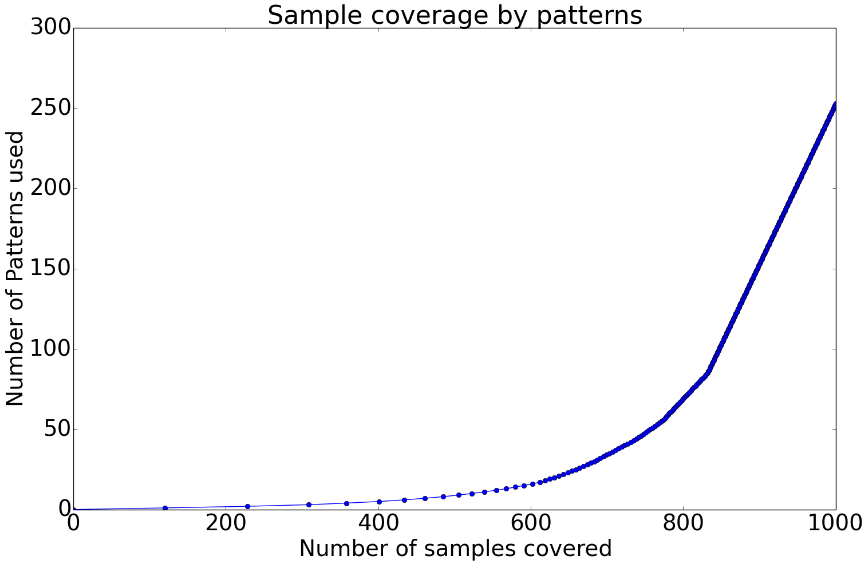

Figure 6.4: Experiment 2: Dataset coverage by number of patterns

Table 6.6: Evaluation of patterns generated during Experiment 2 on Dataset 3

Total samples	26254
Total patterns	302
Patterns used in detection of this dataset	251
Samples detected	4506
Sample to pattern ratio (SPR)	17.952
Detection Rate	17.163%
False Positives	0
FP Rate	0%

from a different dataset get detected shows that the pattern groups generated are not specific only to dataset used to generate them.

Comparison with human created pattern groups

This section provides a brief comparison of pattern groups created by MPG Multiple-Events using Dataset 2 with patterns which were created by human malware analysts at Blue Coat Norway.

Both human generated pattern groups and MPG Multiple-Events generated pattern groups were able to detect 100% of Dataset 2. Furthermore, both human generated and MPG Multiple-Events generated pattern groups had no false positives. To detect all samples of Dataset 2, only 27 human created pattern groups were enough. Whereas, 253 pattern groups generated by MPG were required for same detection rate.

The pattern groups created by malware analysts were much complicated then the ones created by MPG. In MPG all patterns in a pattern group have *and* (intersection) relationship with each other, meaning that all the patterns must occur in a sample for it to be detected. Whereas, analyst created patterns had sometimes *or* (union) relationship which makes it difficult to compare them side by side with MPG pattern groups. Furthermore, regular expressions and partial sub-patterns are also often used by malware analysts which is not the case in MPG.

MPG provides same detection and false positive rate as human generated pattern groups. However, there is still much room for improvement in MPG Multiple-Events by using regular expressions and *or* (union) relationship among pattern groups to find patterns with better SPR without loosing detection. This is added as a future work proposal in chapter 7.

MPG excels in providing most important pattern groups for thousands of malware samples in very small period of time (hours). It will take months for human malware analysts to generate pattern groups for such a dataset manually. However, human malware analysts can use intelligence to select very concise pattern groups. If both approaches are combined and human malware analysts see the generated pattern groups from MPG and improve on them while inserting that intelligence in MPG can provide very valuable results.

To conclude, MPG produces pattern groups which are capable of same detection rate and false positive rate as human generated pattern groups. Furthermore, it takes much less time to generate pattern groups using MPG compared to using human malware analysts. And finally, MPG tends to generate more pattern groups than human malware analysts.

6.4 MPG Single-Events

In this section MPG Single-Events will be evaluated. The MPG Single-Events takes the events of a malicious dataset as input and breaks the events into slices based on their type. Then, events of each type are processed independently to get pattern groups. All patterns in a pattern group generated by MPG Single-Events are of the same type. Due to slicing, events of each sample gets sliced into many slices and each sample gets a different pattern group from each slice. Due to this, the number of patterns generated becomes very large. To reduce the amount of pattern groups, only the pattern groups which cover more than one sample are chosen.

6.4.1 Experiment 3

In this experiment the MPG Single-Events is evaluated on Dataset 1. Table 6.7 shows that 8609 samples in Dataset 1 were used to generated 4757 pattern groups. All the collected pattern groups detect at least two samples.

Table 6.7: Experiment 3: Pattern groups generated by MPG Single-Events using Dataset 1

Total samples	8609
Patterns generated	4757

The frequency of these patterns is shown in Figure 6.5. 2754 (57.89%) of pattern groups have only a single pattern while 876 (18.41%) pattern groups have 2 patterns in them. this shows that majority of the pattern groups generated by MPG Single-Events are statistically weaker and have higher chances of false positives.

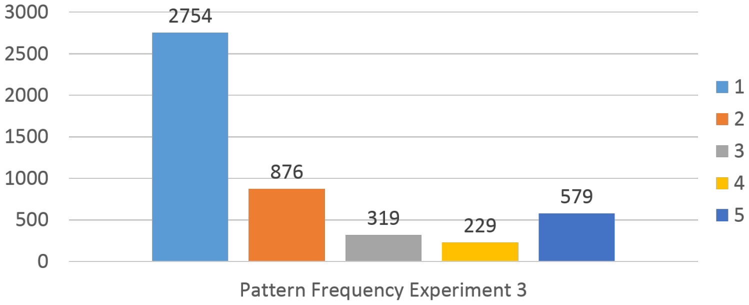

Figure 6.5: Experiment 3: Frequency of number of patterns in pattern groups

Afterwards, the pattern groups shown in Table 6.7 are evaluated against Dataset 1. Results for this evaluation are shown in Table 6.8. Only the pattern groups which detect two or more samples were chosen in this case, therefore only 75.07% of the dataset was detected. But the number of patterns required were much lesser (668) with a SPR of 9.675.

Number of samples covered by number of pattern groups is shown in Figure 6.6. This curve look exponential until approximately 6150 samples mark. After which the curve becomes linear. Even in the linear part patterns detect approximately two samples for each pattern group. And because of loss of approximately 25% detection

Table 6.8: Experiment 3: Evaluation of MPG Single-Events on Dataset 1

Total samples	8609
Total patterns	4757
Patterns used in detection of this dataset	668
Samples detected	6463
Sample to pattern ratio (SPR)	9.675
Detection Rate	75.07%
False Positives	0
FP Rate	0%

the linear part (1 pattern group for each sample) of the curve, which was observed in Experiment 1 and Experiment 2 is missing from this curve.

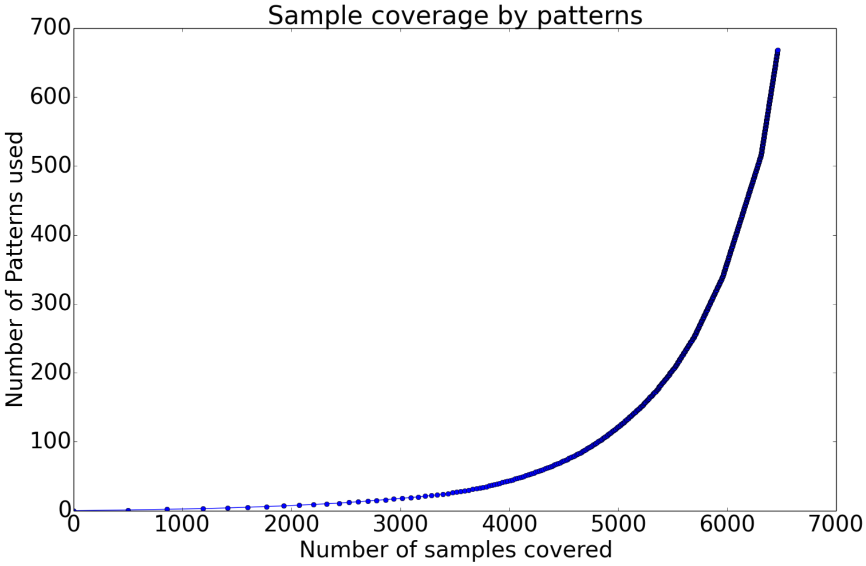

Figure 6.6: Experiment 3: Dataset coverage by number of patterns

In the final step of this experiment the generated pattern groups were matched against Dataset 3. The results are shown in Table 6.9. Out of 4757 possible pattern groups, 759 were used to detect 16334 samples providing a detection rate of 62.2%. The trend again shows that pattern groups are capable of detecting samples other than the samples used to generate them.

Table 6.9: Evaluation of patterns generated during Experiment 3 on Dataset 3

Total samples	26254
Total patterns	4757
Patterns used in detection of this dataset	759
Samples detected	16334
Sample to pattern ratio (SPR)	21.52
Detection Rate	62.215%
False Positives	0
FP Rate	0%

6.4.2 Experiment 4

In this experiment, MPG Single-Events was evaluated using Dataset 2. From the 1000 samples of Dataset 2, MPG Single-Events generated 1070 pattern groups as shown in Table 6.10.

Table 6.10: Experiment 4: Pattern groups generated by MPG Single-Events using Dataset 2

Total samples	1000
Patterns generated	1070

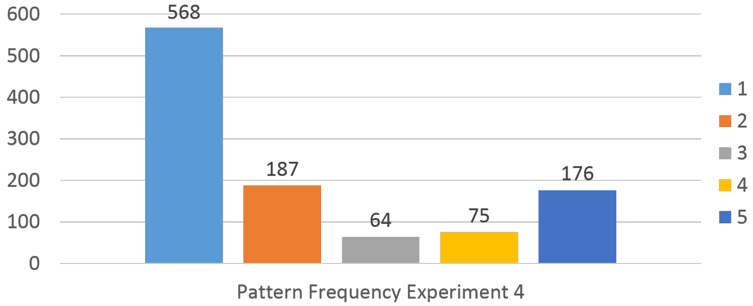

Figure 6.7: Experiment 4: Frequency of number of patterns in pattern groups

The frequency of these pattern groups is shown in Figure 6.7. Frequency of pattern groups from this experiment is similar to frequency of pattern groups from Experiment 3 shown in Figure 6.5. 568 (53.08%) pattern groups have only one pattern present in them and 187 (17.47%) patten groups have 2 patterns. The rest (29.4%) of pattern groups have more than two pattern groups. It shows that majority of pattern groups generated by MPG Single-Events have one or two patterns.

In the next step of this experiment, patterns were evaluated against Dataset 2. Out of 1070 pattern groups only 115 were used to detect 939 samples out of total 1000 samples. The matching percentage is again less than 100% (93.9%) percent. However, detection improved on Dataset 2 compared to Dataset 1 which was 75.07%. The reason of improvement is the same again: since, Dataset 2 contains of malware samples which have more malicious events, it was possible to generate more pattern groups which detect two samples each. Results can be seen in Table 6.11.

Table 6.11: Experiment 4: Evaluation of MPG Single-Events on Dataset 2

Total samples	1000
Total patterns	1070
Patterns used in detection of this dataset	115
Samples detected	939
Sample to pattern ratio (SPR)	8.16
Detection Rate	93.9%
False Positives	0
FP Rate	0%

A total of 115 pattern groups generated by MPG Single-Events were used to detect 93.9% of the dataset. However, only 27 pattern groups generated by human analysts were enough for 100% detection. Human generated pattern groups provide better detection rate than patterns generated by MPG Single-Events but false positive rate is same for both cases. As explained in section 6.3.2, the human generated pattern groups are more complicated than MPG patterns which makes the direct comparison difficult.

The number of patterns needed to detect a certain percentage of dataset is again non-linear as can be seen in Figure 6.8. The curve rises slowly in the start with rise becoming steeper with increase in number of samples. The last part of the curve is linear with each pattern group covering two samples as was the case in Experiment 3.

Table 6.12 shows the matching results of pattern generated in Experiment 4 against Dataset 3. 205 out of 1070 pattern groups were used to detect 9043 samples. The overall detection rate of dataset is 34.44%.

6.5 MPG Fine

MPG Fine is an extension of MPG Multiple-Events. MPG Multiple-Events take into consideration all events and performs clustering using Jaccard distance. The Jaccard distance is calculated by taking intersection and union of complete events produced by samples. MPG Fine extends on this idea by taking into consideration

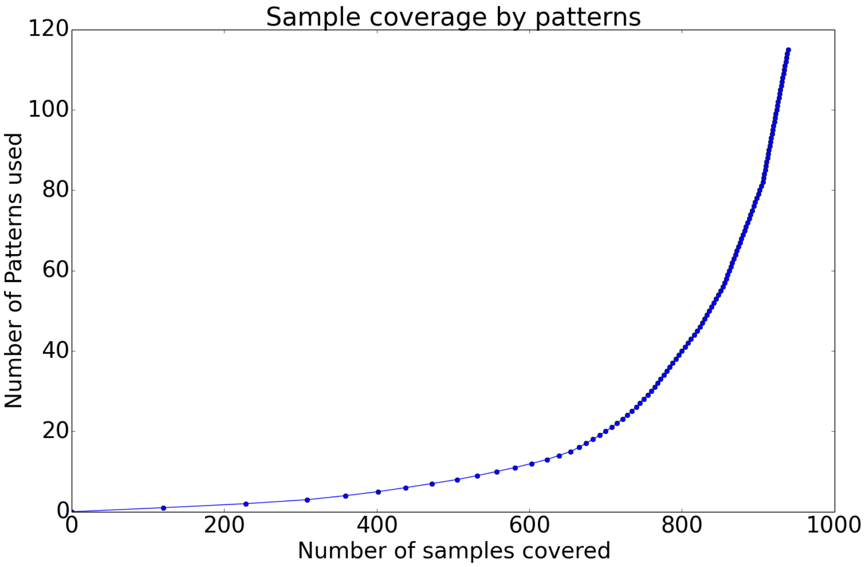

Figure 6.8: Experiment 4: Dataset coverage by number of patterns

Table 6.12: Evaluation of patterns generated during Experiment 4 on Dataset 3

Total samples	26254
Total patterns	1070
Patterns used in detection of this dataset	205
Samples detected	9043
Sample to pattern ratio (SPR)	128.068
Detection Rate	34.444%
False Positives	0
FP Rate	0%

also properties of events to get finer results. In the sections below first MPG Fine will be evaluated against Dataset 1 followed by evaluation against Dataset 2.

6.5.1 Experiment 5

In this experiment the MPG Fine is evaluated on Dataset 1. Table 6.13 shows that 3411 pattern groups are generated from 8609 samples of Dataset 1.

The frequency of these patterns is shown in Figure 6.9. 1500 (43.97%) pattern

Table 6.13: Experiment 5: Pattern groups generated by MPG Fine using Dataset 1

Total samples	8609
Patterns generated	3411

groups have five patterns present in them while 1267 (37.14%) pattern groups have only one pattern in them. The frequency chart shows that considerable amount of pattern groups have five or four patterns implying statistically they have less chances of false positives.

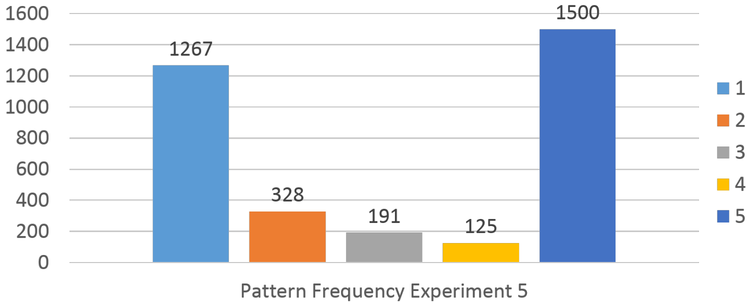

Figure 6.9: Experiment 5: Frequency of number of patterns in pattern groups

The patterns generated are then matched against Dataset 1. Results of matching can be seen in Table 6.14. Out of 3411 total pattern groups, 2593 were used to detect 8608 samples giving a matching percentage of 99.98%. The SPR of the matching is 3.32 i.e. on average each pattern group detects 3.32 samples.

Table 6.14: Experiment 5: Evaluation of MPG Fine on Dataset 1

Total samples	8609
Total patterns	3411
Patterns used in detection of this dataset	2593
Samples detected	8608
Sample to pattern ratio (SPR)	3.32
Detection Rate	99.98%
False Positives	0
FP Rate	0%

The number of patterns needed to detect a certain percentage of dataset for this experiment can be seen in Figure 6.10. The curve rises slowly in start and then converts into a linear curve just like previous experiments. Approximately last 2000 samples require 2000 pattern groups for detection.

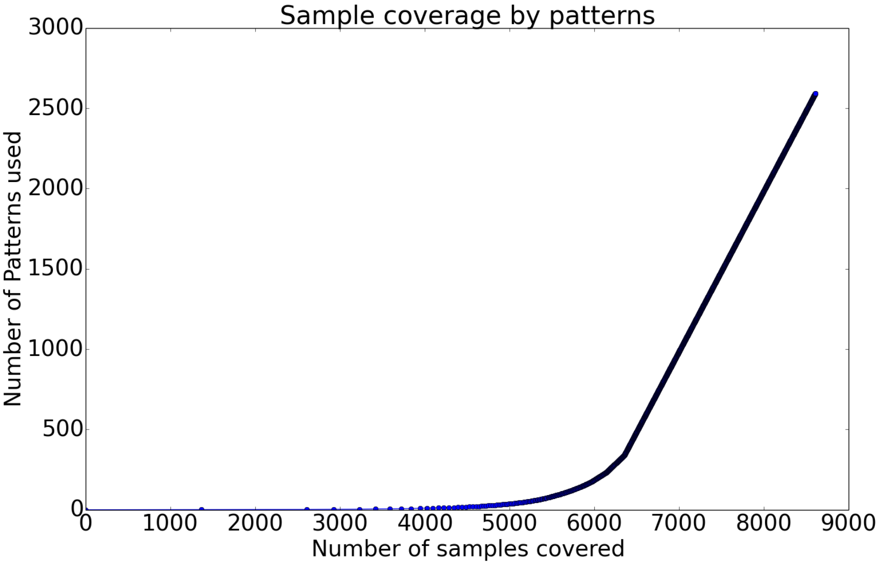

Figure 6.10: Experiment 5: Dataset coverage by number of patterns

In the final step of this experiment the patterns generated were matched against Dataset 3. Out of 3411 pattern groups 2606 pattern groups were used to detect 17875 samples which is 68.085% of the dataset as shown in Table 6.15.

Table 6.15: Evaluation of patterns generated during Experiment 5 on Dataset 3

Total samples	26254
Total patterns	3411
Patterns used in detection of this dataset	2606
Samples detected	17875
Sample to pattern ratio (SPR)	6.859
Detection Rate	68.085%
False Positives	0
FP Rate	0%

6.5.2 Experiment 6

Experiment 6 is the last of experiments presented in this chapter. In this experiment, MPG Fine is evaluated against Dataset 2. From 1000 samples, 249 pattern groups were generated as shown in Table 6.16.

Table 6.16: Experiment 6: Pattern groups generated by MPG Fine using Dataset 2

Total samples	1000
Patterns generated	249

Frequency chart in Figure 6.11 shows that majority of the pattern groups (231, 92.77%) have 5 patterns in them. This implies that most of patterns generated by MPG Multiple-Events are statistically strong patterns.

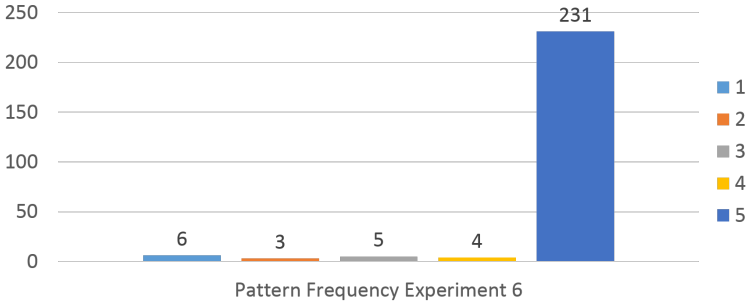

Figure 6.11: Experiment 6: Frequency of number of patterns in pattern groups

Similar to previous experiments, the patterns were then evaluated against Dataset 2. Only 118 out of 249 patterns were enough to detect 100% of the dataset. The SPR for this matching is 8.47. These results can be seen in Table 6.17.

Table 6.17: Experiment 6: Evaluation of MPG Fine on Dataset 2

Total samples	1000
Total patterns	249
Patterns used in detection of this dataset	118
Samples detected	1000
Sample to pattern ratio (SPR)	8.475
Detection Rate	100%
False Positives	0
FP Rate	0%

The samples covered vs patterns used curve in Figure 6.12 is similar to previous experiments. The curve rises slowly in start and finally becomes linear.

The evaluation of these pattern groups against Dataset 3 is shown in Table 6.18. Out of 249 pattern groups 122 were used to detect 6975 (26.546%) of the dataset. Because of low number of patterns the SPR is 57.172.

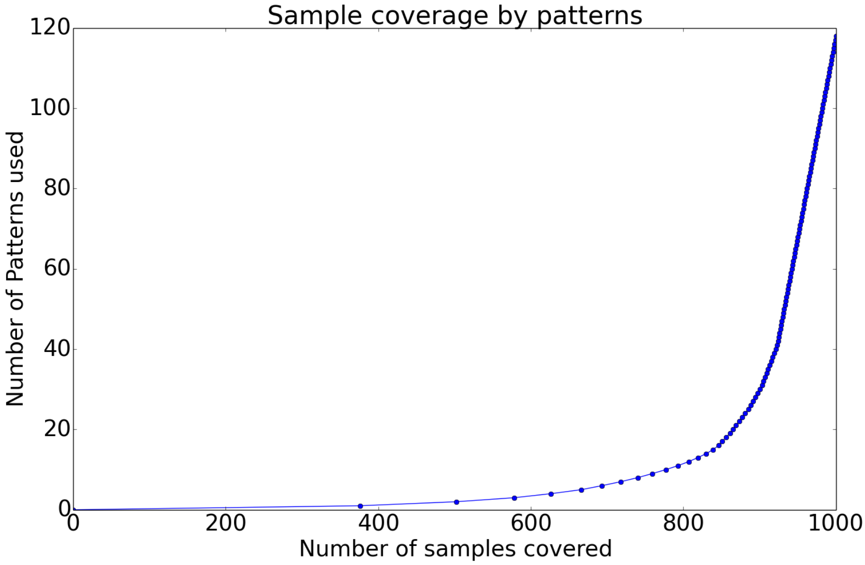

Figure 6.12: Experiment 6: Dataset coverage by number of patterns

Table 6.18: Evaluation of patterns generated during Experiment 6 on Dataset 3

Total samples	26254
Total patterns	249
Patterns used in detection of this dataset	122
Samples detected	6975
Sample to pattern ratio (SPR)	57.172
Detection Rate	26.567%
False Positives	0
FP Rate	0%

6.6 Conclusion of Experiments

Table 6.19 shows a summary and comparison of all MPGs when run using Dataset 1. The matching results of patterns generated by Dataset 1 against Dataset 3 are also shown. Furthermore, the time it took to match the patterns against Dataset 3 are also included to compare the matching speed of the final patterns.

Because of lack of space in tables, DS1 will be used as abbreviation of Dataset 1, DS2 as abbreviation of Dataset 2, DS3 as abbreviation of Dataset 3, MPG Single as abbreviation of MPG Single-Events and MPG Multiple as abbreviation of MPG Multiple-Events.

Table 6.19: Comparison of MPG evaluation against Dataset 1

	MPG Single	MPG Multiple	MPG Fine
Total patterns	4757	3643	3411
Patterns used in detection of DS1	668	3146	2593
Samples detected DS1	6463	8606	8608
SPR DS1	9.68	2.73	3.32
Detection Rate DS1	75.07%	99.96%	99.98%
Patterns used in detection of DS3	759	3135	2606
Samples detected DS3	16334	16596	17875
SPR DS3	21.52	5.29	6.859
Detection Rate DS3	62.21%	63.21%	68.08%
FPs according to Clean Dataset	0	0	0
Matching time DS3	1 day, 7:34:45	2 days, 9:02:39	11:13:35

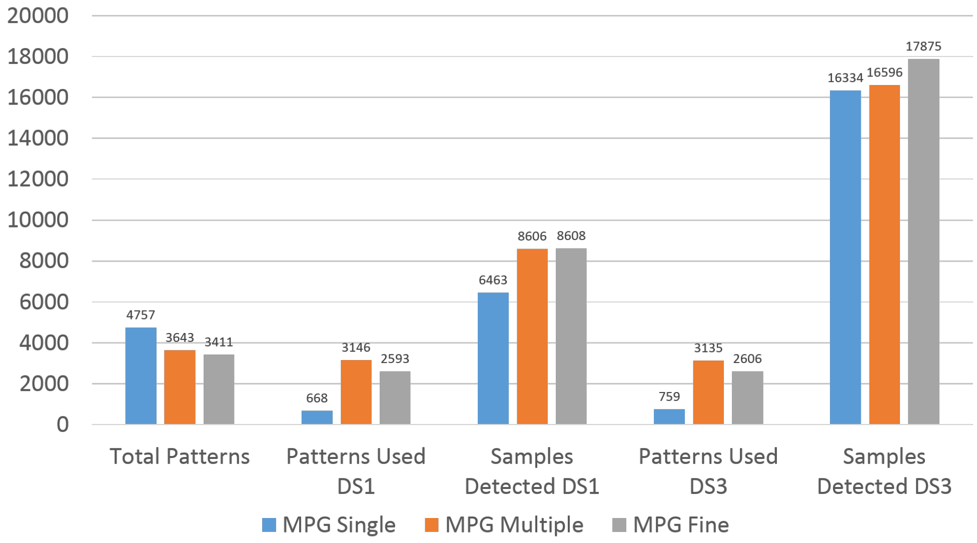

Figure 6.13: Comparison of MPG evaluation against Dataset 1

It can be seen in Table 6.19 and Figure 6.13 that MPG Single-Events has highest number of pattern groups despite taking only the pattern groups which detect at least 2 samples followed by MPG Multiple-Events and finally MPG Fine. However, the number of patterns which were actually used in detection of Dataset 1 were highest for MPG Multiple-Events followed by MPG Fine and finally MPG Single-Events. Although the number of patterns used are lowest for MPG Single-Events, but if they are correlated with the detection percentage of Dataset 1 (75.07%) it is not extraordinary as the detection of last 25% of samples takes most of pattern groups as shown in Figure 6.10 and other similar curves. The detection percentage of Dataset 1 is highest for MPG Fine followed by MPG Multiple-Events and finally MPG Single-Events.

The matching results with Dataset 3 in Table 6.19 show that MPG Multiple-Events uses most pattern groups followed by MPG Fine and finally MPG Single-Events. The detection percentage is highest for MPG Fine followed by MPG Multiple-Events and then MPG Single-Events. However, if number of patterns used and detection rate is seen together, it can be seen that MPG Single-Events uses far less pattern groups than MPG Multiple-Events without loosing too much detection. MPG Single-Events uses 2376 less pattern groups however detects only 262 samples less than MPG Multiple-Events. Similarly, MPG Single-Events has a better SPR than MPG Fine but the margin is lesser. MPG Single-Events uses 1875 less pattern groups but detects 1541 less samples.

It can also be seen in the table that MPG Fine is fastest when it comes to matching the pattern groups against DS3. It is mostly because the pattern groups are simpler than MPG Multiple-Events. The number of overall patterns are lowest for MPG Fine compared to other MPGs.

The evaluation of these results according to criteria mentioned in section 6.1 is as follows. All MPG variants have no false positives, secondly MPG Fine has highest coverage in both Dataset 1 and Dataset 3. And finally MPG Single-Events has highest SPR. But according to the importance of criteria, MPG Fine outperforms the MPG Multiple-Events completely and outperforms MPG Single-Events by coverage which is more important than SPR.

Table 6.20 and Figure 6.14 summarizes results of all MPG variants when tested against Dataset 2. Number of pattern groups generated using Dataset 2 followed by their matching statistics against both Dataset 2 and Dataset 3 are given. MPG Single-Events produces the most number of pattern groups (1070) followed by MPG Multiple-Events (302) and finally MPG Fine (249). For Dataset 2 both MPG Multiple-Events and MPG Fine provide 100% detection whereas MPG Single-Events provides 93.90% detection. The number of pattern groups used are highest for MPG Multiple-Events while MPG Single-Events and MPG Fine are almost the same.

However, the trend changes when matching results against Dataset 3 are analyzed. MPG Single-Events provides the best detection of Dataset 3 and also provides

Table 6.20: Comparison of MPG evaluation against Dataset 2

	MPG Single	MPG Multiple	MPG Fine
Total patterns	1070	302	249
Patterns used in detection of DS2	115	253	118
Samples detected DS2	939	1000	1000
SPR DS2	8.16	3.95	8.47
Detection Rate DS2	93.90%	100%	100%
Patterns used in detection of DS3	205	251	122
Samples detected DS3	9043	4506	6975
SPR DS3	128.07	17.95	57.17
Detection Rate DS3	34.44%	17.16%	26.57%
FPs according to Clean Dataset	0	0	0
Matching time DS3	6:05:17	8:22:40	1:12:52

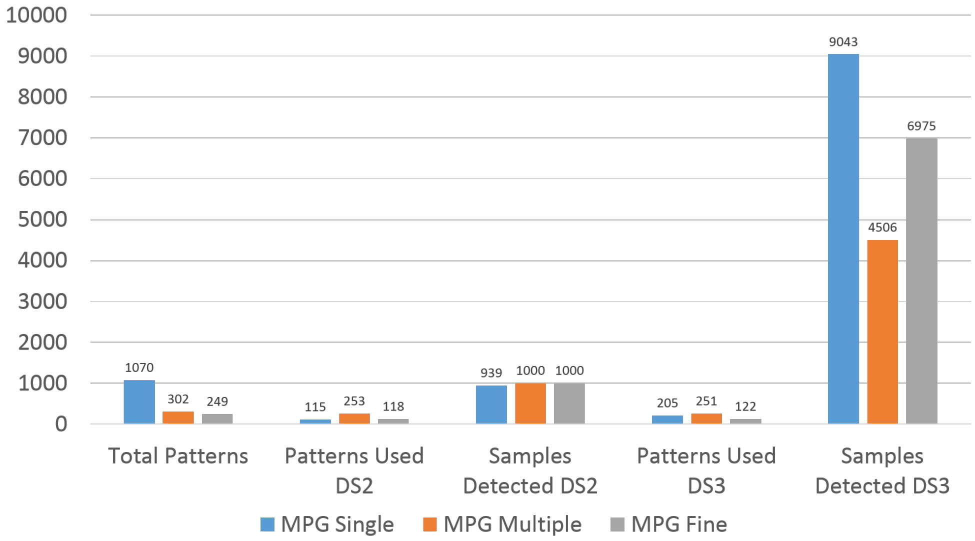

Figure 6.14: Comparison of MPG evaluation against Dataset 2

the highest SPR. The detection percentage is comparatively low but taking into consideration that patterns were only generated using 1000 samples, the detection seems significant.

The above results show that MPG Fine performs better than other variants regardless of dataset used. The patterns generated are low in number and provide good detection. However, MPG Single-Events perform better if the training dataset consists of malware samples which have easily extractable pattern groups (All samples of Dataset 2 have already known pattern groups generated by human analysts). Finally, MPG Fine again takes lowest matching time when matching patterns against Dataset 3.

MPG Fine is also, closest to human generated pattern groups because it provides 100% coverage with only 118 pattern groups. However, it is still far from 27 human generated pattern groups which shows there is still room for improvement.

6.7 Summary

In this chapter evaluation of all three variants of MPG against real-world datasets is presented. In the start of the chapter the criteria of evaluation of results was given i.e. Zero FPs, low False Negatives (FNs) and maximum coverage of the dataset.

In section 6.2 the criteria for selecting a dataset and details of each dataset were presented. A total of three datasets were used. Dataset 1 contained 8609 malware samples which were chosen randomly. Dataset 2 contained 1000 malware samples. All of samples in Dataset 2 had pattern groups generated by human analysts. Dataset 3 contained 26254 samples chosen randomly. And a Clean Dataset containing 74530 contained non-malicious samples.

Section 6.3, section 6.4 and section 6.5 contains the experiments on each variant of MPG. Each variant was first executed with Dataset 1 to generate pattern groups. These pattern groups were then matched against Dataset 1 to get matching results. As the last step of experiment the pattern groups were matched against Dataset 3. In the second experiment, each variant of MPG was tested the same way against Dataset 2. Finally, all the results were compared with each other and concluded in section 6.6.

Chapter 7 Future Work

This chapter presents a few research topics which can be pursued to improve MPG.

7.1 Scalability

The rate of new malware development continued to increase over the years. It is safe to say that thousands of new malware samples are found in the wild every week. Current implementation of MPG is not suitable to cope with that level of load.

One of the key limitations of the MPG is that it performs hierarchical clustering which is $O(n^2)$ operation. It is very difficult to scale such an operation for millions of files. There are a few alternatives that can be explored. [BCH$^+$09] presents a scalable method of clustering which uses LSH for malware clustering. This approach can be further explored to see if it can be useful for generating malware behavior patterns.

Another limitation of hierarchical clustering is that no parallel algorithms exist for computing it in its standard form. However, there are other schemes which provide approximate hierarchical clustering and can be parallelized. One such scheme is presented in [GCF12] called hierarchical affinity propagation. This algorithm can be parallelized while providing approximate hierarchical clustering.

Hierarchical clustering is not only processing intensive but also memory intensive. The biggest of the memory requirements comes from the fact that a distance matrix has to be created and complete distance matrix is required in the memory for standard clustering algorithms. Some research can be done to decrease the memory requirements.

MPG uses multiprocessing to take advantage of multiple cores of the system. But it is more expensive to purchase a single very powerful machine compared to a cloud of small machines. Therefore, exploration of cloud based implementations which takes advantage of map-reduce algorithm [DG08] would ensure the ultimate scalability. Hadoop [Apa14] is one such cloud implementation.

7.2 Incremental pattern generation

The current implementation of MPG relies on a dataset of malware. After a dataset is processed, no new files can be added to the analysis. In fact, adding new samples to analysis requires re-execution of the system resulting in re-calculation of previous calculations. Some further research can be done on how to make the system incremental so adding new samples to current analysis would update the patterns without repetition of already done work.

7.3 Intelligence in pattern extraction

In MPG, when patterns are extracted from the pattern tree (section 5.5.4) the patterns are selected based on their coverage. The patterns which detect maximum samples have higher priority for extraction. However, some times a number of events provide equal coverage. In that case, currently no deeper intelligence is introduced in the system which can choose the patterns which are strongest. Furthermore, some research can be done in finding out the optimum number of patterns for each pattern group.

Another problem which has been observed in the system was that sometimes a malicious event is common between many malicious files but it is weak because the string is too short. An example of such event is creation of a mutex named “A”. As the length of “A” is only 1 byte, even if this event is not found in any of available uninfected files, there are chances that it might exist. Some research can be done to overcome such weaknesses. One approach can be to combine it with other malicious events to create multiple pattern groups. This will not only decrease the SPR but also decrease the potential false positives.

Another area which can be explored by research is static scores. A Static score can be assigned to patterns to based on their strength. It can depend on the factors such as length of sub-patterns in a pattern, number of patterns in a patter group etc. This score can then be used in pattern extraction from pattern tree step of MPG (section 5.5.4). The pattern tree can be traversed and and only those pattern groups can be extracted which have static score higher than a predefined threshold.

7.4 Improvement of MPG Fine

The MPG Fine approach takes into consideration properties of events. However, the results showed that some properties of the events which are not very significant but very common rise up in the tree and hence result in creation of pattern groups which are very weak. Some examples of such properties are like port no 80, network traffic on TCP protocol etc. More research can be done in choosing the properties which make a less FP prone, strong pattern.

Moreover, research can be conducted in extracting regular expressions from similar but not identical event properties.

Currently MPG only uses *and* relationship between all patterns in a pattern group. Some resesearch can be done in exploring pattern groups which consist of patterns having a combination of *and* and *or* relationship among them.

7.5 Summary

This chapter presented several research topics which can be explored to improve the performance, scalability and accuracy of MPG. Several challenges related to scalability exist which can be resolved by implementing a memory-efficient cloud based implementation. Secondly, more and more malware samples get discovered every day. To keep the patterns up to date, some work can be done to make the system incremental for processing of new malware samples without recalculating all the work done before. And finally some intelligence can be added to the step of pattern extraction to choose the best pattern out of possible patterns which provide the same coverage.

Chapter 8 Conclusion

This thesis researched the problem of generating automatic behavior patterns for malware. Based on hierarchical clustering a novel software tool named Malicious Pattern Generator (MPG) was developed. The tool is capable of generating behavior patterns from any given malicious dataset. All the generated pattern groups are free of FPs based on a given non-malicious dataset. Three slightly different versions of MPG (MPG Single-Events, MPG Multiple-Events and MPG Fine) were developed. MPG Single-Events produces pattern groups with low detection rate, but number of patterns are comparatively less. MPG Multiple-Events and MPG Fine provide almost 100% detection on the dataset which was used to generate pattern groups and significant detection rate on other datasets. It can be concluded from evaluation results that it is indeed possible to generate reasonable pattern groups from any given malicious dataset with good detection rate.

The thesis started with introduction to the problem and goals in chapter 1 and provided an overview of previous related work in chapter 2. Chapter 3 introduced the malware and different software events which can be used to detect the malware. Chapter 4 presented different components of clustering and specifically explained hierarchical clustering in detail. Based on hierarchical clustering and software events, chapter 5 presented MPG. Three different variants of MPG were presented to produce behavior patterns. Chapter 6 evaluated all versions of MPG using three real world datasets and summarized the evaluation results. Chapter 7 suggested some research topics to further improve MPG.

The evaluation results show that all versions of MPG can be used in two different scenarios. If a user possesses a set of malware samples and wants to find the behavior patterns which can be used to detect those samples, MPG can be used for this. Secondly, a training dataset can be given to MPG to generate some behavior patterns which can later be used for generic malware identification. The evaluation results show that MPG is more successful in the first scenario and produces reasonable results in the second scenario with much room for improvement.

From all the variants of MPG, MPG Fine produces best results on a random dataset, providing best coverage and highest SPR.

References

[ACKW95] William C Arnold, David M Chess, Jeffrey O Kephart, and Steven R White. Automatic immune system for computers and computer networks, August 1995. US Patent 5,440,723.

[Anu14] Anubis, February 2014. URL: http://anubis.iseclab.org.

[Apa14] Apache. Hadoop, 2014. URL: http://hadoop.apache.org/.

[Ask06] Karin Ask. Automatic malware signature generation. Master's thesis, KTH Royal Institute of Technology, 2006.

[Bar96] D. Barr. Common DNS Operational and Configuration Errors. RFC 1912, IETF, February 1996. URL: https://www.ietf.org/rfc/rfc1912.txt.

[BBFP12] B Bencsáth, L Buttyán, M Félegyházi, and G Pék. skywiper (a.k.a. flame a.k.a. flamer): A complex malware for targeted attacks. *www.crysys.hu*, 2012.

[BCH$^+$09] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[BD04] Nahla Barakat and Joachim Diederich. Learning-based rule-extraction from support vector machines. In *The 14th International Conference on Computer Theory and applications ICCTA'2004*, 2004.

[BOA$^+$07] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Recent Advances in Intrusion Detection*, pages 178–197. Springer, 2007.

[CC11] Chire and Wikimedia Commons. File:slink-gaussian-data.svg, 2011. URL: http://upload.wikimedia.org/wikipedia/commons/thumb/b/b7/SLINK-Gaussian-data.svg/1000px-SLINK-Gaussian-data.svg.png.

[Coa14] Blue Coat. Malware analysis appliance, 2014. URL: https://www.bluecoat.com/products/malware-analysis-appliance.

[Coh] Dr. Frederick B Cohen. A short course on computer viruses. In *A Short Course on Computer Viruses*, chapter 1.

[Coh87] Dr. Frederick B Cohen. Computer viruses: theory and experiments. *Computers & security*, 6(1):22–35, 1987.

[Cuc14] Cuckoo, March 2014. URL: http://www.cuckoosandbox.org.

[CV05] Rudi Cilibrasi and Paul MB Vitányi. Clustering by compression. *Information Theory, IEEE Transactions on*, 51(4):1523–1545, 2005.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[GCF12] Inmar Givoni, Clement Chung, and Brendan J Frey. Hierarchical affinity propagation. *arXiv preprint arXiv:1202.3722*, 2012.

[GSHC09] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-Cker Chiueh. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection*, pages 101–120. Springer, 2009.

[HHW+10] Cheng Huang, Nick Holt, Y Angela Wang, Albert Greenberg, Jin Li, and Keith W Ross. A dns reflection method for global traffic management. In *USENIX ATC*, 2010.

[Inc10] Trend Micro Inc. Zeus: A persistent criminal enterprise. *Trend Micro whitepapers*, March 2010. URL: http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp_zeus-persistent-criminal-enterprise.pdf.

[ISI81] University of Southern California Information Sciences Institute. Transmission Control Protocol. RFC 793, IETF, September 1981. URL: http://www.ietf.org/rfc/rfc793.txt.

[Jac01] P. Jaccard. Distribution de la flore alpine dans le bassin des drouces et dans quelques regions voisines. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37(140):241–272, 1901.

[JD88] Anil K Jain and Richard C Dubes. *Algorithms for clustering data.* Prentice-Hall, Inc., 1988.

[JOP01] Eric Jones, Travis Oliphant, and Pearu Peterson. Scipy: Open source scientific tools for python. *http://www. scipy. org/*, 2001.

[KC04] Christian Kreibich and Jon Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *ACM SIGCOMM Computer Communication Review*, 34(1):51–56, 2004.

[Kep95] Jeffrey O Kephart. Methods and apparatus for evaluating and extracting signatures of computer viruses and other undesirable software entities, September 1995. US Patent 5,452,442.

[KK04] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX security symposium*, volume 286. San Diego, CA, 2004.

[KPL$^+$12] Sungkwan Kim, Junyoung Park, Kyungroul Lee, Ilsun You, and Kangbin Yim. A brief survey on rootkit techniques in malicious codes. *JISIS. v2 i3/4*, pages 134–147, 2012.

[KR09] Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons, 2009.

[LCB10] DongJin Lee, Brian E Carpenter, and Nevil Brownlee. Observations of udp to tcp ratio and port numbers. In *Internet Monitoring and Protection (ICIMP), 2010 Fifth International Conference on*, pages 99–104. IEEE, 2010.

[LWLR08] Zhuowei Li, XiaoFeng Wang, Zhenkai Liang, and Michael K Reiter. Agis: Towards automatic generation of infection signatures. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 237–246. IEEE, 2008.

[MC13] McAfee and CSIS. The economic impact of cybercrime and cyber espionage, 2013. URL: http://www.mcafee.com/sg/resources/reports/rp-economic-impact-cybercrime.pdf.

[McA13] McAfee. The state of malware, 2013. URL: http://www.mcafee.com/us/resources/misc/infographic-state-of-malware.pdf.

[Mic14] Microsoft. Msdn library, February 2014. URL: http://msdn.microsoft.com/en-us/library/windows/desktop.

[Mil13] Nikola Milošević. History of malware. *arXiv preprint arXiv:1302.5392*, 2013.

[Mül13] Daniel Müllner. fastcluster: Fast hierarchical, agglomerative clustering routines for r and python. *J. Stat. Softw*, 53:1–18, 2013.

[NKS05] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Security and Privacy, 2005 IEEE Symposium on*, pages 226–241. IEEE, 2005.

[Oll14] Ollydbg, March 2014. URL: http://www.ollydbg.de/.

[PB05] Georgios Portokalidis and Herbert Bos. Sweetbait: Zero-hour worm detection and containment using honeypots. *Elsevier Journal on Computer Networks, Special Issue on Security through Self-Protecting and Self-Healing Systems*, 2005.

[PLF10] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *NSDI*, pages 391–404, 2010.

[RC58] Sokal R and Michener C. A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin*, 38:1409–1438, 1958.

[RHW$^+$08] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[Rus07] Mark Russinovich. Inside windows vista user account control. *Microsoft TechNet Magazine*, 21, 2007.

[SEVS04] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, volume 4, pages 4–4, 2004.

[She14] Jason Sherman. ssdeep, March 2014. URL: http://ssdeep.sourceforge.net/.

[SK03] Stelios Sidiroglou and Angelos D Keromytis. Countering network worms through automatic patch generation. 2003.

[TC05] Yong Tang and Shigang Chen. Defending against internet worms: A signature-based approach. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 2, pages 1384–1394. IEEE, 2005.

[Vir12] ESET VirusRadar. Win32/psw.fakeskype, February 2012. URL: http://www.virusradar.com/en/Win32_PSW.FakeSkype.A/description.

[Wes12] P. Wessa. Hierarchical clustering (v1.0.3) in free statistics software (v1.1.23-r7), office for research development and education, 2012. URL: http://www.wessa.net/rwasp_hierarchicalclustering.wasp/.

[XW05] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *Neural Networks, IEEE Transactions on*, 16(3):645–678, 2005.

[YY10] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *BWCCA*, pages 297–300, 2010.

Appendix A Sample Events

List of Events generated by a malware sample. The “type” field explains the type of the event, while “event” field contains the actual contents of the event. The contents of event can contain many subcomponents such as “new_pid” or “application”.

```
[{
    "type": "obj_createprocess",
    "event": {
        "new_pid": 1578,
        "application": "C:\\Program Files\\Windows Media
           Player\\setup_wm.exe",
        "commandline": "\"C:\\Program Files\\Windows
           Media Player\\setup_wm.exe\" /RunOnce:\"C:\\
           Program Files\\Windows Media Player\\wmplayer.
           exe\" /OCX /NoLibraryAdd /Play \"file://C:\\
           Documents and Settings\\Admin\\Local Settings
           \\Temp\\old_holarh.vi_.mpeg\" /prefetch:10"
    }
}
{
    "type": "obj_createprocess",
    "event": {
        "new_pid": 1579,
        "application": "C:\\Program Files\\Windows Media
           Player\\wmplayer.exe",
        "commandline": "\"C:\\Program Files\\Windows
           Media Player\\wmplayer.exe\" /OCX /
           NoLibraryAdd /Play \"file://C:\\Documents and
           Settings\\Admin\\Local Settings\\Temp\\
           old_holarh.vi_.mpeg\" /prefetch:10"
    }
}
{
    "type": "obj_createprocess",
    "event": {
        "new_pid": 1660,
        "application": "C:\\WINDOWS\\system32\\regsvr32.
            exe",
        "commandline": "regsvr32 /s C:\\WINDOWS\\system32
            \\smtp.ocx"
    }
}
{
    "type": "obj_createprocess",
    "event": {
        "new_pid": 1723,
        "application": "C:\\WINDOWS\\Temp\\old_holarh.vi_
            .exe",
        "commandline": "\"c:\\windows\\temp\\old_holarh.
            vi_.exe\" "
    }
}
{
    "type": "obj_createprocess",
    "event": {
        "new_pid": 1580,
        "application": "C:\\WINDOWS\\explorer.exe",
        "commandline": "explorer.exe C:\\DOCUME~1\\Admin
            \\LOCALS~1\\Temp\\old_holarh.vi_.mpeg"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "CTF.Asm.MutexDefaultS
            -1-5-21-1078081533-842925246-854245398-1003"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "CTF.Layouts.MutexDefaultS
           -1-5-21-1078081533-842925246-854245398-1003"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "SHIMLIB_LOG_MUTEX"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "Local\\ZonesLockedCacheCounterMutex"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "WMSetup-UI"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "AMResourceMutex2"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "CTF.Compart.MutexDefaultS
           -1-5-21-1078081533-842925246-854245398-1003"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "MSCTF.Shared.MUTEX.EI"
    }
}
```

```
{
    "type": "obj_createmutex",
    "event": {
        "name": "CTF.LBES.MutexDefaultS
            -1-5-21-1078081533-842925246-854245398-1003"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "eed3bd3a-a1ad-4e99-987b-d7cb3fcfa7f0 - S
            -1-5-21-1078081533-842925246-854245398-1003"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "Local\\ZoneAttributeCacheCounterMutex"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "
            Microsoft_WMP_70_CheckForOtherInstanceMutex"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "CTF.TimListCache.FMPDefaultS
            -1-5-21-1078081533-842925246-854245398-1003
            MUTEX.DefaultS
            -1-5-21-1078081533-842925246-854245398-1003"
    }
}
{
    "type": "obj_createmutex",
    "event": {
        "name": "CTF.TMD.MutexDefaultS
            -1-5-21-1078081533-842925246-854245398-1003"
        }
    }
    {
        "type": "obj_createmutex",
        "event": {
            "name": "VideoRenderer"
        }
    }
    {
        "type": "obj_createmutex",
        "event": {
            "name": "Local\\ZonesCounterMutex"
        }
    }
    {
        "type": "obj_createmutex",
        "event": {
            "name": "Local\\ZonesCacheCounterMutex"
        }
    }
    {
        "type": "obj_createthread",
        "event": {
            "thread_id": 608,
            "thread_handle": 1456,
            "first8": 5254719917207715723,
            "eip": 2011137162
        }
    }
    {
        "type": "obj_createevent",
        "event": {
            "name": "Global\\userenv:  User Profile setup
               event"
        }
    }
    {
        "type": "obj_createevent",
        "event": {
            "name": "Global\\crypt32LogoffEvent"
        }
```

```
}
{
    "type": "obj_createevent",
    "event": {
        "name": "DINPUTWINMM"
    }
}
{
    "type": "obj_createsemaphore",
    "event": {
        "name": "shell.{210A4BA0-3AEA-1069-A2D9-08002
            B30309D}"
    }
}
{
    "type": "obj_createsemaphore",
    "event": {
        "name": "OleDfRoot000024245"
    }
}
{
    "type": "obj_createsemaphore",
    "event": {
        "name": "C:?WINDOWS?TEMP?OLD_HOLARH.VI_.EXE"
    }
}
{
    "type": "obj_createsemaphore",
    "event": {
        "name": "shell.{7CB834F0-527B-11D2-9D1F-0000
            F805CA57}"
    }
}
{
    "type": "obj_createsemaphore",
    "event": {
        "name": "shell.{A48F1A32-A340-11D1-BC6B-00
            A0C90312E1}"
    }
}
{
    "type": "reg_openkey",
    "event": {
        "key": "HKLM\\software\\microsoft\\windows nt\\
           currentversion\\drivers32"
    }
}
{
    "type": "reg_openkey",
    "event": {
        "key": "HKCR\\exefile\\shell\\open\\command"
    }
}
{
    "type": "reg_openkey",
    "event": {
        "key": "HKCU\\exefile\\shell\\open\\command"
    }
}]
```

Appendix B
Sample Pattern Group

A malware pattern group consists of multiple patterns. These patterns either have an "and" or "or" relationship between them. If the relationship is "or" then if either of the patterns gets matched the whole pattern group will be considered matched. Similarly for "and" all patterns must match for pattern group to match. Each pattern has multiple subpatterns. They also have "and" or "or" relationship between them. An example is shown below.

```
"pattern": {
    "data_created": "2014-03-14 18:06:47.160325",
    "date_modified": "2014-03-14 18:06:47.160331",
    "description": "No description",
    "is_enabled": 1,
    "is_external": 0,
    "is_global": 1,
    "mode": "all_of",
    "name": "Saad Test Pattern 808",
    "owner": "Saad",
    "pattern_group_id": 808,
    "patterns": [
        {
            "event_type": "fs_open",
            "pattern_group_id": 808,
            "pattern_id": 2498,
            "simple_subpatterns": [
                {
                    "event_property": "path",
                    "operator": "equals",
                    "operator_id": 1,
                    "pattern_id": 2498,
                    "subpattern_id": 7351,
                    "value": "c:\\Windows\\System32\\x2.dat"
                },
                {
                    "event_property": "mode",
                    "operator": "equals",
                    "operator_id": 1,
                    "pattern_id": 2498,
                    "subpattern_id": 7352,
                    "value": 18874432
                }
            ],
            "sub_mode": "all_of"
        },
        {
            "event_type": "fs_write",
            "pattern_group_id": 808,
            "pattern_id": 2499,
            "simple_subpatterns": [
                {
                    "event_property": "first8",
                    "operator": "equals",
                    "operator_id": 1,
                    "pattern_id": 2499,
                    "subpattern_id": 7353,
                    "value": 29296
                },
                {
                    "event_property": "path",
                    "operator": "equals",
                    "operator_id": 1,
                    "pattern_id": 2499,
                    "subpattern_id": 7354,
                    "value": "c:\\Windows\\System32\\x2.dat"
                },
                {
                    "event_property": "size",
                    "operator": "equals",
                    "operator_id": 1,
                    "pattern_id": 2499,
                "subpattern_id": 7355,
                "value": 2
            },
            {
                "event_property": "actual_size",
                "operator": "equals",
                "operator_id": 1,
                "pattern_id": 2499,
                "subpattern_id": 7356,
                "value": 2
            },
            {
                "event_property": "offset",
                "operator": "equals",
                "operator_id": 1,
                "pattern_id": 2499,
                "subpattern_id": 7357,
                "value": 0
            }
        ],
        "sub_mode": "all_of"
    },
    {
        "event_type": "fs_create",
        "pattern_group_id": 808,
        "pattern_id": 2500,
        "simple_subpatterns": [
            {
                "event_property": "path",
                "operator": "equals",
                "operator_id": 1,
                "pattern_id": 2500,
                "subpattern_id": 7358,
                "value": "c:\\Windows\\System32\\x2.dat"
            },
            {
                "event_property": "mode",
                "operator": "equals",
                "operator_id": 1,
                "pattern_id": 2500,
                "subpattern_id": 7359,
                    "value": 83886176
                },
                {
                    "event_property": "attributes",
                    "operator": "equals",
                    "operator_id": 1,
                    "pattern_id": 2500,
                    "subpattern_id": 7360,
                    "value": 128
                }
            ],
            "sub_mode": "all_of"
        },
        {
            "event_type": "fs_delete",
            "pattern_group_id": 808,
            "pattern_id": 2501,
            "simple_subpatterns": [
                {
                    "event_property": "path",
                    "operator": "equals",
                    "operator_id": 1,
                    "pattern_id": 2501,
                    "subpattern_id": 7361,
                    "value": "c:\\Windows\\System32\\x2.dat"
                }
            ],
            "sub_mode": "all_of"
        }
    ],
    "revision": 1,
    "risk_score": 10,
    "type": "simple",
    "uuid": "06ae2504-ab9b-11e3-bbb3-80ee7383c060"
}
```

Non-exclusive licence to reproduce thesis and make thesis public

I, Saad Usman Khan

(date of birth: 30-04-1989),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   "Identification of malicious behavior patterns for software" (title of thesis),

   supervised by Dr. Dominique Unruh, Dr. Colin Boyd and Dr. Felix Leder,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 20-05-2014