DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

"Botnet Battlefield": A Structured Study of Behavioral Interference Between Different Malware Families.

Bishwa Hang Rai

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

"Botnet Battlefield": A Structured Study of Behavioral Interference Between Different Malware Families.

"Botnet Battlefield": Eine strukturierte Fallstudie über Verhaltensinterferenz zwischen verschiedenen Malware Familien.

Author:	Bishwa Hang Rai
Supervisor:	Prof. Dr. Alexander Pretschner
Advisor:	M.Sc. Tobias Wüchner
Submission Date:	February 15, 2016

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, February 15, 2016 Bishwa Hang Rai

Acknowledgments

I would like to thank my supervisor Prof. Alexander Pretschner for providing me the opportunity to work on the thesis topic of my choice.
I will forever be thankful to my advisor Mr. Tobias Wüchner for providing me keen guidance from the very beginning of the thesis till the end.

I would like to thank Prof. Christopher Kruegel and Prof. Giovanni Vigna for providing me the opportunity to work at the Security Lab, UCSB and providing me advice and support for conducting the research.
I would like to thank Dr. Ali Zand and Dr. Dhilung Kirat for giving me helpful insights and advice throughout the research.

I would like to thank my friends and family for the motivation they provided to keep working on without loosing composure. Finally, I would like to thank my friends Paul J. Petrich, Suvash Sedhain, and Nick D. Stephens for proofreading the thesis.

Abstract

We present a novel approach for finding the behavioral interference between malware families at large scale. Driven by monetary profit, malware from different families might try to uninstall each other before infecting a system, to get the larger share on the underground market. This is an interesting case of 'environment-sensitive malware' and behavioral interference between malware families. Exploration of this behavior of malware will provide better insights over different families of malware and their associated underground economy. To the best of our knowledge, there is no prior research addressing this problem in a systematic way. We explore this scene in the wild, on millions of malware samples, collected over time by *Anubis*, a dynamic full-system-emulation-based malware analysis environment, from its public web interface and a number of feeds from security organizations. We find the malware pair that have the possibility of behavioral interference and analyze them in the *Anubis* system to detect the interference between the malware. Because of the large dataset and lack of reliable source for labeling malware samples into families, we use machine learning techniques (document clustering) to cluster the malware samples, based on malware behavioral profiles, into different malware families. We use heuristics to find the probable malware pair with behavioral interference, based on common resources (files, registries, and others), interactions (modify or delete), and belonging to different clustered families. Our approach could find X number of malware pairs with the true positive rate of X%. This novel research will help better understand the dynamic aspect of malware behavior.

Contents

1 Introduction

Malware, short for malicious software, is a generic term referring to any illegitimate software that can cause damage or steal the data, disrupt the operation, or gain unauthorized access, on a computer or computer network [Cis]. Malware can infect the system by being bundled with some other program or being attached as macros in a file. When a user opens such a program or file, the malware gets installed onto the user's machine. Malware can also install themselves by exploiting some common vulnerabilities in an operating system (OS), a network device, or other common software. A vast majority of malware is installed by some action by the user, such as clicking an e-mail attachment or downloading a file from the Internet [Cis].

Malware can be classified into self-replicating, such as *viruses and worms*, and non-replicating, such as *Trojans* [Che]. Unlike a virus, which needs user action (opening the infected file), worms are capable of automated replication and propagation, through computer networks [Che]. Trojans disguise themselves to be legitimate and harmless, but have hidden malicious functions. A malware can also be categorized into *polymorphic* and *metamorphic*, by its ability to change its code-structure every time it infects a new victim. *Polymorphic* malware uses code obfuscation (e.g. dead-code insertion, subroutine reordering, instruction substitution) and encryption to create a new variant of itself, whereas, *metamorphic* malware uses only code obfuscation to change its code-structure [RMI11; RMI12]. However, the semantics of the program (malware) remains same. We discuss in details about types of malware in section 2.1.

As the Internet increased growth over the last two decades, more and more devices are now connected to the Internet. Many of our daily life activities are now depended on its usage, such as email, banking, bill payment, and social networking. Along with this, the underground Internet economy, illegal trading of valuable data and personal information, has also been on the rise. Malware authors now do not write malware just for fun, to create annoyance or break into some system for bragging rights, but also for profit. Malware authors look for banking credentials, credit cards, and personal information that they can gather and sell in the underground market. In 2006, annual losses caused by malware were estimated to be 2.8 billion dollars in United States and 9.3 billion euros in Europe [MCA09]. Driven by the high profit and rise in easily

available tools to create polymorphic and metamorphic malware, there has been rapid increase in the number of new malware being introduced each day [Tia+10]. According to *AV-Test*, an independent IT-Security Institute based in Germany, about 140 million new malware were introduced in 2015 alone, making the total number of malware recorded so far almost half a billion [AVT].

Malware has become a profitable business. Malware authors are now vying with each other to take sole control of their victim for larger profit. Many cases have been found, where one malware tries to delete another malware from the victim's machine. Further, once they have infected the victim, they prevent the host from being infected by other malware. In 2010, *SpyEye*, a Trojan, was found to have a feature *KillZeus* that would remove *Zeus*, which is also another Trojan, making *SpyEye* the only malware to run in the compromised system [Har]. In 2015, *Shifu*, a banking Trojan, was found to have a similar behavior as Anti Virus (AV), i.e., preventing the host from further infection by other malware, once it had taken control over the victim's machine [Lim]. We have discussed in details, about such anecdotal evidence of interaction between malware families, in section 2.2.

1.1 Problem Statement

The purpose of our research is to identify the existence of aforementioned behavioral interference between the malware families. The study of behavioral interference between malware families will provide novel knowledge for understanding the dynamic aspect of modern malware, the inter-family relations, and their associated underground economy. This behavior is also a case for environment-sensitive malware, where malware change their behavior depending on different factors of their running environment, such as presence or absence of files, programs, or running services. To the best of our knowledge, there hasn't been a study of such interference between two malware families so far. We present a systematic way to find out such interferences from a large number of malware samples. We analyze our malware sample dataset to find the malware pairs that could possibly exhibit the interfering behavior with each other. The malware pair is then run in malware analysis system together and examined to detect any behavioral interference.

1.2 Research Process

A holistic overview of our research approach is shown in Figure 3.1. To perform the research, we worked with dataset of 22,154,180 malware samples, collected overtime

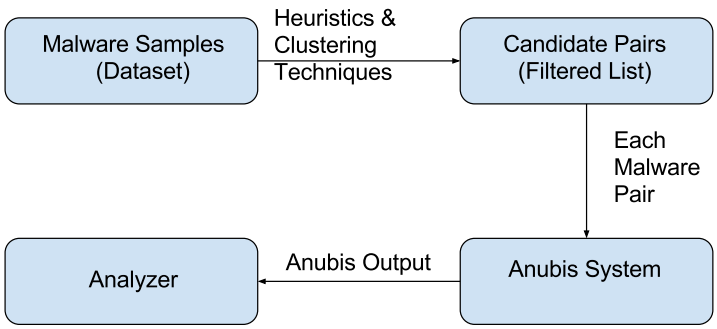

Figure 1.1: System Overview

(2007–02–07 to 2012–09–10) by *Anubis* [Anu], a dynamic full-system-emulation-based malware analysis environment, from its public web interface and a number of feeds from security organizations. Detailed discussion on *Anubis* and malware analysis is done in section 2.3.

Randomly picking a malware pair to analyze from the available dataset would not be a good approach, whereas analyzing all the possible pairs of millions of malware is not scalable. We use heuristics and clustering techniques to filter our dataset and get list of malware pairs that has high probability of exhibiting the behavioral interference. We call those malware pairs **candidate pairs**. The candidate pairs, which is the subset of whole dataset (thousands in numbers), are then run in malware analysis system, and the results are further analyzed to detect if there is any behavioral interference between the malware samples.

We generate a list of probable candidate pairs, from our malware dataset, based on the common resource that the malware sample operates (create, access or delete) on. **Resource** refers to any entity, such as Files, Registry, Section, Sync, that can be modified or queried by malware using system calls. Since *Anubis* emulates the Windows OS, **system calls** refers to the Windows Application Programming Interface (API) functions. The trace of all the resources modified or queried by malware during its execution is referred to as the **behavioral profile** of that malware. In subsection 3.1.1 and subsection 3.1.2 we define behavioral profiles and different resource types in detail respectively.

In order to minimize our search space for candidate pair selection, and maximize the probability for finding a pair with behavioral interference, we take into account only those malware which would create a resource successfully and another malware that

tries to access or delete that same resource, but with a failure. If malware makes a failed attempt to access or delete resources created by another malware, there should be some interaction between those malware, as these actions are suspicious. It is trying to detect the presence of other files or registries in the environment it is currently running, and its behavior is affected by the positive or negative detection. This shows the dynamic aspect of malware behavior and its environment-sensitive nature.

The simple heuristic of finding interfering malware candidates based on common resource gave us concrete candidate pairs. The candidate pairs were still too large to run in a scalable manner and find substantial results. In order to further study the dataset, we used clustering of malware (based on behavior of malware). **Malware Clustering** is the process of grouping a set of malware in such a way that malware in the same group are more similar, based on the behavioral profile in our case, to each other than to those in other group. Such groups are called **clusters**. Clustering of malware based on behavioral profiles has been used extensively in previous research, which we have listed in details in section 2.4. We treat each cluster as a family, and malware falling in those clusters as variants of that family. Candidate pairs are chosen in such a way that two malware representing a pair, do not belong to the same family. The algorithm for candidate selection based on clustering results and the common resource heuristic is described in subsection 3.2.5. Each candidate pair from the list of candidate pairs were then run in the *Anubis* system for analysis.
Two malware sample belonging to a single candidate pair were run in the *Anubis* system consecutively, with a time interval between the two run. The procedure has been described in detail in subsection 3.2.6. The result of Anubis was then analyzed to see if one malware deletes the resource created by another, which provides proof for behavioral interference between the two malware. The details of result analysis is described in subsection 3.2.7.

1.3 Contribution and Structure

Our research will provide the following contributions:

- To the best of our knowledge, we are the first to perform a systematic study of interferences between malware families. Our work is based on wide variety of a large malware dataset collected over time in the wild. This novel research will help better understand the dynamic aspect of malware behavior.
- A novel approach to malware clustering based on malware behavior profiles.

- An automated system that detects interfering malware samples on a large scale.

We discuss malware types and families, evidence of interaction between malware families, malware analysis system, previous works on the use of malware clustering based on malware behavior generated from dynamic malware analysis, and motivation for our research in chapter 2. In chapter 3, following the motivation, we give an overview, design rationale, and give a technical description of our work flow and system. In chapter 4, we explain how we implemented the system, problems faced during the implementation, and how we solved them. We present the findings and evaluation of our work in chapter 5. Finally, we conclude with a discussion of our work and possible improvement in future in chapter 6.

2 Literature Review

In this chapter we will discuss anecdotal evidence of malware interference, malware analysis systems, and previous research on malware clustering. With the insight from the literature review, we reason the motivation and design rationale of our research work; to study the behavioral interference between malware families at a large scale. We first describe commonly known malware types and families.

2.1 Malware Types and Families

Some of the commonly known malware types are viruses, worms, Trojans, and bots.
A **Virus** is a piece of malicious code attached to some other executable host program or file, which gets executed when the user runs the host program. It can replicate itself to form multiple copies but cannot propagate on its own. It spreads when it is transferred or copied, by the user, to another computer via the network by file sharing or as an email attachment.
A **Worm**, unlike a virus, is capable of running independently and is self-propagating. It propagates by exploiting some vulnerability in a machine's OS, a device driver or taking the advantage of file-transfer features, such as email or network share.
A **Trojan** is a non-replicating malware which gets its name from the ancient Greek story about the *Trojan Horse*. It is a software that looks legitimate, but is harmful. Trojans are non reproducing and non self-replicating. It spreads only by user actions such as opening infected Internet downloads and email attachments [Cis]
A **Bot** is a malicious software that allows the infected host machine to be accessed and controlled remotely by its author, also called *"botmaster"*. The botmaster does this from a central server called the Command and Control (C&C) server. A network of such many Internet-connected bots is called **Botnet**. A bot is capable of logging keystrokes, gathering passwords and financial information, and sending it to the C&C server. A Botnet can also be used to launch *Distributed Denial of Service (DDoS)* attacks, by flooding a single server with many requests, or sending spam at a large scale.

Some malware is able to change their structure and create a new variant of itself each time it infects a new victim. On the basis of how a malware changes its structure, it can be broadly divided into Polymorphic or Metamorphic malware. **Polymorphic** malware

divides its program into two sections of code: a "decryptor" and "encryptor". The first code section, the "decryptor", decrypts the second section of the code and hands the execution control to the decrypted section of code. When the second section of the code (encryptor) executes, it creates a new "decryptor" to encrypt itself, and links the new encrypted code section (encryptor) and the new "decryptor" to construct a new variant of the malware. Code obfuscation techniques (e.g. dead-code insertion, subroutine reordering, instruction substitution) are used to mutate the "decryptor" to build the new one for a new victim [RMI11]. **Metamorphic** malware are body-polymorphic malware [SF01]. Instead of generating new "decryptor", a new instance is created using similar code obfuscation techniques as used in polymorphic malware. Unlike Polymorphic malware, it has no encrypted part [RMI12]. In both cases the new variants may have different syntactic properties, but the real behavior of the malware remains same.

Malware are categorized to different families in accordance to the malware author and similar behavior. In this section, we briefly describe three most common malware families to provide and idea of their behaviors, capabilities, and mode of operation. Even between these three families we can see stark differences in their behavior and approach of infecting a victim's machine permanently, keeping low presence. Each malware family creates or modifies some specific resource (such as file or registry), which can be used to detect their presence.

2.1.1 Conficker

Conficker is a computer *worm* that targets the *Microsfot Windows* Operating system which was first found in November 2008. According to *Microsoft* the detection of *Conficker* worm increased by more that 225 percent since the start of 2009 [Micn]. It is capable of infecting and spreading across a network without any human interaction. If the user of the compromised machine does not have admin privilege, it is capable of trying different common weak passwords (such as *'test123'*,*'password123'*) in order to gain admin privilege of the network share directory, and drop a copy of itself there.
The worm first tries to copy itself into the Windows system folder as a hidden Dynamic Link Library (DLL) using some random name. When unsuccessful it tries to copy itself in *%ProgramFiles%* directory. In order to run on startup, every time Windows boots, it also changes the registry as in Listing 2.1.

Listing 2.1: Registry key created by Confiker worm for autostart

```
In subkey: HKCU\Software\Microsoft\Windows\CurrentVersion\Run
Sets value: "<random string>"
```

```
With data: "rundll32.exe <system folder>\<malware file name>.dll,<malware
    parameters>"
```

2.1.2 Zeus

Zeus is another malware of *Trojan* type, that affects the *Microsoft Windows* OS. It attempts to steal confidential information once it infects the victim's machine. The information may include systems information, banking details, or online credentials of the compromised machine. It is also capable of downloading configuration files and updates from the Internet and change its behavior based on new update. The Trojan is generated by a toolkit which is also available in the underground criminal market, and distributes itself by spam,phising and drive-by downloads [Sym, Trojan.Zbot].
Zeus tries to create a copy of itself as any of the names as *ntos.exe, sdra64.exe,twex.exe* in the *<system folder>*. In order to make itself run every time the system starts, after reboot or shutdown, it changes the registry as in Listing 2.2 [Micm, Win32/Zbot].

Listing 2.2: Registry key modified by Zeus Trojan to autostart

```
In subkey: HKLM\Software\Microsoft\Windows NT\Currentversion\Winlogon
Sets value: "userinit"
With data: "<system folder>\userinit.exe,<system folder>\<malwar"
```

2.1.3 Sality

Sality is a family of polymorphic malware that infects files on the *Microsoft Windows* OS with extensions *.EXE* or *.SCR*. It spreads by infecting executable files and replicating itself across network shares and steals sensitive information like cached password and logged keystrokes.
Each infected host becomes the part of peer to peer (p2p) botnet that would create a hard to take down decentralized network of C&C servers. The botnet is also used to relay spam, proxy communications, or achieve DDoS [Nic, Sality].
Sality targets all files in the system drive (usually *'C'* drive in Windows) and tries to delete files related to anti-virus. It stops any security (anti-virus) related process and changes Windows registry keys related to AV software in order to lower the computer security. One of the symptoms to diagnose if a machine is infected by the *Sality* family of malware is the presence of files, listed in Listing 2.3, in the system [Micl, Win32/Sality].

Listing 2.3: Files created by Sality in the infected machine

```
<system folder>\wmdrtc32.dll
```

```
<system folder>\wmdrtc32.dl_
```

2.2 Interference between Malware Families

We discussed malware families and their supposed behavior in the previous section. In this section we show some of the evidence of interference between different malware families. We look upon those incidents to know the families involved, nature of interference, and the reasons behind it. One of our main goals is finding such behavioral interference between the malware families at a large scale. This anecdotal evidence, quite popular among the security related media, will serve as a ground truth for our research work.

2.2.1 Bagle, Netsky, and Mydoom feud

This feud between the malware family dates back to 2004, where there was an exchange of words between the creators of *Bagle, Netsky* and *Mydoom*. All three malware families were computer worms spreading through email as an attachment and making victim curious enough to download and open it [Wikd]. The creators inserted their message inside the malware itself, making it visible for the victims too. This also gained much media attention and even appeared in [Bri, BBC]. Message *"don't ruine (sic) our business, wanna start a war?"* was seen inside the *Bagle.J*, where as *NetSky.F* responded with message *"Bagle you are a looser!!!! (sic)"*. Similar messages with profanity were also seen in variants of *Mydoom.G* families.

The reason for the war between these malware families was *Netsky* trying to remove the other two malware, *Bagle* and *MyDoom*, from the victim's machine. The creator of *Netsky*, *Sven Jaschan*, admitted that he had written the malware in order to remove the infection with *Mydoom* and *Bagle* worms from the victim's computer [Wikd].

2.2.2 Kill Zeus

This was another war between the two *Trojan* malware families, *Spy Eye* and *Zeus*.
SpyEye is a Trojan malware which, like *Zeus*, was created specifically to facilitate online theft from financial institutions, especially targeting the U.S. It infected roughly 1.4 million computers, mainly located in the U.S, and got the personal identification information and financial information of victims in order to transfer money out of victim's bank accounts [Fed]

SpyEye came with the feature called *Kill Zeus* that was successful in removing large varieties of the *Zeus* family. The battle between these two families were quite dynamic. Many releases of the Trojan toolkit were made from both families in order to negate each others dominance [Har].

2.2.3 Shifu

Shifu, a highly evasive malware family, targeting mainly Japanese (82%), Austrian-German (12%), and rest of European (6%) banks for sensitive data, was detected recently in the year 2015 [Lim]. It is evasive because it terminates itself if it finds out that it is being run inside the virtual machine or is being debugged. It knows if it is being run inside a virtual machine by checking for the presence of files such as *pos.exe, vmmouse.sys, sanboxstarter.exe* in the system [Diw]. In order to check if it is being debugged, it calls *IsDebuggerPresent* Windows API which detects if program is being debugged [Diw].
Shifu also prevented any other malware from infecting its victim. It keeps track of all the files being downloaded from the Internet. For any files that were downloaded from unsecured connections (not HTTPS) or are not signed, it considered those files suspicious, and renamed it to *infected.exe*. It stops those suspicious files from being installed on the system and also sends a copy of the file to its C&C server, probably for further analysis [Lim]. If there was already presence of other malware, it stops any updates to those other malware, by disconnecting them from their botmaster [Lim].

2.3 Malware Analysis

In the previous sections, we discussed malware families and interference between them. We need a malware analysis system to detect such behavioral interference from a large dataset. In this section we describe techniques of malware analysis and *Anbuis*, the dynamic analysis system, we chose for our research.

Malware analysis is the process of dissecting different components of a malware sample in order to study the behavior of the malware when it infects a host or victim's machine. It is done using different analysis tools and reverse engineering the binary. There are two main types of malware analysis techniques, *Static* and *Dynamic*.

Static anaylsis is done without running the binary, but studying the assembly code of binary to detect the benign or malicious nature of program. One way to perform static analysis is by disassembling the binary with a disassembler, decompiler, and source code analyzer to find the control flow graph (CFG) of all the code segment and

path that the program might take under different given conditions. If the analysis result crosses the preset threshold of malicious activity then it is marked as infected [SS14]. *Static analysis* gives all the possible behavior of program but for large number of huge programs it is hard and time consuming.

Dynamic Analysis is done by running the binary in a closed controlled system such as *Sandbox* and logging all the activities (activity related to Registry, File, Process, Network etc.) of the malware while it runs in host. Detection of malicious activities, such as attempt to open other executables with intent to modify its content, changing Master Boot Record or concealing themselves from the operating system, would label the program as malware [SS14]. The main drawback of dynamic analysis is that only a single execution path is examined. Chipounov et al. proposed an approach to multi-path execution by triggering and exploring different execution path during run-time [CKC12]. Also, many evasive malware can evade detection by *dynamic analysis* by identifying the presence of analysis environment, and refraining from performing malicious activities [KVK14]. For an instance, *Shifu* malware that we discusses above.

We chose dynamic analysis because we can run the candidate malware pair together, to find the behavioral interferences, in live action. Static analysis would be tedious if we were to analyze each malware sample of a candidate pair and correlate them for interference, within the time constraint.

2.3.1 Anubis

For our experiment we use **Anbuis**, a dynamic malware analysis platform for analyzing Windows PE-executatbles [Anu]. Anubis is a whole-system emulator for PCs and Intel X86 architecture. It executes binary inside the emulated environment and gets the details of all the system calls invoked by binary [Bay+09]. With the trace of system calls, *Anubis* generates a detailed report about modifications made to the Windows registry or the file system, about interactions with the Windows Service Manager or other processes and logs of all generated network traffic [Anu]. We chose *Anbuis* because of the following reasons:

- We had direct access to Anubis and multiple instance of Anubis system to perform our malware analysis.
- Our dataset of malware were those collected by Anubis over time, and we had access to already generated reports of those malware.
- It is one of the well accepted dynamic analysis platforms among security researchers.

2.4 Malware Clustering

We wanted to select pairs that have high probability of behavioral interference and analyze them in *Anubis*, a dynamic analysis system, to confirm. We cluster our large dataset (millions of malware samples) into different clusters (families) based on the malware behavior. This helps us find the candidate pair from smaller candidate set (clusters), malware in a pair belonging to two different clusters, without compromising the quality of the dataset. Anti-virus companies do provide labels to categorize malware into different families (such as listed in *VirusTotal* [Vir] report), but they are not reliable. They do not fulfill the criteria of consistency, completeness, and conciseness [Bai+07].

- **Consistency.** Different AV vendors place the malware into different categories, and these categories also do not hold the same meaning across the vendors. This leads to inconsistency in the AV vendors labels.
- **Completeness.** Not every malware has been labeled by the AV vendors, and many malware go undetected because of this. A month old binary could still be unlabeled, which makes the AV system labeling incomplete.
- **Conciseness.** The label provided by an AV vendor are either too little or too much information with not much substantial meaning. They usually give a general idea of broad term as "trojan" and "worm" and not a concise description of the specific malware or its family.

As malware continue to evolve and rise in numbers [KVK14; AVT], researchers are investigating machine learning techniques such as classification and clustering to analyze malware. **Classification** is an instance of supervised learning where a trained set of correctly identified data is used as a reference to classify new data. Some examples of classification algorithms are *Naive Bayes, Random Fores, and Neural Networks* [Wikc]. **Clustering**, on other hand, is an instance of unsupervised learning, i.e, grouping unlabeled data object which are similar in some sense, in the same cluster. Some examples of clustering algorithms are *expectation-maximization and k-means* [Wika]. A good clustering system would be able to study behavior of a new binary to find, if the new binary is a variant of previously known malware and classify them accordingly.

Because AV label on malware families are not reliable, we look onto different machine learning techniques to categorize our dataset into different families. **Pirscoveanu et al.** used *Random Forest* classifier on the behavior data set of *42,068* malware to achieve a high classification rate with a weighted average Area Under Curve (AUC) value

of 0.98 [Pir+15]. **Moskovitch et al.** used *Naive Bayes* classifier based on 20 malware-behavior to detect malware with mean detection accuracy of over 90% [MER08]. **Yavvari et al.** used behavioral mapping approach to cluster *1,727* unique malware pair samples [Yav+12]. **Firdausi et al.** in their paper did the measurement of five different classifiers, *k-Nearest Neighbor, Naive Bayes, J48 decision trees, Support Vector Machine, and Multilayer Perception Neural Network*, to classify the malware based on behavioral data, and state that machine learning techniques based on behavioral profile can detect malware quite efficiently and effectively [Fir+10]. Previous research work, showed that clustering and classification yields superior performance in timely detection of malware, against the arms race with malware authors, compared to signature based, dynamic or static analysis [Bai+07; Bay+09; Rie+09].

Bailey et al. introduced the first clustering system based on observed behavior of malware [Bai+07], which was later extended by Bayer et al. [Bay+09], to be scalable [Rie+09]. We studied the work of Bayer et al. in detail, as their work clustered malware in large scale; 75 thousands malware under three hours [Bay+09]. The clustering was done for tens of thousands of malware samples, but the approach still was not scalable to millions of samples we needed to cluster. The approach has a linear bootstrapping phase of Locality-sensitive hashing (LSH), after which the $O(n^2)$ hierarchical clustering starts. The whole premise of faster execution lies within careful tuning of multiple parameters and hash functions (to make the initial phase take care of most of the load), which had not been done by the authors for millions of samples (the biggest execution they had was for 75,000 samples). This means that we would have needed to tune and change the code as we go forward to get the results we hope for.
Another approach was to use a clustering algorithm that is scalable and capable to cluster millions of samples. We decided to map the problem to document clustering, considering each malware as a document, and their behavioral profile as the words in document. As **term frequency-inverse document frequency** (*tf-idf* a statistically measured weight to evaluate the importance of a word in document with respect to the whole corpus [Wike]) approaches have a large memory footprint, $O(\#documents \times \#words)$, we switched to clustering algorithms whose memory footprint does not depend on the number of documents, and decided to use *latent Dirichlet allocation* (**LDA**). *Latent Dirichlet Allocation* (**LDA**) is a generative probabilistic model for collections of discrete data such as text corpora [BNJ03, LDA]. **LDA**, equivalent to dimension-reduction algorithms for high-dimensional clustering, is one such algorithm which does not depend on the number of documents and its memory footprint is $O(\#words \times \#clusters)$. This gives us a more fine grained clustering compared to previously proposed LHS (Locality-sensitive hashing) based approach.

2.5 Summary and Motivation

Motivated by economic profit, there has been an increase in the number of new malware, and fighting between the malware families to control the larger share of the underground economy. To get the larger piece of the economy and show their superiority, malware family try to negate the existence and influence of another family. We showed some evidence of interferences between the malware families where they removed other malware, added a feature to remove other malware, or blocked other malware from infecting its victim. Such behavioral interference between the malware families is an interesting case to study and would provide a novel aspect of dynamic behavior to evolving malware. Our research provides a systematic way to find such behavioral interference of malware families on a large scale.

We gave an overview on two common malware analysis approaches and reason for choosing *Anubis*, a dynamic malware analysis platform, as we could run our candidate pairs in real time to check for behavioral interference. We see that different malware families have different techniques to inject themselves and keep themselves alive even after the system reboot. As described in section 2.1, they copy themselves in certain system paths, disguise themselves with benign filenames and register themselves to autostart services. The names of different resources created by malware can be random or peculiar. These patterns related to each family can be used to check the presence of particular family. We use this rationale in our heuristics to find probable malware candidates with behavioral interference. To select the sample candidate pair, we look for the malware that tries to access or delete resources, which was created by another malware, from another family. The detailed approach of selections of pair is described in subsection 3.2.5. We believe such behavior of malware is suspicious and interesting case of interference between two malware family.

We discussed why AV vendors labeling of malware family are unreliable and how we will be using machine learning to achieve the clustering of malware to families. We discussed the use of machine learning techniques in previous research for clustering and classification of malware based on their behavioral profiles. The results of the work so far on classification and clustering of malware were good in order to improve efficiency and detection rate of the malware analysis system. However, the clustering of malware performed so far are on a small number of samples, consisting of tens of thousands of malware samples, and are not scalable. We present a different approach for large scale malware clustering, using LDA for document clustering and python library *Gemsim* for the implementation.

Our working dataset consists of millions of malware samples. We select a subset of malware sample from the whole dataset that has the probability of behavioral interference and examine them to detect if any. The study of behavioral interference between the malware families in this large scale is new research in the field of malware study. It will help to understand dynamic behavior of environment sensitive malware.

3 Methodology

In this chapter we will give overview of our system design and work flow. First, in section 3.1, we will give technical descriptions of terms we will be using to describe our work flow. In section 3.2, we will describe the overall procedure of our research work.

3.1 Terminology

In this section we give a detailed description of different terms we will be using through out the report.

3.1.1 Behavioral Profile

Bayer et al. define **behavioral profile** as, "the abstraction of a program's execution trace that provides information on the OS objects that the program operated on, along with the operations". OS Object refers to resource type such as file, registry or section, that could be modified or queried with the system calls. System calls consisted of Windows NT, native API and the Windows API functions. A program uses Windows API and native API functions to communicate with the Windows system such as creating a file or deleting a registry.

The behavioral profile was based on the execution traces of programs irrespective of order execution as reordering system calls could be done without changing the semantics of program [Bay+09]. It consisted of a list of different operations operated on the different OS objects during the execution of binary. The system calls that had same purpose as resultant output but different calling API name were generalized under single name—*NtCreateProcess* and *NtCreateProcessEx* would be generalized as, *'create'* OS Operation for, *'porcess'* OS object [Bay+09].

OS Objects

OS Object were primarily the resource that were created, delete, modified during the program execution. Bayer et al. define *OS Objects* as:

```
OS Object ::= (type, object-name)
type ::= file|registry|process|job|
       network|thread|section|
       driver|sync|service|random|
       time|info
```

An OS Object consists of its type and object-name. The type of OS Object are file, registry, process, job, network, thread, section, driver, sync, service, random, time, and info. An object-name is the name of the OS object. For an instance, `'c:\ntlos.exe'` is an object-name for object type *file*.

OS Operations

Broadly, OS Operation is the generalization of a system call. Bayer et al. define *OS Operations* as:

```
OS operation ::= (operation-name,
              opeartion-attributes?,
              successful?)
```

An OS operation object consists of name of the operation, any additional attributes, and the status of the operation whether it was successful or not.
A snippet of behavioral profile is shown in Listing 3.1 with sample of operations executed on OS Objects file, registry, and section.

Listing 3.1: Behvaioral Profile sample

```
op|file|'C:\\Program Files\\Common Files\\sumbh.exe'
 create:1
 open:1
 query:1
 write:1

op|registry|'HKLM\\SOFTWARE\\CLASSES\\CLSID\\{00021401-000000000046}'
 open:1
 query:1
 query_value(''):1
 query_value('InprocServer32'):0
```

```
op|section|'BaseNamedObjects\\MSCTF.MarshalInterface.FileMap.ELE.B.FLKMG'
 create:1
 map:1
 mem_read:1
 mem_write:1
```

To explain the behavioral profile shown in Listing 3.1, *'create'* operation for file OS object `'C:\ProgramFiles\CommonFiles\sumbh.exe'` was successful with success value 1, whereas *'query_value'* operation with operation-attribute value *'InprocServer32'* for registry OS Object `'HKLM\SOFTWARE\CLASSES\CLSID\{00021401-000000000046}'` failed with success value 0.

We had the "behavioral profiles" of the malware samples that we used to create new database which is described in section 4.4. Informations from *OS Objects* and *Operations* were used to extract resource activities (execution traces) of the malware (program).

3.1.2 Resource Type and Activities

As discussed before in section 3.1.1, an OS Object, are representation of resource types. For our research, we considered following 8 resource types into consideration, that could best determine the behavioral interference between malware. These resource types are the most common resource that malware modifies in order to infect the system and disable security services. We will give a short description of each resource type according to Microsoft Developers Network documentation [Mice, MSDN].

File

A *file* is a means of storing resourceful information which can be retrieved or modified in future. File objects function as the logical interface between kernel and user-mode processes and the file data that resides on the physical disk. It not only holds the data written on the file but also a set of attributes maintained by the kernel for system purposes such as *File name, Current byte offset, Share mode, I/O mode* [Micb].
File type in the behavioral profile encompasses not only general file, but named pipe and mailslot resources. File is an important resource type as many malware creates or deletes file in order to infect the system or remove another malware from the system. These file activities will be important behavior to find the interference between malware families.

Registry

A *Registry* is a database defined by a system where different applications and system components store and retrieve data such as configurations settings for its use. The data stored in the registry varies according to the version of Microsoft Windows. Application performs the basic add, modify, retrieve, or delete operation in the registry through the registry API [Micg].
Registry keys provide vital information on malware type and behavior as we have seen in section 2.1 that different malware families create different registry keys when they infect a system. Malware from same family modify specific registry key, which is used to detect their presence.

Service

A *Service* is a computer program that operates in system background, similar to UNIX daemon [Micj]. A *service* can be started at system boot through the Service Control Panel, or an application can also use service functions such as *StartService, OpenService, DeleteService* to configure services. However, it must conform to the interface rules of Service Control Manager (SCM) [Mici].
Malware can stop other services related to security or other malware, create a new service, or hook itself into autostart services. These service related activity are thus interesting to study malware behavior and interaction with other malware.

Section

A *section* object is sharable memory which is used by process to share its memory address space (memory sections) with other processes. It is also used by process to map a file into its memory address space [Mich].
In case of behavioral profile, it broadly represents memory mapped files—file with content in virtual memory, enabling application and multiple processes to modify the file by reading and writing directly to the memory [Micd].

Process

A binary can spawn one or more *processes*. A *process* is simply an instance of a computer program being executed that consists of instructions and current activity of program [Micf].
Malware trying to detect and kill the process created by another malware is interesting case of behavioral interference.

Job

A *Job* object makes grouping of process as single unit to manage possible. It can be named and shared securely to control attributes of processes grouped together and operation on a job makes the affect on all the process in its group [Micc].

Sync

A *sync object* is used to coordinate the execution of multiple threads as more than one process could share the handle of single synchronization object which helps for the interprocess synchronization between these processes [Mick].
The sync object type covers all the synchronization activities. [Mick].

Driver

A *device driver* is a program that is associated with certain device for its operation and control. It is used as an software interface to communicate between the hardware device and the operating system and other software [Wikb, Device Driver]
Windows represent devices with device objects, and one device could be represented by more than one device objects. All operation on device is conducted via device object [Mica].
We capture those loading and unloading of Windows Device Driver recorded in the behavioral profile.

Resource activities in our work refers to operations, such as *create, delete, modify, or access*, performed by a malware (program), on the resource types *File, Registry, Service, Section, Process, Job, Sync, or Driver*, during its execution. **Resource name** is same as object-name in *OS-Objects*[section 3.1.1]—name of the resource created, modified, deleted, or accessed. We use resource activities of malware samples to select the malware candidates that could exhibit behavioral interference [section 4.3]. Further, resource activities related to malware were represented as *"words"* related to *"document"* (malware) to create a text corpora for document clustering [section 4.5]. We describe the terms *"words"*, *"document"*, and *"corpora"* in subsection 3.1.3.

3.1.3 Words, Document, Corpora

In this section, we describe the terms *"words"*, *"document"*, and *"corpora"* and how we associate, malware samples and resource activities, with those terms in our work. The terms will be used extensively when describing clustering [section 3.2.4] and its implementation [section 4.5].

Words

In their work, Blei et al., describe word as, 'A *word* is a basic unit of discrete data, defined to be item from a vocabulary indexed by $\{1, \ldots, V\}$" [BNJ03].
We represent a single resource activity of a malware as single word. For an instance, a file creation activity will be a single distinct word and a file deletion activity will be another distinct word. Same resource activity—with same resource type, resource name and operations—will be considered as repetition of same word. For example, multiple registry keys created with same name by same or different malware.

Document

According to Blei et al., "a *document* is a sequence of N words denoted by $\mathbf{w} = (w_1, w_2, \ldots, w_N)$, where w_N is the nth word in the sequence" [BNJ03].
A single malware sample will be represented as a single document and all the resource activities related to that malware will be the words in that document.

Corpora

"A *corpus* is a collection of M documents denoted by $D = \{w_1, w_2, \ldots, w_M\}$" [BNJ03].
The total resource activities (words) related to all the malware samples (documents), in our dataset, will be our corpus.

3.2 Work Flow

In this section we outline an overview of our work flow. Figure 3.1 depicts the overall structure of of our system. We describe about the old database that we had access to and the need for the creation of new database in subsection 3.2.1. We create reverse index from the database and use heuristics to select candidate pair which is described in subsection 3.2.2 and subsection 3.2.3 respectively. To lower the number of candidate pair and find malware pair belonging to different families, we cluster the malware based on their behavioral profile as described in subsection 3.2.4. The candidate pair selection based on clustering is described in subsection 3.2.5. In subsection 3.2.6 and subsection 3.2.7, we describe running the candidate pair and analyzing the result respectively.

3.2.1 Database

We required resource activities of the malware samples to process and find malware with behavioral interference. In oder to monitor the resource activity of binary, Anubis uses *virtual machine introspection (VMI)* technique. According to Garfinkel et al., "VMI is the approach of inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it" [G+03, p. VMI]. For every binary submitted to Anubis for analysis, it executes the binary inside the emulated environment and keeps track of operating system services that are called by the binary. Anubis does this by keeping the log of all the Windows native system calls and Windows API functions that the binary invokes. To know which operating system function is called, it compares the current value of virtual processor's instruction pointer register to the start address of Windows NT and Windows API functions [Bay]. Resource activity such as creating a file or a registry, by the binary with the Windows API functions (*CreateFile, RegCreateKeyEx*), are monitored and recorded.

A portion of sample log file is shown in Listing 3.2. In Line 6, *'test.exe'* process was included for the analysis. Each process is uniquely identified by its page directory base address, which is *0X04660000* in this case. The page directory address ensures that each process has its own virtual memory space. The page directory address of currently running process is stored in page-directory base register (PDBR) [Bay]. For the analysis subject *'test.exe'*, we see the log file has record of the loading of *ntdll.dll* (module that contains Windows NT system functions [Proa]) and *RPCRT4.dll* (Remote Procedural Call API used by Windows application for network and Internet communication [Prob]) in Line 5 and Line 13 respectively. In Line 8–12, we can see the log for different native system calls such as *NtOpenKey, NtDeleteKey,NtCreateKey* related to opening, delete, and creating registry key respectively. With the log, Anubis generates the report of the analysis (resource activities of the analysis subject), and also keeps the record of the analysis report and the binary file, in the relational database for further record. We had access to those databases. The report database has record of resource activities of *File, Registry, Service, Process, and Network* for operations *create, delete, modify, read*.

However, the old database was missing the resource activities that were not successful; for an example, failed attempt by malware to delete a file. For each run, Anubis analyzed malware one at a time in a total new OS environment; so the resources created during one analysis run were not present during the analysis run of another malware. Because of this, malware trying to detect or delete the resource created by some other malware would be unsuccessful. A new database with record of such failed attempts was essential.

Listing 3.2: Snippet of Anubis Log

```
1  Anubis Logfile Version: 1.1
2  Starting Anubis at 02/04/16, 02:22:23 UTC.
3  Logging all function calls: 0
4  00:00:18.389876 [ANALYSIS_INFORMATION]: X X Added all of ntdll.dll's
       exported functions to the function Addresses Map.
5  00:00:18.390846 [ANALYSIS_INFORMATION]: X X ntdll.dll was loaded.
6  00:00:18.391247 [ANALYSIS_COORDINATOR]: X X Going to include test.exe
       with PDB 0x04660000 in the analysis.
7  00:00:45.363931 [ANALYSIS_COORDINATOR]: "test.exe" 0 1 Event 'First
       instruction of the process' happened.
8  00:00:45.365954 [FUNCTION_COORDINATOR]: "test.exe" 0 1 Added callback for
        function NtOpenKey (7C90D5B0)
9  00:00:45.366673 [FUNCTION_COORDINATOR]: "test.exe" 0 1 Added callback for
        function NtDeleteKey (7C90D230)
10 00:00:45.367363 [FUNCTION_COORDINATOR]: "test.exe" 0 1 Added callback for
        function NtCreateKey (7C90D0D0)
11 00:00:45.555742 [ANALYSIS_INFORMATION]: "test.exe" 0 1 Added all of RPCRT
       4.dll's exported functions to the function Addresses Map.
12 00:00:45.556358 [ANALYSIS_INFORMATION]: "test.exe" 0 1 RPCRT4.dll was
       loaded.
```

With the log of execution trace and system calls, Anubis also creates the "behavioral profile", described in subsection 3.1.1, for the submitted malware sample, which has record of both the successful and failed resource activities. The "behavioral profile" of malware sample was used to get the resource name (object-name)[see section 3.1.1] and operations (operation-name)[see section 3.1.1] with successful or failed status, to create the new database. The new database was created for the above discussed 8 types of resource (*File, Registry, Service, Section, Job, Process, Job, Sync, and Driver*) and operations were generalized to *modify, read, and delete*. The implementation is described in section 4.4.

3.2.2 Reverse Indexing

After we created the database, we created reverse index of the resource activities. The database had all the resource activities (failed and success) of the resource types: such

as file created, file deleted, registry modified, registry accessed and so on. We mapped 'resource name', with all the malware, that had operations on that 'resource name'. For instance, we mapped the resource name of the *files*, with all the malware that modified it. Same reverse index was generated for read and delete operations, and for other resource types, *registry*, *Section*, and others. With the reverse index, for each resource, we have a list of all the malware that modified, accessed, or deleted that resource. We used the reverse index in our heuristics to select candidate malware pairs.

3.2.3 Heuristics

We mapped reverse indexes, based on common resource name, to get malware set, say set *'A'*, that created a resource *'r'* with another malware set, say set *'B'*, that tried to delete or read the same resource *'r'*. As described in section 2.1, malware families have different ways of infecting and taking control of system, peculiar to themselves. One can detect the presence of malware in the system, by checking the presence of unique resource (files, registries, and others) associated with that malware, in the system. Thus, when a malware tries to access or delete some resource created by another malware (with a failed attempt because each malware was analyzed by Anubis in new OS environment), that activity is interesting for our candidate selection.

We selected malware sample from each of those set *'A'* and *'B'*, to get a candidate pair, with interference corresponding to the resource *'r'*. For an instance, if (x_1, x_2) belonged to set 'A' and (y_1, y_2) belonged to set 'B', then the candidate pairs related to resource 'r' are: (x_1, y_1), (x_2, y_2), (x_2, y_1), and (x_2, y_2). With initial testing, we found that a single resource name was created by tens of thousands of malware and being accessed by tens of thousands of another malware. Large number of candidate pair would be generated even for single resource with this approach. We had to lower the number of candidate pair to get substantial number that we can test for behavioral interference. Also, the candidate pair were chosen only based on common resource, and did not represent different families. We use clustering of malware based on the similarity of behavioral profile to address these problems.

3.2.4 Clustering

Malware variants belonging to same family, will have similar behavior pattern (code semantics) despite of code obfuscation. With clustering, we group such similar behaving malware into same cluster (family).

Many previous research based on malware clustering were done for small number of malware samples [see section 2.4]. We needed an approach to cluster millions

of malware samples. We modeled malware clustering to *document clustering* and used *LDA* for its low memory footprint, $O(\#words \times \#clusters)$, compared to *tf-idf*, $O(\#documents \times \#words)$. We use *Gensim*—python library—to implement the document clustering.

Latent Dirichlet Allocation

A single resource activity—such as 'file read', 'registry delete' or 'service modify'—would count as single *'word'*. All such resource activities of a single malware sample, is the total 'words' in the single document (malware). We created the corpus for document clustering with all the resource activities of all the malware dataset. We gave each resource type, operation name, and resource name a unique numeric code to represent the resource activity as a word. The number coding is given in Listing 3.3

Listing 3.3: Numeric codes given to resource and operation

```
RESOURCE_CODE = {"file" : "1", "registry" : "2", "section" : "3", "
    service" : "4", "driver" : "5", "sync" : "6", "process" : "7", "job" :
     "8"}
OPERATION_CODE = {"access" : "1", "delete" : "2", "modify" : "3"}
```

A file delete activity of filename, *'c:\\gbot.exe'*, with *file_name_id*, "4986" in our database, by some malware "A", would be represented as single word *"1_2_4986"*. With the corpus created, we use *Gensim* (python library) to perform the document clustering.

Gensim

"Gensim" [ŘS10] is an efficient python library. We preferred using it because of its simplicity, well documented API, and ability to work on large corpus. We use the *multicore* models of *Gensim* for scalability as it utilizes the multi cores processor of the machine efficiently with parallelization making the clustering process faster [Genb]. The document can be fed sequentially to the algorithm making it possible to input large number of documents with the help of iterator.

3.2.5 Candidate Selection

In this section, we illustrate the candidate pair selection algorithm based on malware clustering. We select the candidate pair in such a way that two malware associated by an interesting resource (resource created by one malware sample and tried to delete/access with failure by another), fall in different cluster.

Let, R, be a set of candidate resources such that each resource "r" in R have some malware set that create it (say set A_r) and some other set of malware that try to (unsuccessfully) access/delete it (say set B_r). We combine all such sets A_r and B_r corresponding to "r" in R to sets A and B, respectively. We combine sets 'A' and 'B' and cluster them to cluster ids $[c_1, c_2, \ldots. c_n]$ (n is number of family) such that any malware sample x in (A union B) can be tagged/mapped to cluster id $C(x)$, where $C(x)$ belongs to $[c_1, c_2, \ldots c_n]$. After the clustering is done we select candidate as shown in algorithm 1.

Algorithm 1 Candidate Selection

1: R = Set of all interesting resource
2: A_r = Set of malware that creates a particular resource 'r'
3: B_r = Set of malware that delete/access (failed) particular resource 'r'
4: N = Maximum number of families to consider
5: E = Set of all probable candidate
6: **function** C(j)
7: c_j = cluster id that malware j belongs to
8: **return** c_j
9: **end function**
10: **for all** $r \in R$ **do**
11: **if** $|C(x_r) : x \in A_r| > N \vee |C(y_r) : y \in B_r| > N$ **then**
12: **continue**
13: **end if**
14: **for all** $(x_r, y_r) \in A_r \times B_r$ **do**
15: **if** $C(x_r) \neq C(y_r)$ **then**
16: $E \leftarrow (x_r, y_r)$
17: **end if**
18: **end for**
19: **end for**

In algorithm 1, we define a function $C(j)$ in line 7–10 which returns the cluster id of given malware j. In line 11–20, for all resource r in set R, we first check if the number of create or access/delete families associated with the resource r is greater than N. We choose N accordingly to filter resource associated with too many families. If true, we simply discard it, as resource modified/deleted by too many families is less interesting. If false, for every possible malware pair between the create set A and access/delete set B, say (x_r, y_r), we populate our experiment set E, such that x_r and y_r belong to different family. The final experiment set E is a set of such (x_r, y_r) for all resources r in R.

We decrease the number of candidate pairs with this approach, as we no more

take into consideration all the possible candidate pair from tens of thousands of malware sample associated with resource, but select candidate pairs based on cluster those malware sample belongs to. For an example, let r_1 be the resource which was created by malware samples x_1, x_2, x_3 and was tried to delete with failure by malware samples y_1, y_2, y_3. Let, x_1, x_2, y_1, y_2 malware samples belong to cluster c_1, x_3 malware sample belongs to cluster c_2, and y_3 malware sample belongs to cluster c_3. With previous heuristics, the total candidate pairs would be all possible pairs from the cross product of both set such as $(x_1, y_1), (x_1, y_2), (x_1, y_3), \ldots, (x_3, y_3)$. With candidate selection algorithm, we would only get two probable candidate pairs corresponding to resource r_1: (x_1, y_3) and (x_3, y_3), such that both malware in the pair belong to different family.

3.2.6 Running the Candidate

Our chosen dynamic analysis system [see subsection 2.3.1], *Anubis*, analyzes only one binary at a time. We needed a way to be able to run two malware sample together in th Anubis. For that, we used the fact that Windows PE executable loader works fine even with extraneous concatenated data at the end of valid executable file. We created *'packer'* and *'unpacker'*, based on the above fact, to execute both malware of a candidate pair, packed together as single binary, with time interval in between two execution.

The **unpacker** is a self reading dropper binary. With **packer**, we append binary of the candidate pair, to the back of *unpacker* binary, one after another in sequence. Followed by the binary, meta-information, such as size of both binary and preferred time delay, was appended to the end of the *unpacker*. We analyze the packed *unpacker* binary in the *Anubis*. When executed, the *unpacker*, would read itself from the behind to get the meta-information, and recreate the packed binaries, and drop them inside the *Anubis* OS environment. After both the binaries are dropped, it execute the first binary, waits for specified time delay, and then runs another binary. The sequence of the binary execution is: candidate malware that creates the resource first, followed by candidate malware that tried to access/delete the resource with failure. In the end, both the binary runs in the *Anubis*, and we analyze the Anubis report. In section 4.2, we implement the *packer* and *unpacker*.

3.2.7 Result Analysis

A binary submitted to Anubis is renamed into *'sample.exe'*, and is considered as the primary analysis subject. All the binaries created and executed by the primary analysis subject, after its execution, is also tracked for its resource activities. We analyze the resource activities of our candidate pair and any other process started by them. We are

interested in 'access/delete' activity of the 'resource', based on which the candidate pair was chosen—'resource' created successfully by one malware, and failed attempt to access/delete the same resource by another. Any successful access/delete of that resource (files or registries or others) will be the change in behavior of one malware caused by the presence of another malware.

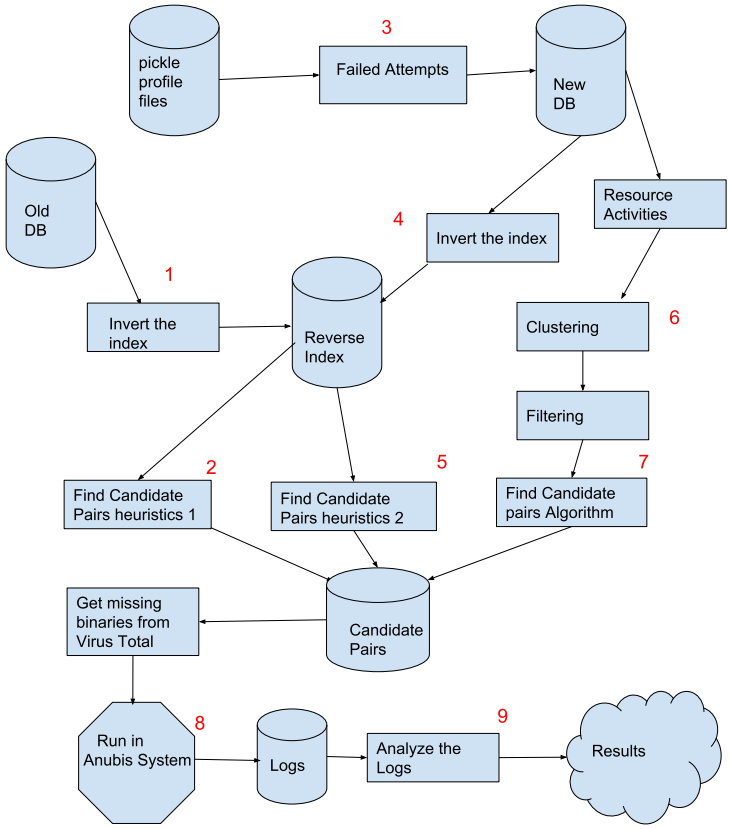

Figure 3.1: Overview of the research and experiment

3.3 Summary

In this chapter, we described different terms used in our work in details and also gave an overview of our work flow. In next chapter, we will show how we implemented the work flow in our system.

4 Implementation

In this chapter we give a detail explanation of how we implemented and conducted our research as described in chapter 3. We start with study of old database structure and creation of reverse index in section 4.1. We describe the implementation of the *packer* and the *unpacker* to run the candidate pair inside *Anubis* in section 4.2 and in section 4.3 we run our preliminary test of candidate pairs. Because the old database did not had record of failed activities, we create the new database with both success and failed resource activity in section 4.4. The implementation of LDA based on resource activities from the newly created database is describe in section 4.5. Finally, in section 4.6, we describe the use of max-flow approach and heuristics to find the optimal sets of candidate malware pairs corresponding to interesting resource.

4.1 Old Database and Reverse Index

Our work was based on wide variety of millions of malware samples, collected over the years, by *Anubis*. We had to effectively and efficiently analyze those to find the behavioral interference between malware families.
We started with studying the Anubis system and its database. We primarily dealt with two database of Anubis backend, the **'db_report'** and **'web_analysis'**. The *'web_analysis'* database was hit first for any binary submitted for analysis from public web interface. Some of the important tables to our work were *'result'*, *'file'*, *and 'file_task'*. Each submission is a unique row data in *'result'* table. A new row is also created in *'file'* table with the *'md5'* and *'sha'* hash of the submitted binary. We refer the unique primary key of the *'file'* table as **malware id**. The *file* and *'result'* tables are associated through *'file_task'* table. The analysis of the sample would be done for behavioral activities related to different resource types such as File, Registry, and Mutex. These activities were saved in '*db_report*' database. The resource activities of the malware in *'db_report'* database was associated with the *'web_analysis'* database by the constraint key *'result_id'* of the *'result'* table. In Listing 4.1, in the first line, we show a simple *sql* statement to get the *result_id* and *md5* of the binaries submitted to Anubis from the *'web_analysis'* database. In the second line, we get the names of the files created by the binary whose *result_id* is '*12345*' from the *'db_report'* database.

Listing 4.1: sql showing database structure to get file created activities of a malware sample

```
1 SELECT result_id, md5 FROM web_analysis.result join web_analysis.
    file_task using (task_id) join web_analysis.file using (file_id) WHERE
     task_id = result_id;
2 SELECT name from db_report.file_created join db_report.file_name using (
    file_name_id) where result_id = '12345';
```

The total number of malware samples that we had in our test database were **22,154,180**. Initially we considered 3 resource types *File, Registry, Mutex*, and queried the database for *create, read, delete* resource activities. For each resource type, their *create, read, delete* resource activities were queried. Our first step was to create a reverse index from the database so that we could get the list of malware that created/deleted/read the resource. Since, the number of malware to process were in millions, trying to save all their activities in a data structure would cause our machine to run out of memory and crash. We approached this problem with map reduce technique. We distributed the work by processing smaller smaller chunk of data and later merged them together to get the final results.

We worked on a batch of 50,000 malware at a time, and saved the reverse index of activity to the file with a numbering for each batch. The resultant numbered files were in the format where resource name and list of unique malware ids were separated by commas (,) as delimiter. The numbered files of each resource types were sorted, and then joined, to get the final reverse index. The sort and join operation is shown in Listing 4.2 and was fast as it was a merge sort $O(n \times \log(n))$. Comma is the field separator for both commands denoted by option *'-t'* and the operation was performed on first field denoted by option *'-k'*, which is the resource name.

Listing 4.2: Sort and join the reverse index

```
LANG=en_EN sort -t, -k 1,1 $file_name
LANG=en_EN join -t , -a1 -a2 $file_name1 $file_name2
```

Listing 4.3: Sample of reverse index created for File activity

```
C:\mbr.exe,189524063,184501719,87504631,86763863
Buttons,111448211
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/telnet.exe
    ,178046895,174206059,183601891,89650247
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/1.jpg,161552035,116241803
```

A snippet of reverse index of file created activity is shown in Listing 4.3, where resource type *'file'*, with resource name *'mbr.exe'*, is created by malware with ids *189524063, 184501719, 87504631, and 86763863*.

From the reverse index of the resource activities of the Malware, we mapped created activities against the deleted or read activities. We started looking for one to one interaction of malware to a single resource. We looked for a set (a, b), where malware 'a' creates the resource r, and malware 'b' deletes the same resource, r, such that no other malware has create or delete activities on that resource r. We ran those malware pairs in the Anubis system using the *Unpacker*.

4.2 Packer and Unpacker

In order to run the pair of malware together inside the Anubis environment we made a Win32 console application (Anubis uses Windows XP as underlying OS.), named **Unpacker**. We used the fact that we can append any extraneous data at the end of Windows PE executable, and it would not affect the executable's program flow. We wrote a **Packer** program that would add the candidate binary pair at the end of the *Unpacker* binary and then further appends the time delay and file size of each candidate binary as meta information. The structure of the Unpacker binary is shown in Figure 4.1. The *Unpacker* binary, when executed, would read itself from the back to get the meta information and read candidate binaries bytes. It will then create the candidate binaries from the read bytes, and run both binaries with a time delay given.

Figure 4.1: Structure of the Unpacker binary that would create the candidate pair and run them with delay.

Code snippet of the *'Unpacker'* binary is shown in Listing 4.4. We used a struct of 3 integer size to read and hold the meta information. The size of last three *unsigned int* byte of *Unpacker* has the binary pair sizes information and the delay time, known as meta information. The offset for meta-information was calculated by deduction of $3 \times$ *size of 'uint'* from the total size of itself (Unpacker). We calculate the starting position of appended binaries, and use `'fseek'` function to set the file position pointer

there. We then start reading the bytes until the exact size of binary is read. The read bytes were then saved to create new binary file. We used *windows.h* standard library's functions: *'CreateProcess'* function to execute the binary, and *'Sleep'* function for delay in between two execution.

Listing 4.4: snippet of Unpacker.c file

```
1  /* stuct for storing meta information */
2  typedef struct {
3    unsigned int delay;
4    unsigned int fsize1;
5    unsigned int fsize2;
6  }meta_info;
7
8  /* reading the meta information and first binary */
9  rfp = fopen(argv[0], "rb");
10 wfp1 = fopen(fileName1, "wb");
11
12 fseek(rfp,0,SEEK_END);
13 size = ftell(rfp);
14 offset = size - sizeof(meta_info);
15 fseek(rfp, offset, SEEK_SET);
16 fread(&info, 1, sizeof(info), rfp);
17
18 /* calculate the unpackersize from the offset and files size. */
19 unpackersize = offset - (info.fsize1 + info.fsize2);
20
21 /* rewind back and to the point of the start of file1 */
22 fseek(rfp,0,SEEK_SET);
23 fseek(rfp, unpackersize, SEEK_SET);
24
25 nread_sofar = 0;
26 while (nread_sofar < info.fsize1) {
27     nread = fread(buf, 1, min(info.fsize1 - nread_sofar, sizeof(buf)), rfp
          );
28     nread_sofar += nread;
29     fwrite(buf, 1, nread, wfp1);
30 }
31 fclose(wfp1);
```

4.3 Initial Experiment

During our initial run of candidate pairs, we found lots of dropper malware causing this interaction. For a candidate pair (a, b), binary 'a' was a dropper that would create many binaries including the binary 'b', and binary 'b' actually read itself upon execution, which was recorded as read activity by Anubis. After running many other candidate pairs and analyzing the results manually, we were able to understand the Anubis report in depth and also found a logical error on our current approach.

Our notion behind checking the malware interaction was finding candidate pairs such that one malware 'a' creates some resource, 'r', and another malware 'b' tries to access or delete the same resource, 'r'. But, Anubis ran each submitted binaries in an isolated environment; hence there was no way that a malware sample 'b' would find the resource that was supposed to be created by malware 'a'. We should have been looking for failed attempt activities, where a malware sample unsuccessfully tries to access or delete a resource created by another malware. But, the current database had no record of such failed attempt. This lead us to look for the alternatives where we could find log of such failed attempt activities during the execution of binary.

We parsed the behavioral profiles, as described in subsection 3.1.1, of malware samples to find failed activities of deletion and access.

4.4 Creation of Database

Each malware submitted to Anubis had its own unique behavioral profile saved as python pickle [Pyt] object. We parsed behavioral profile and extracted the object names and operations from the *OS Object* and *OS Operations* respectively. The object name is the name of the resource and operation is the activity performed on the resource. The status of the operation, failed or successful, was taken into account this time. The resource name, type of operation, and status of operation represented the resource activities of a malware. We create a new database with resource activities of all our malware samples.

We considered eight resource type into account while parsing the profile files: File, Registry, Sync, Section, Process, Service, Job, and Driver [see subsection 3.1.2]. Windows Native and Windows API calls were generalized into OS Operation during the creation of behavioral profile [see subsection 3.1.1]. We went through the list of OS Operations [section 3.1.1] and mapped them into the broad three categories *Modify,*

Access, and Delete. The modify, access, and read list for the resource types *File, Registry, Sync, Section, Process, Driver, Job, and Service* are shown in Listing 4.5.

We used **MySql Version 5.5.46** as our database engine. MySQL [MySb] is highly scalable and flexible relational database system which complies with the ACID model [MySa] design principle. We used multiple workers to create the database, and ACID compliance prevented from inconsistency of data when being updated across multiple tables.

Recreating the database was one of the bottleneck in our project and consumed much time. The behavioral profile files had to be accessed via network file system. We need to walk through large list of directories consisting of hundreds of thousands of file, to search the behavioral profile pickle [Pyt] of specific malware. We found *16,031,518* behavioral profile files out of *22,154,180* malware.

While creating the new database, we hit the integer overflow for the primary index key of '*registry access*' table. We did not anticipate the number of records to cross maximum *'int (10)'* limit *4,294,967,295*. We created another "*registry access*' table, with different table name, and start inserting the registry access records in the new table. When accessing the data for later use, we re-factored our program so that we checked both the registry tables in order to find the registry access activities of a malware sample.

We processed 16,031,518 behavioral profiles. The final database size was 1.2 Terabyte. About 62% of execution time was taken to load the gzip pickle [Pyt] of malware profile and the normal SQL CRUD (create, read, update, and delete) operation, which can be seen in Figure 4.2.

After the creation of new database, we made a reverse index, like before, for the '*resource name*' to '*malware ids*' modifying, accessing, and deleting the resource. This time the refined database had both the success and failed resource activities. The reverse indexes were used to map resources that were successfully modified with resources that were read/accessed with failed attempt. For instance, we made a mapping between file successfully modified and failed file deletion, and mapping between file successfully modified and failed file access. We made same mapping for all other 7 resource types. All the mapping was done based on the common '*resource name*'.

With the reverse index mapping, we chose the candidate pairs such that one malware created the resource and another malware made a failed attempt to delete/access the same resource. But, the number of possible pairs were too many with this simple heuristic and we hit the *n-combination* problem. A single resource `'r'` would have been

Listing 4.5: Mapping of generalized OS Operation from behavioral profile

```
MODIFY_LIST = {
        file: ["create_named_pipe", "create_mailslot", "create", "rename",
             "set_information", "write", "flush_buffer", "map"],
        registry: ["create", "restore_key", "save_key", "set_value", "
            set_information", "mem_write"],
        process: ["create", "set_information", "suspend", "resume", "unmap
            ", "map"],
        job: ["assign", "set_information"],
        driver: ["unload", "load"],
        section: ["create", "map", "unmap", "mem_write", "set_information"
            ],
        sync: ["create", "map", "set_information", "mem_write"],
        service: ["create", "start", "control"]
        }

ACCESS_LIST = {
        file: ["query_file", "query", "open", "query_directory", "
            query_information", "read", "monitor_dir", "query_value"],
        registry: ["enumerate", "enumerate_value", "monitor_key", "open",
            "query", "query_value", "mem_read"],
        process: ["open", "query"],
        job: ["open", "query"],
        driver: ["query"],
        section: ["open", "query", "mem_read", "read", "query_file", "
            query_system"],
        sync: ["open", "query"],
        service: ["open"]
        }

DELETE_LIST = {
        file: ["delete", "open_truncate"],
        registry: ["delete", "delete_value"],
        process: ["delete"],
        job: ["delete"],
        driver: ["delete"],
        section: ["delete"],
        sync: ["delete"],
        service: ["delete"]
        }
```

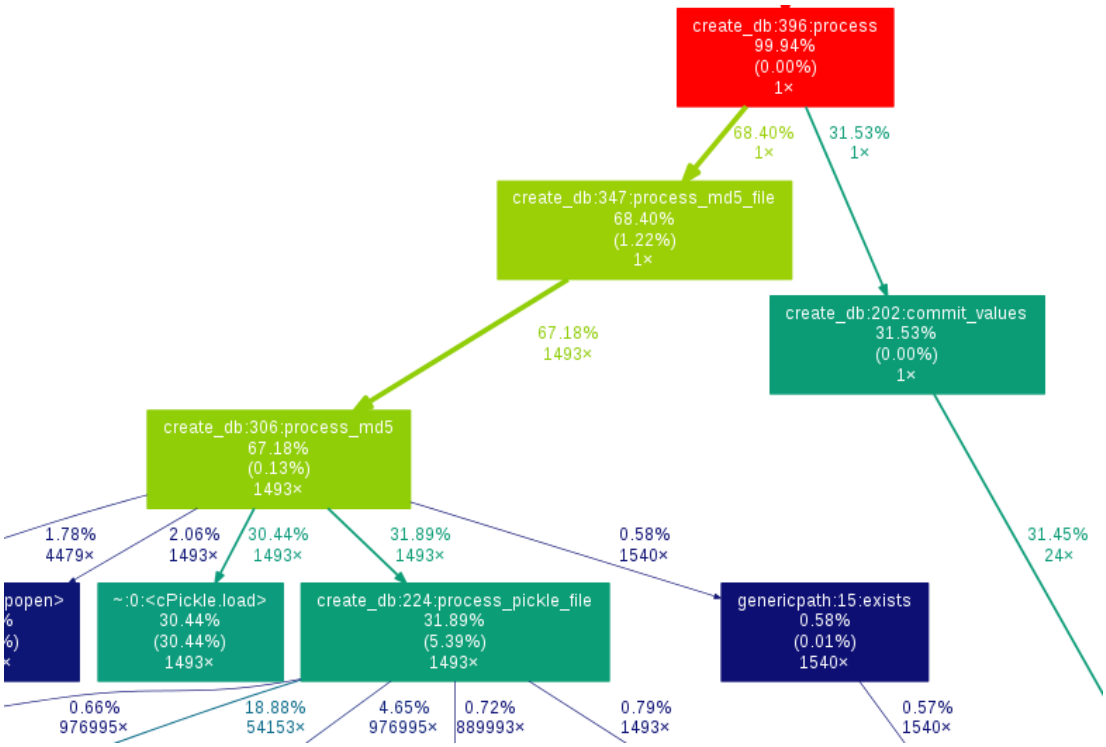

Figure 4.2: Profile of new database creation

created by tens of thousands of malware and another tens of thousands of malware trying to access/delete it unsuccessfully. The combination of candidate pairs from the candidate sets with thousands of malware are of large number to run and analyze result.

We wanted to filter our candidate sets without compromising the quality. We performed clustering of the malware based on their resource activities (behavior) to address this problem. Malware dataset were clustered into different families and candidate pairs were chosen such that they belong to different families.

4.5 Document Clustering

Before the clustering, we filtered our malware dataset from the reverse index mapping by removing all resource activity that we confirmed as benign based on our benign list and knowledge. We sorted the reverse index (resource name to malware ids) with respect to the count of malware ids associated with the resource name in descending order. With this, we had most commonly modified/accessed/created resource at the

top of the list. We went through the resource name until we found some interesting resource name which does not look benign. We excluded all benign resources preceding the first interesting suspicious resource name. For instance, file resource name such as *'Dr. Watson.exe'* and *'Sample.exe'* were some resources with highest number of malware interacting with them. The reasons are *'Dr Watson'* file is created whenever a binary crashes and *Sample.exe* is the name, the submitted binary gets renamed to, by Anubis. We filtered all such benign resource activities related to resource types File, Registry, Sync, Section, Process, Driver, Job, and Service. After the filtering, the total number of malware samples were 7,362,635.

We created the text corpora for those 7,362,635 malware, and performed the document clustering. We converted all the resource activities of the malware samples into the corpora text as discussed in subsection 3.2.4. Each resource activity was represented as word, each malware was represented as the document, and the collection of documents were text corpora. The process was database intensive; as seen in Figure 4.3, about 94.50% of the run time was spent on executing the database cursor (traversal, addition, retrieval of database records). It took almost 7 days for us to get complete text copora. The size of resultant text corpora was about 81 Gigabytes.

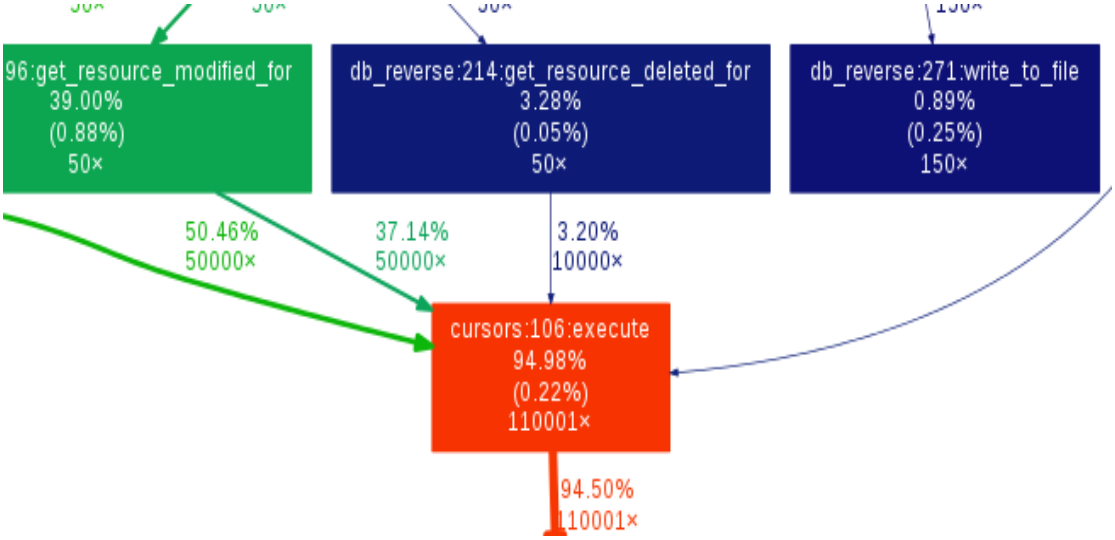

Figure 4.3: Profiling of building a corpus

4.5.1 LDA Model

We used python module Gensim [ŘS10, Gensim] for the clustering and used the LDA multicore model which we described in section 3.2.4. The LDA model generation code snippet is shown in Listing 4.6. The corpora consisted of one document per line. We created a dictionary from the corpora using Gensim *'corpora.Ditionary'* module [Gena]. Dictionary is representation of all the unique tokens (words) by a unique token-ids. We filtered the dictionary by discarding any resource activity that occurred in less than 10 or more than 1 million for clustering, as those activities are too less or too common to take into consideration. We kept only 200,000 tokens after the filtering. This is done in line 7 [Listing 4.6], where the arguments passed into *'filter_extremes'* method are minimum threshold (which is 10), maximum threshold in fraction (which is 0.14: 1,000,000 out of total 7,362,635 documents), and total number of frequent words to keep after filtering (which is 200,000). In line 9–12 [Listing 4.6], we make an iterator to read the corpus. The iterator operates on each line, which represents a document, of corpus text file and yields a bag of words of the document. Bag of words is a list of two-element tuple, which represents word (token-id) and its count (number of occurrence) in document. In line 15 [Listing 4.6], we create a LDA model for our corpus with 1000 topics. The LDA gives the probability of different clusters that a document can belong to and we assign the document to the cluster with highest probability.

The number of clusters created were **"662"**. The largest cluster size was 227,940 and the lowest was 1. The histogram and cumulative distributive graph showing the sizes of cluster can be seen in 4.4, 4.5, and 4.6. From the graph we can see that distribution of cluster size was mostly below 50,000. 90% of the cluster, about *629 out of 662*, falls under the cluster size of less than 50,000. Similarly, *4 million* malware, which is the 65% of total malware clustered, are in the cluster size of less than 50,000.

4.5.2 Inter-Distance and Intra-Distance

We calculated the distance based on the average number of common words between the malware pair. **Intra-distance** represents the average number of common words between the malware pair from same cluster. **Inter-distance** represents the average number of common words between the malware pair from different clusters. Since we want to cluster malware with similar behavioral activity together in same cluster; the intra-distance should be high (higher similarity among malware samples of same cluster), and the inter-distance should be low (lower similarity among malware samples from different cluster).

Listing 4.6: Script to run Gensim LDA

```
from gensim import corpora, models
import sys
document_name = sys.argv[1]
# read the file with malware activities and create dictionary
dictionary = corpora.Dictionary(line.split() for line in open(
    document_name))
# filter the dictionary for extreme
dictionary.filter_extremes(no_below=10,no_above=0.14,keep_n=1000000)
# create a iterator corpus as the file is large 80GB
class MyCorpus(object):
    def __iter__(self):
        for line in open(document_name):
            yield dictionary.doc2bow(line.split())

corpus = MyCorpus()
lda = models.ldamulticore.LdaMulticore(corpus=corpus,id2word=dictionary,
    num_topics=1000)
```

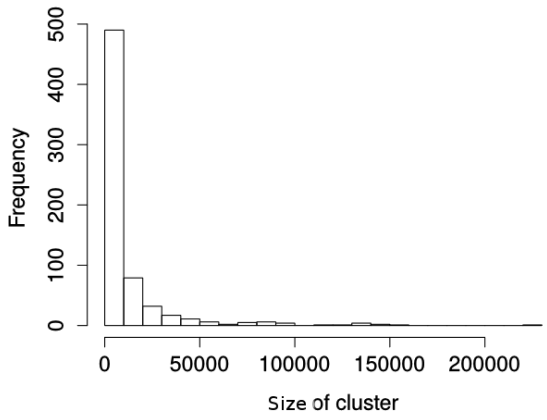

Figure 4.4: Histogram showing the distribution of Cluster size

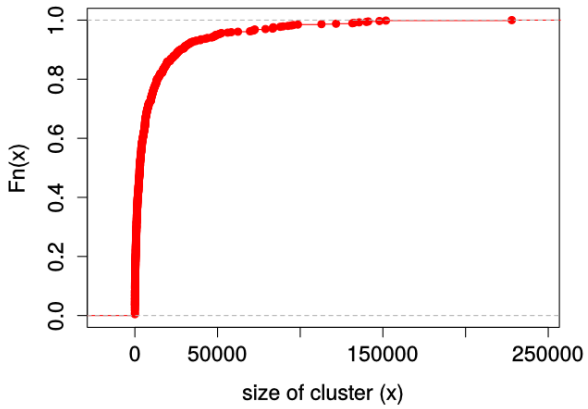

Figure 4.5: CDF between size of cluster and topic fraction

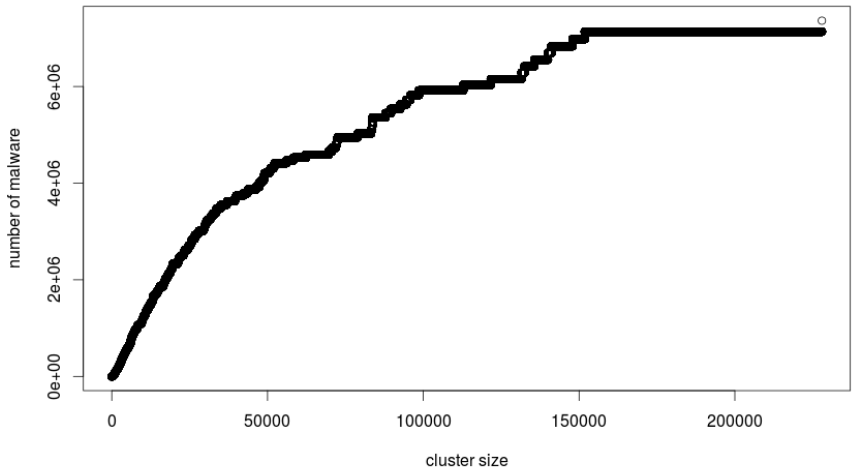

Figure 4.6: CDF between size of cluster and total number of malware

To show the quality of our clustering we provide the following graph showing the graph of inter-cluster and intra-cluster distance in Figure 4.7 and Figure 4.8.
In case of intra-distance, for a cluster, we randomly selected 1000 malware pair (if possible because there were some clusters of size as small as 1), from the combinations of malware samples belonging to that cluster. We calculated the average number of common resource activities*(words)* present among the malware*(document)* pair. We can see that the average common words between the intra malware sample were good with intra-distance of at least *500* covering 80% of family topics [Figure 4.7]. For the inter-distance calculation, a cluster was paired with other clusters, and from those cluster pair, we randomly selected 1000 malware pair (if possible because there were some clusters of size as small as 1). We calculated the average common resource activities between those malware pairs to get the inter-distance. We can see that the inter-distance of only 10 was covering almost 90% of family topic [Figure 4.8].

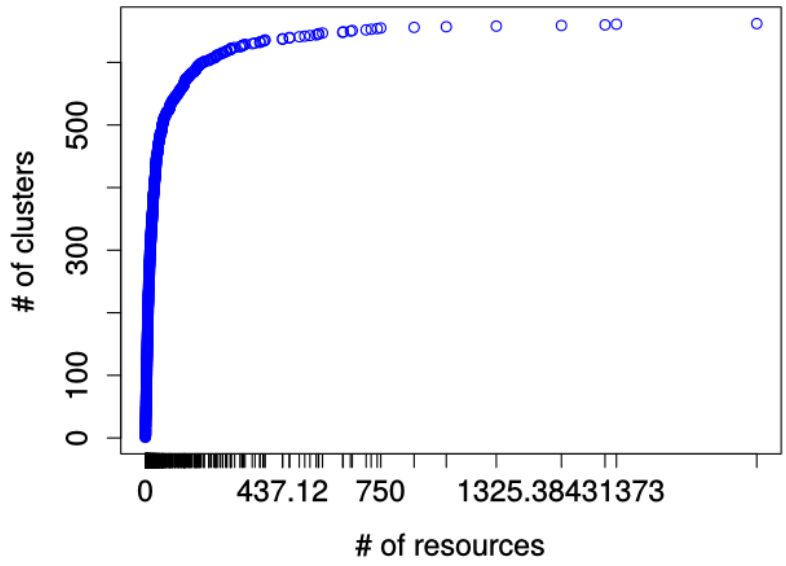

Figure 4.7: Graph showing cdf distribution of common resource between same family topic

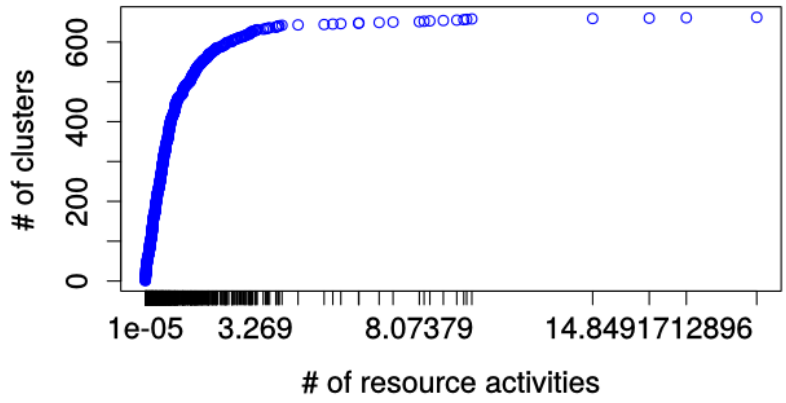

Figure 4.8: Graph showing cdf distribution of common resource between different family topic

4.6 Finding Optimal Candidate Pairs

After the clustering, tens of thousands of malware associated with the resource, would now be reduced and represented by hundreds of families (clusters), that those malware belong to. As described in subsection 3.2.5, we chose the candidate pair corresponding to a resource, such that each malware in the pair belongs to different clusters. In order to find the optimal set of candidate pairs we interpreted the interaction of malware and resource, with respect to cluster, into Flow Network.

4.6.1 Max Flow Approach

We wanted to find the optimal pair of malware such that we have all the interesting mapped resource covered along with the cluster associated with it. We represented the mapping between the resource, malware, and cluster as a flow network, and run the max flow algorithm as given in Figure 4.9.

The network is made of 4 layers, other than sink and source. Layer one is read/delete malware, the malware samples that tried to access or delete interesting resources but failed. Layer two is combinations of clusters (*ic=input-cluster*), that the malware with failed access or delete attempt belongs to, and the resource on which the malware tried to operate. Layer three is the combinations of clusters (*oc=output-cluster*), that the

malware with successful modification operation belongs to, and the resource that they could successfully modify. Layer four is the malware that successfully modified the resource.
The flow rules were:

- All malware from layer one is connected from sources and also connected to the input cluster and resource combination it belongs to.
- All input-cluster and resource combination in layer two is connected to all output cluster resource combination in layer 3 that has a matching resource.
- All output-cluster and resource combination in layer 3 is connected to the corresponding malware in layer 4.
- Each malware in layer 4 is connected to the sink (T).

The capacities were:

- All connections have capacity of infinite except connections from layer 2 to layer 3.
- All connections from layer 2 to layer 3 have a capacity of one.

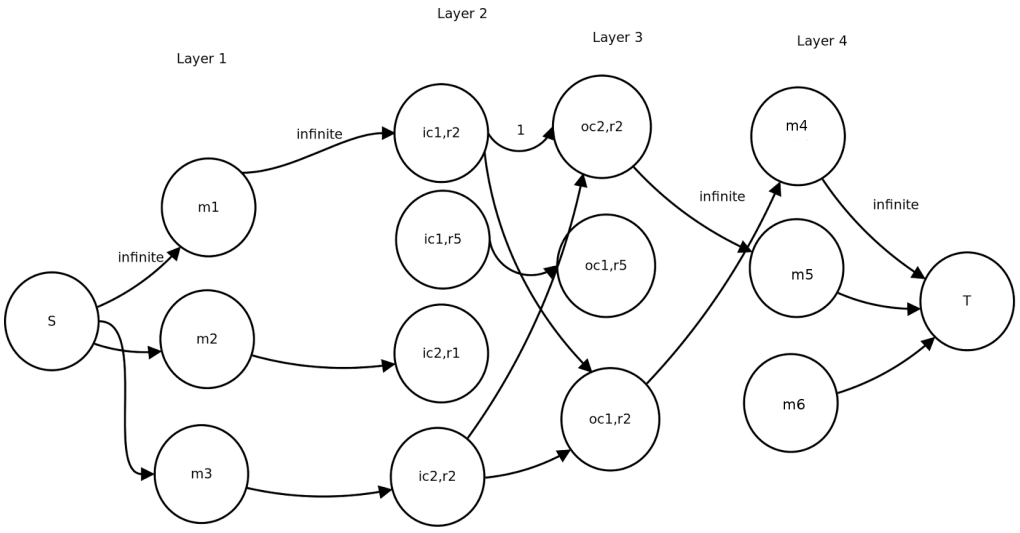

Figure 4.9: Graph representing the max flow implementation

The maximum flow in this network corresponds to an optimum match up. To explain, in Figure 4.9, 'm_1' in layer 1 is connected to (ic_1, r_2) in layer 2. There are two

cluster-resource combination, (oc_2, r_2) and (oc_1, r_2), in layer 3, that has resource 'r_2' common with (ic_1, r_2) cluster-resource combination. So our candidate pairs would be, (m_1, m_5) and (m_1, m_4) operating on 'r_2'. Path from source (S) to m_2 in layer 1 and (ic_2, r_1) in layer 2 does not reach the sink (T); hence, it is omitted.

The max flow approach gave us optimal candidate pairs such that each pair of cluster is only covered once. However, the approach did not take care of repetition of cluster pair corresponding to different resource. If malware m_1 is belongs to input cluster ic_1 and malware m_2 belongs to output cluster oc_2 and both are associated with resource (r_1, r_2, r_3), same candidate pairs (m_1, m_2), belonging to same cluster pair (ic_1, oc_2), gets repeated three times. We wanted to group those repetition so that we get single candidate pair (m_1, m_2) that covers the unique cluster pair (ic_1, oc_2), for all its corresponding resources $(r_1, r_2, \ldots, r_n)$. We used heuristics approach to eliminate those repetition.

4.6.2 Heuristics Approach

The problem was to find the "minimal set" of malware pairs that covers all unique cluster (family) pairs (dictated by the set of all candidate pairs) and it also covers all resources that help build the candidate pairs, i.e., for each resource *'r'* from resources *'R'*, at least one malware pair from the final set should correspond the resource *r*.

The relations has been represented in the graph below, where, *I* is a set of input clusters from set *'A'* and *'O'* is the set of output clusters from set *'B'*. Set *'A'*, *'B'* are the sets of malware samples successfully modifying the resource and accessing/deleting the resource with failed attempt, respectively. *'R'* is the set of all interesting resources.

In the Figure 4.10, m_1, m_2, m_5 samples create resource r_1, which is accessed (failed attempt) by m_6, m_3, and m_4. The malware m_1 and m_2 maps to cluster c_1. Malware m_6 and m_3 maps to cluster c_4, and so on. We need to find the minimal number of paths from each element c_x of set *'I'* to all reachable elements in *'O'* (reachable from c_x), such that all reachable elements *'r'* in *'R'* (reachable from c_x) are traversed. From such minimal set, we generated the candidate pairs by taking members of *A* and *B* from the path.

We created the data structure as shown in 4.7. The data is a dictionary. All candidate cluster pairs are keys and the value is a dictionary whose keys are malware pairs (all candidate malware pairs belonging to the cluster pair) and values are the list of corresponding resource.

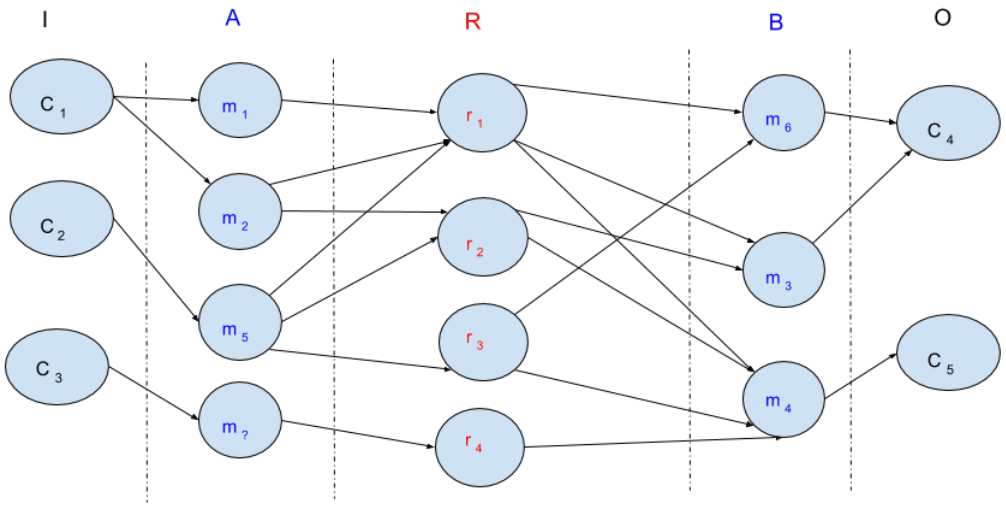

Figure 4.10: Heuristics approach to optimal malware pair selection

Listing 4.7: Database Structure

```
db = { (c_i, c_o) :
    { (m_a, m_b) : [ r1, r2 ,...],
        ...,
     }
}
```

The reduced candidate set was computed as shown in pseudo-code 4.8. It is a greedy algorithm starting from the malware pair that corresponds most number of resource interactions. For each malware pair it checks if that pair has been already chosen, and if not, adds the pair to the candidate list.

Listing 4.8: Pseudo code (Python) to get minimal set of candidates for all resource

```
candidate_set = set()
for c_pair, v in db.iteritems():
    # reverse sort malware pairs by number of associated resources
    x = sorted( [(len(b), a) for a,b in v.iteritems()] , reverse=True)
    r_set = set()
    for c, m_pair in x:
        cur_r = v[m_pair]
        if not r_set.issuperset(cur_r):
            r_set = r_set.union(cur_r)
            candidate_set.add(m_pair)
```

5 Evaluation

Our hypothesis is the existence of behavioral interference between malware samples of different families. So far we implemented the system to find the minimal set of candidate pairs, from our test dataset, that has probability of exhibiting behavioral interference. Now we generate the candidate pairs from the system and execute them together in the Anubis system for analysis. We believe that when run together, the malware supposed to create the resource corresponding to the candidate pair will do so, and the second malware will delete that specific resource created by the former malware.

In this chapter, we first give a detail break down of candidate pairs that we generate from our algorithm in section 5.1. Then, we describe the specifications of Anubis system and experiment setup in **??**. We show our results and discuss the effectiveness of our approach and some interesting cases in section 5.3 We elaborate on the threats to validity of our approach in section 5.4. Lastly, we end this chapter with summary in section 5.5.

5.1 Candidate Pairs

To find the candidate pairs, the maximum number of families, the value of '*N*' in Algorithm 1, was set to 10; i.e., any interesting resources with more than 10 corresponding malware families, associated by create or access/delete operation, were discarded. We do this because resource associated with too many families is less interesting [see subsection 3.2.5]. With our approach, in total we found *263,691* candidate pairs with probable behavioral interference. The breakdown of the number is shown in Table 5.1.

The largest number of candidate pairs was *213,171* for resource type *File* and the lowest number was *54* for for resource type *Process*. Registry resource type was second with number of candidate pairs *39,899*. Sync and Section resource type were third and fourth with number of candidate pairs *7,781* and *2,786* respectively. With our data, we can say that file and registry are the most modified resource type, whose presence in the system is checked by another malware.

Although, in our dataset, we had taken resource types File, Registry, Sync, Section, Process, Driver, Job, and Service into consideration, our algorithm did not find any

Table 5.1: Number of candidate pairs w.r.t. Resource types

Resource types	# candidate pairs
File	213,171
Registry	39,899
Sync	7,781
Section	2,786
Process	54
Total	263,691

candidate pair with behavioral interference based on resource type *Job, Device, and Driver*. This infers that in our dataset, there were no two malware sample from two different families, where one malware created a Job, Device or Driver and another malware accessed or deleted it.

5.2 Experiment Setup

In order to run the candidate pairs, we had remote access to 7 instances of Anubis system. Anubis uses Qemu, "a machine emulator which can run an unmodified target operating system (such as Windows or Linux) and all its applications in a virtual machine [Bel]", as emulator component. Anubis modifies Qemu's dynamic translation, runtime conversion of target CPU instruction into host instruction set [Bel], to closely monitor the process under analysis. Anubis does so by invoking its callback routine before every basic block (sequence of instruction ending in jump statement or changing the CPU state) is executed on the virtual processor [Bay]. Each of the 7 instance of Anubis system emulates an entire PC computer system, running complete and unmodified version of Windows XP (with Service Pack 2) as an underlying operating system, to provide an accurate environment for malware experiment [Bay].

Candidate pair were packed together using the *'Packer'* and dropped inside Anubis system for analysis using the *'Unpacker'*, as described in 4.2. Some of the missing binary from the candidate pairs were downloaded from *VirusTotal*. The time delay between the execution of two binaries in the *Unpacker* was set to 4 minutes. This allowed the first binary enough time to perform its resource activities before the second binary starts. The total time for a single Anubis run was set to 10 minutes; 4 minutes for each malware and 2 minutes for PC emulator to boot.

5.3 Results

We sampled malware pairs randomly from the probable candidate pairs list (Table 5.1) and ran them in Anubis system to test for behavioral interference. The log of resource activities from the experiment run was then parsed and analyzed as described in subsection 3.2.7 to detect the behavioral interference between the malware samples. The number of sampled candidate pair and the result of the experiment is shown in Table 5.2.

Table 5.2: Results of candidate pair run

Resource types	# tested pairs	# true positive	prediction accuracy
File	5,000	1032	20.64%
Registry	5,000	731	14.62%
Sync	1,000	119	11.9%
Section	1,000	93	9.3%
Process	54	6	11.11%

5.3.1 Effectiveness

With our results shown in Table 5.2, we can say that our system was able to find malware pair with an average prediction accuracy of 14.25%. Prediction accuracy is the ratio of true positive to tested pairs and average is taken upon all the 5 experimented resource types. Resource type File had the highest prediction accuracy with 20.64% followed by resource type Registry with 14.62%. We can say that in our dataset more malware pair were showing behavioral interference associated with File and Registry. But, this also might be true because we had larger sample tested from these two resource type, 5000 each, compared to 1000 samples in the case of Sync and Section, and only 54 for resource type Process.

5.3.2 Interesting Cases

We studied the report of analysis in details for some of the positive pairs and found some interesting cases. We present some of these interesting case here. The labeling of the generic malware family name we give to the malware sample for these cases are from Virus Total. As discussed earlier, there is inconsistency between the labels provided by different AV vendors. We chose the name based on common name given by most of the AV company.

In one case the malware supposed to delete the resource, does the deletion, but then later creates a new file with the same name but through the rename operation. The malware variants of *Gen:Trojan.Heur.buW@Ijs6jxj* creates the file *C:\windows\system32\svrchost.exe* with its run. When the malware variants of *Gen:Trojan.Heur.RP* was run, it first deletes the current *'svrchost.exe'* file and then renames the file *'fuc1.tmp'*, which was created by it in *Temp* windows folder, to the new name *'svrchost.exe'*.

Another case was with pair consisting malware variants of *Trojan.Win32.Skillis*. The variants of *Trojan.Win32.Skillis* family deleted all the files presented in the *C* drive. Thus, deleting everything including the resources created by the malware sample it was paired with and was run before it. This is a false positive case which our system did not handle.

There were also normal deletion of resource created by malware run first, by the malware run second as expected. We found a case where *Artemis* malware variants creates the file *C:\old.exe* and the file is deleted by malware variants from *Trojan.Win32.Cosmu*.

5.4 Threats to Validity

The results we presented here is in the context of our approach, with dependent variables which we controlled throughout the experiment. In this section we discuss such threats to validity of our results.

During candidate selection, we chose the cutoff number for families to be 10. Different values of N would drastically change the list of candidate pairs [Table 5.1] generated from the candidate selection algorithm [see Algorithm 1]. The change in candidate malware sample pairs will also change the statistics of results that we presented above. We also did not tackle the fact that many resources can have randomly generated names. The exact resource name we considered for the selection of malware pair might not be generated next time, when we run the malware pair for testing. We could have increased the quality of candidate pair selection if we had filter out malware pair with such randomly generated corresponding resource name.

When we ran the experiment on the probable candidate pairs, the number of delay between the execution of two malware was set to be 4 minutes. We chose the delay times after running some initial candidate pairs and recording their time of execution. We chose 4 minutes based on the highest time taken by the malware sample from initial test candidate pairs to finish its execution. The reason was to let the first malware run completely and create all the resources it is supposed to, so that second malware can access it. But, there can be some malware samples that runs more than 4 minutes. We do not know the exact maximum malware execution time. Increasing the time delay arbitrarily would mean increasing the time of experiment for each candidate pair

without certainty to get better results, which we avoided.

In our experiment, we ran the candidate pairs just once and did not repeat the experiment to see if we get the same results. We only ran few malware pair with positive result more than once and got the same result (deletion of resource occurred). Also, we only ran the candidate pair, *(a,b)*, in single order of execution, such that *'a'* runs first, followed by *'b'*, where *'a'* was supposed to create the resource and *'b'* was supposed to access/delete it. We do not know if the behavioral interference persists if we run the candidate pair in reverse order; with *'b'* being executed first followed by *'a'*. We ran some candidate pairs with positive result in reverse order and found that the deletion of resource did not occur if the order of execution was reversed. However, for both of the above cases, we cannot confirm that our results will always be same, as we did not perform the test in all malware pairs with positive result of behavioral interference; because of the time constraint.

Even though we found the concrete evidence where a malware deletes the resource created by another malware, we still do not know the true semantics of the malware. The malware could have deleted just just one single resource that could be very common such as *'Desktop.ini'* file. Also as shown in subsection 5.3.2, a malware could just delete all the files and folder from the drive leading to false positive detection. Further in depth analysis of individual malware belonging to the candidate pair must be done in order to confirm that the purpose of one malware was indeed to negate the infection of other malware from the system.

5.5 Summary

We have shown with our results that our system can find malware pair from different families with behavioral interference. During the experiment, there were many aspects that we overlooked and many variables that we controlled to get the current results. But those threats only challenge the effectiveness of our result, and not the fact that there were malware samples which deletes the resources created by another malware. With this results, we provide strong support to our hypothesis for the existence of such behavioral interference between the malware of different families, and that our system can detect such behavioral interference.

6 Conclusion and Future Work

We have presented a systematic approach to find the behavioral interference between the malware families at large scale. We analyzed the behavioral profile of millions of malware samples to find the probable candidate malware pairs that could interfere with each other. Few thousands candidate pairs were sampled randomly from the total list and then run in the Anubis system, to detect the interference during live run.

Using document clustering, where each malware was represented as a document and their resource activities as words, we clustered the malware dataset into different families. We used heuristics based on common resource to select candidate pairs, such that one malware creates a resource, and another malware tries to delete or access the same resource. Also, the two malware sample of a candidate pair were chosen from two different families.

Our work in its current form has many constraints. As a future work, we need to find efficient way to run the experiment multiple times, while changing the values of variables such as family count cutoff, analysis run time, and sequence of malware pair execution. We also need to extend our system to be able to do thorough static analysis of malware pair, which had positive behavioral interference detection. This will help us know the semantics of malware pair and find if there is true animosity between the malware.

We conclude that the behavioral interference between the malware families exists, and also illustrated an approach to find such case where malware belonging to different family try to detect each others' presence and remove the resources, such as files or registries, created by another malware. This behavior is also one of the traits of environment-sensitive malware.

Acronyms

ACID Atomicity, Consistency, Integrity, and Durability.

API Application Programming Interface.

AV Anti Virus.

C&C Command and Control.

CRUD Create, Read, Update, and Delete.

DDoS Distributed Denial of Service.

DLL Dynamic Link Library.

LDA Latent Dirichlet Allocation.

OS Operating System.

P2P Peer to Peer.

SQL Structured Query Language.

TUM Technische Universität München.

UCSB University of California, Santa Barbara.

VMI Virtual Machine Introspection.

List of Figures

List of Tables

Bibliography

[Anu] Anubis. *Anubis - Malware Analysis for Unknown Binaries*. https://anubis.iseclab.org/. Last Accessed: 2016-01-03.

[AVT] AV-Test. *Malware Statistics*. https://www.av-test.org/en/statistics/malware/. Last Accessed: 2016-01-07.

[Bai+07] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. "Automated classification and analysis of internet malware." In: *recent advances in intrusion detection*. Springer. 2007, pp. 178–197.

[Bay] Bayer, Ulrich and Kruegel, Christopher and Kirda, Engin. *TTAnalyze: A Tool for Analyzing Malware*. https://www.cs.ucsb.edu/~chris/research/doc/eicar06_ttanalyze.pdf. Last Accessed: 2016-02-04.

[Bay+09] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. "Scalable, Behavior-Based Malware Clustering." In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. 2009.

[Bel] F. Bellard. "QEMU, a Fast and Portable Dynamic Translator." In:

[BNJ03] D. M. Blei, A. Y. Ng, and M. I. Jordan. "Latent Dirichlet Allocation." In: *J. Mach. Learn. Res.* 3 (Mar. 2003), pp. 993–1022. ISSN: 1532-4435.

[Bri] British Braodcasting (BBC). *E-mail users caught in virus feud*. http://news.bbc.co.uk/2/hi/technology/3532009.stm. Last Accessed: 2015-12-22.

[Che] Chen, Thomas M. and Tally, Gregg W. *Malicious Software*. https://lyle.smu.edu/~tchen/papers/malicious-software.pdf. Last Accessed: 2016-01-21.

[Cis] Cisco. *What Is the Difference: Viruses, Worms, Trojans, and Bots?* http://www.cisco.com/web/about/security/intelligence/virus-worm-diffs.html. Last Accessed: 2016-01-11.

[CKC12] V. Chipounov, V. Kuznetsov, and G. Candea. *S2E: a platform for in-vivo multi-path analysis of software systems*. Vol. 47. 4. ACM, 2012.

[Diw] Diwakar Dinkar (McCafe Labs). *Japanese Banking Trojan Shifu Combines Malware Tools*. https://blogs.mcafee.com/mcafee-labs/japanese-banking-trojan-shifu-combines-malware-tools/. Last Accessed: 2015-12-23.

[Fed] Federal Bureau of Investigation. *Botnet Bust: SpyEye Malware Mastermind Pleads Guilty*. https://www.fbi.gov/news/stories/2014/january/spyeye-malware-mastermind-pleads-guilty/spyeye-malware-mastermind-pleads-guilty. Last Accessed: 2015-12-22.

[Fir+10] I. Firdausi, C. Lim, A. Erwin, and A. Nugroho. "Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection." In: *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*. Dec. 2010, pp. 201–203. DOI: 10.1109/ACT.2010.33.

[G+03] T. Garfinkel, M. Rosenblum, et al. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." In: *NDSS*. Vol. 3. 2003, pp. 191–206.

[Gena] Gensim. *corpora.Dictionary()*. https://radimrehurek.com/gensim/corpora/dictionary.html. Last Accessed: 2016-01-24.

[Genb] Gensim. *models.ldamulticore*. https://radimrehurek.com/gensim/models/ldamulticore.html. Last Accessed: 2015-12-20.

[Har] Harshit Nayyar. *Clash of the Titans: ZeuS v SpyEye*. https://www.sans.org/reading-room/whitepapers/malicious/clash-titans-zeus-spyeye-33393. Last Accessed: 2015-12-22.

[KVK14] D. Kirat, G. Vigna, and C. Kruegel. "BareCloud: Bare-metal Analysis-based Evasive Malware Detection." In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 287–301. ISBN: 978-1-931971-15-7.

[Lim] Limor Kessem (Security Intelligence). *Shifu: 'Masterful' New Banking Trojan Is Attacking 14 Japanese Banks*. https://securityintelligence.com/shifu-masterful-new-banking-trojan-is-attacking-14-japanese-banks/. Last Accessed: 2015-12-23.

[MCA09] T. Moore, R. Clayton, and R. Anderson. "The economics of online crime." In: *The Journal of Economic Perspectives* 23.3 (2009), pp. 3–20.

[MER08] R. Moskovitch, Y. Elovici, and L. Rokach. "Detection of Unknown Computer Worms Based on Behavioral Classification of the Host." In: *Comput. Stat. Data Anal.* 52.9 (May 2008), pp. 4544–4566. ISSN: 0167-9473. DOI: 10.1016/j.csda.2008.01.028.

[Mica] Microsoft. *Device Object*. https://msdn.microsoft.com/en-us/library/windows/hardware/ff548014(v=vs.85).aspx. Last Accessed: 2016-01-03.

[Micb] Microsoft. *File Objects*. https://msdn.microsoft.com/en-us/library/windows/desktop/aa364395(v=vs.85).aspx. Last Accessed: 2016-01-03.

[Micc] Microsoft. *Job Objects*. https://msdn.microsoft.com/en-us/library/windows/desktop/aa364395(v=vs.85).aspx. Last Accessed: 2016-01-03.

[Micd] Microsoft. *Memory-Mapped Files*. https://msdn.microsoft.com/en-us/library/dd997372(v=vs.110).aspx. Last Accessed: 2016-01-24.

[Mice] Microsoft. *MSDN Library*. https://msdn.microsoft.com/library. Last Accessed: 2016-01-03.

[Micf] Microsoft. *Process*. https://msdn.microsoft.com/en-us/library/windows/desktop/ms684841(v=vs.85).aspx. Last Accessed: 2016-01-03.

[Micg] Microsoft. *Registry*. https://msdn.microsoft.com/en-us/library/windows/desktop/ms724871(v=vs.85).aspx. Last Accessed: 2016-01-03.

[Mich] Microsoft. *Section Objects*. https://msdn.microsoft.com/en-us/library/windows/hardware/ff563684(v=vs.85).aspx. Last Accessed: 2016-01-03.

[Mici] Microsoft. *Service Object*. https://msdn.microsoft.com/en-us/library/windows/desktop/dd389245(v=vs.85).aspx. Last Accessed: 2016-01-03.

[Micj] Microsoft. *Service Overview*. https://technet.microsoft.com/en-us/library/cc783643(v=ws.10).aspx. Last Accessed: 2016-01-19.

[Mick] Microsoft. *Sync Objects*. https://msdn.microsoft.com/en-us/library/windows/desktop/ms686364(v=vs.85).aspx. Last Accessed: 2016-01-03.

[Micl] Microsoft. *Win32/Sality*. https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Virus%3aWin32%2fSality. Last Accessed: 2015-12-22.

[Micm] Microsoft. *Win32/Zbot*. https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Win32/Zbot. Last Accessed: 2015-12-22.

[Micn] Microsoft. *Worm:Win32/Conficker.B*. https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Worm%3aWin32%2fConficker.B. Last Accessed: 2015-12-22.

[MySa] MySQL. *ACID Properties*. https://dev.mysql.com/doc/refman/5.6/en/glossary.html#glos_acid. Last Accessed: 2016-01-24.

[MySb] MySQL. *MySQL*. https://www.mysql.com/. Last Accessed: 2016-01-03.

[Nic] Nicolas Falliere (Symantec). *Sality: Story of a Peer-to-Peer Viral Network*. http://www.symantec.com/connect/sites/default/files/sality_peer_to_peer_viral_network.pdf. Last Accessed: 2015-12-22.

[Pir+15] R. S. Pirscoveanu, S. S. Hansen, T. M. Larsen, M. Stevanovic, J. M. Pedersen, and A. Czech. "Analysis of Malware behavior: Type classification using machine learning." In: *Cyber Situational Awareness, Data Analytics and Assessment (CyberSA), 2015 International Conference on*. IEEE. 2015, pp. 1–7.

[Proa] Process Library. *ntdll.dll*. `http://www.processlibrary.com/en/directory/files/ntdll/23004/`. Last Accessed: 2016-02-05.

[Prob] Process Library. *rpcrt4.dll*. `http://www.processlibrary.com/en/directory/files/rpcrt4/23580/`. Last Accessed: 2016-02-05.

[Pyt] Python. *Python Pickle*. `https://docs.python.org/2/library/pickle.html`. Last Accessed: 2016-01-03.

[Rie+09] K. Rieck, P. Trinius, C. Willems, and T. Holz. *Automatic analysis of malware behavior using machine learning*. TU, Professoren der Fak. IV, 2009.

[RMI11] B. B. Rad, M. Masrom, and S. Ibrahim. "Evolution of computer virus concealment and anti-virus techniques: a short survey." In: *arXiv preprint arXiv:1104.1070* (2011).

[RMI12] B. B. Rad, M. Masrom, and S. Ibrahim. "Camouflage in malware: from encryption to metamorphism." In: *International Journal of Computer Science and Network Security* 12.8 (2012), pp. 74–83.

[ŘS10] R. Řehůřek and P. Sojka. "Software Framework for Topic Modelling with Large Corpora." English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. `http://is.muni.cz/publication/884893/en`. Valletta, Malta: ELRA, May 22, 2010, pp. 45–50.

[SF01] P. Ször and P. Ferrie. "Hunting for metamorphic." In: 2001.

[SS14] A. Sharma and S. Sahay. "Evolution and Detection of Polymorphic and Metamorphic Malwares: A Survey." In: *arXiv preprint arXiv:1406.7061* (2014).

[Sym] Symantec. *Trojan.Zbot (Zeus)*. `http://www.symantec.com/security_response/writeup.jsp?docid=2010-011016-3514-99`. Last Accessed: 2015-12-22.

[Tia+10] R. Tian, R. Islam, L. Batten, and S. Versteeg. "Differentiating malware from cleanware using behavioural analysis." In: *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*. IEEE. 2010, pp. 23–30.

[Vir] VirusTotal. *VirusTotal - free service for identification of malware detected by antivirus engines and website scanners*. `https://www.virustotal.com/en/about/`. Last Accessed: 2016-01-03.

[Wika] Wikipedia. *Cluster Analysis*. https://en.wikipedia.org/wiki/Cluster_analysis. Last Accessed: 2016-01-21.

[Wikb] Wikipedia. *Device Driver*. https://en.wikipedia.org/wiki/Device_driver. Last Accessed: 2016-01-03.

[Wikc] Wikipedia. *Statistical classification*. https://en.wikipedia.org/wiki/Statistical_classification. Last Accessed: 2016-01-21.

[Wikd] Wikipedia. *Sven Jaschan*. https://en.wikipedia.org/wiki/Sven_Jaschan. Last Accessed: 2015-12-22.

[Wike] Wikipedia. *tf-idf*. https://en.wikipedia.org/wiki/Tf%E2%80%93idf. Last Accessed: 2016-01-21.

[Yav+12] C. Yavvari, A. Tokhtabayev, H. Rangwala, and A. Stavrou. "Malware characterization using behavioral components." In: *Computer Network Security*. Springer, 2012, pp. 226–239.