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 Verhaletnsinterferenz 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

Abstract

Rewrite at the last.

Contents

1 Introduction

Malware, short for malicious software, is a generic term referring to all kinds of software that poses threat of causing damage, disrupt, steal or some other illegitimate action, on computer, server, or computer network. Malware can infect the system by being bundled with some other program or being attached as macros in the file. When the user runs or opens such program or file, the malware gets installed into the user's machine. They can also install themselves by exploiting some common vulnerabilities in Operating System (OS), network devices, or other common software. The vast majority is installed by some action from user, such as clicking an e-mail attachment or downloading a file from Internet [**ciscodif**].
Some of the commonly known malware types are viruses, worms, Trojans, and bots.
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, file sharing or as an email attachment.
Worm, unlike virus, is capable to run independently and is self-propagating. It propagates by exploiting some vulnerability in machine OS, device drivers or taking the advantage of file-transfer features, such as email or network share.
Trojan are non-replicating malware which gets its name from ancient Greek story *Trojan Horse*. It is a software that looks legitimate, but is harmful in disguise, which usually spreads through user interaction such as Internet downloads and email attachments.
Bot, is a word derived from *"robot"*, which is capable of performing automated tasks. The host machine infected by a bot, can 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 bot is called **Botnet**. A bot is capable of logging keystroke, gather passwords, and financial information, and send it to C&C server. Bot can also be used to launch *Distributed Denial of Service (DDoS)* attack, by flooding single server, with many requests or sending spam in large scale.
Some malware are able to change its structure and create a new variant of itself each time it infects new victim. On the basis of how the malware changes its structure, it can be broadly divided into Polymorphic and Metamorphic malware. **Polymorphic**

malware divides its program into two section of code, "decryptor" and "encryptor". The first code section, "decryptor", decrypts the second section of the code and hands the execution control to the decrypted section of code. When the second section of code (encryptor) executes, it creates a new "decryptor" to encrypt itself, and links the new encrypted code section (encryptor) and new "decryptor" to construct a new variant of 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 new infected victim [**rad2011evolution**]. **Metamorphic** malware are body-polymorphic malware, i.e., 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 [**rad2012camouflage**]. In both cases the new variants may have different syntactic properties, but the real behavior of the malware remains same.

As the Internet increased its penetration over the last two decade, more and more devices are being connected via Internet. Many of our daily life activities are now depended on its usage, such as email, banking, bill payment, and social network. Along with this, the underground Internet economy has also been on the rise. Malware authors now do not write malware just for fun, to create annoyance or break into some system as bragging rights, but also for profit generation. Malware authors look for banking credentials, Credit cards, personal informations and identifications 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 [**moore2009economics**]. 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 [**tian**]. According to *AV-Test*, an independent IT-Security Institute based on Germany, about 140 million new malware were introduced in year 2015 alone, making the total number of malware recorded so far to almost half a billion [**avtest**].
With the increase in numbers of malware as it has become a lucrative business, malware authors are vying with each other to take sole control of their victim for larger profit. Many cases have been found in the past, where one malware try 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. In 2015, *Shifu*, a banking Trojan, was found to have behavior similar to Anti Virus (AV), i.e., preventing the host from further infection by other malware, once it had taken control over the victim's machine. We have discussed in details, about such anecdotal evidence of

interaction between malware families, in section 2.2.
The study of aforementioned behavioral interference between the malware family will provide a novel knowledge to understand dynamic aspect of modern malware and the inter family relations. This behavior is also a case for environment-sensitive malware, where malware change their behavior depending on different factors of its running environment, such as presence or absence of files, programs, or running services. To the best of our knowledge, there hasn't been study of such interference between two malware families so far. We present a systematic way to find out such interferences from large number of malware samples.
Our dataset consists of 22,154,180 malware samples, collected by *Anubis* [**anubis**], a dynamic full-system-emulation-based malware analysis environment, over time 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 good approach, whereas running all the possible pairs of millions of malware is not scalable. We needed to filter our data set without losing possible candidates exhibiting behavioral interference. In order to minimize our search space for candidate selection, and maximize the probability for finding 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. **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 Windows OS, **system calls** refers to Windows Application Programming Interface (API) functions. The trace of all the resource modified or queried by a malware during its execution is referred as **Behavioral Profile** of that malware. In section 3.2 and section 3.3 we define behavioral profiles and different resource types in details. If a malware makes a failed attempt to access or delete resource 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 registry 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 candidate based on common resource gave us concrete candidate pairs. The candidate pairs were still large to run them in scalable manner and find substantial result. In order to further study the dataset, we used clustering of malware (based on behavioral profile), on those filtered candidates (7,362,635 in numbers). **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 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 profile has been used extensively in previous

research works, which we have listed in details in section 2.4. We treated each cluster as a family, and malware falling in those cluster as variant of that family. Candidate pairs were chosen in such a way that two malware representing a pair, did not belong to same family. The algorithm for candidate selection based on clustering result and common resource heuristic is described in section 3.4. An *unpacker* was designed in order to run a candidate pair inside the *Anubis* system for analysis. **Unpacker** drops the candidate pair in the analysis system, and executes both malware with a set time interval in between two execution [see section 4.2]. We were able to successfully find X pair of malware with interfering behavior with a true positive rate of X.

Our research will provide following contributions:

- To the best of our knowledge, we are the first to perform a systematic study of interferences between malware families. This novel research will help better understand dynamic aspect of malware behavior.
- Our work is based on wide variety of large malware dataset collected over time in wild.
- A novel approach to malware clustering based on their behavior profiles. We show our clustering approach is pretty good in grouping similar behaving malware in same cluster.
- An automated system that detects interfering malware samples in large scale. Our system was able to find malware pairs with interfering behavior with good accuracy.

We discuss about malware families, evidences of interaction between malware families, 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. Following the motivation, we give an overview, design rationale, and technical description of our work flow and system in chapter 3. In chapter 4, we explain how we implemented the system, problems faced during the implementation, and how we solved them. We present the findings 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 on anecdotal evidences and previous research works and show the necessity of our study to add a new aspect on understanding malware behavior. We start the chapter with brief summary of some popular malware families in section 2.1 and different incident of interferences between malware families in the past in section 2.2. We discuss about malware analysis techniques and *Anubis* system in section 2.3. We then discuss why AV vendor are not reliable in case of malware family determination, and how clustering has been used so far in different research work to cluster similar behaving malware in **??**.
We end the chapter with summary of existing knowledge and motivation for new research.

2.1 Malware Families

We discussed about Malware and its types in chapter 1. 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 on their behaviors, capabilities, and mode of operation. Even between these three families we can see stark difference in their behavior and approach of infecting victim's machine permanently keeping their presence low. Also each malware families created or modified 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 start of 2009 [**conficker**]. It is capable of infecting and spreading across network without any human interaction. If the user of the compromised machine does not has admin privilege, it is capable of trying different common weak passwords (password such as *'test123'* or *'password'*) in order to gain admin privilege of network share directory, and drop a copy of itself.
The worm first tries to copy itself in 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 the Window 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 underground criminal market, and distributes itself by spam,phising and drive-by downloads [**zeus**].
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 [**zeusmicro**].

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 *Microsfot 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.A
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 [**salitysym**]
Sality targets all files in system drive (usually *'C'* drive in Windows) and tries to delete files related to anti-virus. It stops any security related process (anti-virus) and changes

Windows registry keys related to av software in order to lower the computer security. One of the symptoms to find out if a machine is infected by *Sality* family malware is the presence of files, listed in Listing 2.3, in the system [**salitymicro**].

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 about malware families and their supposed behavior in previous section. In this section we show some of the evidences of interference between different malware families, recorded early in year 2004 to latest in year 2015. We look upon those incidents to know the families involved, nature of interference, and reason behind it. One of our main hypothesis is finding such behavioral interference between the malware families in large scale. This anecdotal evidences 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 exchange of words between the creators of *Bagle, Netsky* and *Mydoom*. All three malware family were computer worms spreading through email as an attachment and making victim curious enough to download and open it. 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 [**bbccover**]. 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 infection with *Mydoom* and *Bagle* worms from victim's computer [**wikinetsky**].

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 the financial institutions, especially targeting U.S. It infected roughly 1.4 million computers, mainly located in U.S, and got the personal identification informations and financial informations of victim in order to transfer money out of victim bank accounts [**fbispyeye**]
SpyEye came with the feature called *Kill Zeus* that was successful in removing large varieties of *Zeus* family. The battle between these two families were pretty much dynamic. Many releases of the Trojan toolkit were made from both the family in order to negate each others dominance [**sanszeus**].

2.2.3 Shifu

Shifu, a highly evasive malware family, targeting mainly Japanese (82%), Austria-Germany (12%), and rest of Europe (6%) banks for sensitive data, was detected recently in the year 2015 [**secintelshifu**]. 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 the presence of files such as *pos.exe, vmmouse.sys, sanboxstarter.exe* in the system [**mccafeshifu**]. In order to check if it is being debugged, it calls *IsDebuggerPresent* Windows API which detects if program is being debugged [**mccafeshifu**].
Shifu also had antivirus kind of feature that prevented any other malware from infecting its victim. It kept the 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 stopped those suspicious files from being installed in the system and also sent a copy of the file to its C&C server, probably for further analysis [**secintelshifu**]. If there were already presence of other malware, it would stop any updates to those other malware, by disconnecting them from their botmaster [**secintelshifu**].

2.3 Malware Analysis

In previous sections, we talked about malware families and interference between them; in this section we describe techniques of malware analysis and *Anbuis*, the analysis system, we chose for our research work.
Malware analysis is a process of dissecting different components of a malware 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 usually 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 of the way to

perform static analysis is by disassembling the binary with use of disassembler, decompilers, source code analyzers 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 [**sharma2014**]. *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 *Virtual Box* or *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 [**sharma2014**]. The main drawback of dynamic analysis is that only single execution path is examined. **chipounov2012s2e** proposed an approach to multi-path execution by triggering and exploring different execution path during run-time [**chipounov2012s2e**]. Also, many evasive malware can evade detection by *dynamic analysis* by identifying the presence of analysis environment, and refraining from performing malicious activities [**barecloud**].

2.3.1 Anubis

For our experiment we use **Anbuis**, a dynamic malware analysis platform for analyzing Windows PE-executatbles [**anubis**]. We needed a dynamic analysis system because we were working on large scale malware dataset to find interference between malware families. With dynamic analysis system, we could run the possible candidate malware pair and look for the interference between them in real action to give proof to our findings. It would be tedious work to perform the same task with static analysis for large number of malware.
Anubis generates a detailed report about modifications made to Windows registry or the file system, about interaction with the Windows Service Manager or other processes and logs of all generated network traffic [**anubis**].anubis 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 report of those malware.
- It is well accepted analysis platform among the security researchers.

2.4 Malware Clustering

In order to find the malware pair that belong to different families and analyze them for behavioral interference in *Anubis*, we need a basis to cluster the malware into different families. We believe that the candidate malware pair belonging to two different families will have high probability of interference in terms of negating each other.
Anti-virus companies do provide labels to categorize malware into different families (such as listed in report of *VirusTotal* [**virustotal**]), but they are not reliable. They have many shortcomings and limitations. According to **bailey** they do not fulfill the criteria consistency, completeness, and conciseness [**bailey**].

- **Consistency.** Different AV vendors place the malware into different categories, and these categories also do not hold 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. Even for month old binaries could be still unlabeled, which makes the AV system labeling incomplete.
- **Conciseness.** The label provided by AV vendor are either too little or too much information with not much substantial meaning. They usually gave general idea of broad term as "trojan" and "worm" and not concise description to the specific malware or its family.

With the advent of dynamic analysis of malware, there has also been rise in different approach to build an automated system that could analyze the malware in large numbers. New malware are on the rise at an exponential rate with about 400,000 of them being introduced each day in the year 2015 [**avtest**]. Malware authors use polymorphic and metamorphic techniques to make malware more complex and change it's form so as to make traditional signature based detection hard to detect them. As malware continue to evolve and rise in numbers, researchers are now looking into 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. **Clustring**, on other hand is an instance of unsupervised learning, i.e., grouping unlabeled data object which are similar in some sense in same cluster. A good clustering system would be able to study new binary to find if the new binary is a variant of previous known malware and classify them accordingly.
As we described earlier that AV label on malware are not much reliable, we look onto different clustering techniques to categorize our dataset into different probable families.

Let us discuss about works done so far on clustering and classification of malware to know about current state of art.
pirscoveanu used *Random Forest* classifier on the behavior data set of *42,068* malware to achieve high classification rate with weighted average Area Under Curve (AUC) value of 0.98 [**pirscoveanu**]. **mosko** used *Naive Bayes* classifier based on 20 malware-behavior to detect malware with mean detection accuracy of over 90% [**mosko**]. **yavvari** used behavioral mapping approach to cluster *1,727* unique malware pair samples [**yavvari**]. **bayer** used behavior clustering to cluster 75 thousand sample claiming to achieve better precision than the previous approaches [**bayer**]. **firdausi** in their paper did the measurement of five different classifiers, *k-Nearest Neighbor, Naive Bayer, J48 decision trees, Support Vector Machine, and Multilayer Perception Neural Network*, to classify the malware based on behavioral data. With their result, they state that machine learning techniques based on behavioral profile could detect malware quite efficiently and effectively [**firdausi**].
Based on the previous research work and its result, we believe that machine learning techniques are good approach for malware clustering to cluster malware into different families. We studied the work of **bayer** in detail to see if we could use their approach of behavioral clustering as their work were the only one to cluster malware in large scale, i.e. 75 thousands malware under three hours [**bayer**]. With further study of the [**bayer**] paper, we found that their approach still did not look like scalable to millions of samples we need 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. Instead we preferred using something ready to use (not to spend time improving something that would not be considered any novelty for our work).

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* is statistically measured weight to evaluate the importance of a word in document with respect to whole corpus) approaches have a large memory footprint, $O(\#docs \times \#words)$, we switched to clustering algorithms whose memory footprint does not depend on the number of documents, and decided to use *latent Dirichlet allocation* (**LDA**).

2.4.1 Latend Dirichlet Allocation

latent Dirichlet allocation (**LDA**), is a generative probabilistic model for collections of discrete data such as text corpora developed by **Blei** [**Blei**]. **LDA**, equivalent to dimension-reduction algorithms for high-dimensional clustering, is one such algorithm which does not depend on 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.
We will show the implementation and result of LDA, using python library *Gensim* for clustering malware in section 4.5.

2.4.2 Gensim

"Gensim" (**generate similar**) [**gensim**] is an efficient python library developed by **gensim** We used *"Gensim"*to perform the LDA for clustering malware into families. We preferred using it because of its simplicity, well documented API, and ability to work on large corpus.
We use the *ldamulticore*[**ldamulticore**] models of the Gensim API.
Some of the benefits of using the *ldamulticore* model were, the model utilized the multi cores processor of the machine efficiently with parallelization making the clustering process faster. The training algorithm is streamed and we could feed the input documents sequentially even for large data. The training algorithm runs in constant memory with respect to number of documents. This made possible for us to process corpora that was larger than the memory size of our machine (32 Gigabytes), as size of training corpus did not affect the memory footprint. The maximum size of our corpus was 81 Gigabytes.

2.5 Summary and Motivation

Motivated by the economic profit, there has been increase in number of new malware, and fighting between the malware families to control the larger share of the underground economy. To get the larger pie of the economy and show their superiority, malware family try to negate the existence and influence of another family. We showed some evidences 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 evolving dynamic behavior of malware. Our research provides a systematic way to find such behavioral

interference of malware families on a large scale data.
We gave an overview on two common malware analysis system and reason for choosing *Anbuis*, 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 family have different techniques to inject themselves keeping them alive even after the system reboot. As described in section 2.1, they copy themselves in certain system path, 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 pattern related to each families could be used to check the presence of particular family. We using this rationale in our heuristics to filter probable malware candidate with behavioral interference. To select the sample candidate pair from different family, we look for those malware that tried to access or delete certain resource which was created by malware from another family. The detailed approach of selection of pair is described in section 3.4.
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 about use of machine learning techniques in the previous research work for clustering and classification of malware based on their behavioral profiles. The results of the work so for on classification and clustering of malware were pretty good in order to improve efficiency and detection of the malware analysis system. However, the clustering of malware performed so far are on a small number of sample, consisting of tens of thousands of malware samples, and were 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 is of size 22,154,180 malware samples. The study of behavioral interference between the malware families in this large scale is a new research work in the field of malware study. Our work will be a cornerstone in that topic. It will help to understand dynamic behavior of environment sensitive malware.

3 Methodology

In this chapter we outlay how we planned and performed our research. We will look upon the overall procedure and also discuss on why we chose certain approach. Let us start with the overview of our experiment.

3.1 Overview

Figure 3.1 shows the overall structure of the procedure we followed to perform our research. We started with old database which had all the analysis result saved for the samples submitted to Anubis. We made a reverse index from the database such that every resource name had the malware id associated with the resource through some resource activity. Initially, we took into account resource type *File, Registry, and Mutex*, and the resource activity *create, read, modify, and delete*.
After the reverse index was created, we mapped the create activity with read, modify, and delete activities. As the result of the mapping we had list of malware that created a *resource name*, and a list of malware that either read/modify/delete the same resource name. We used some heuristics such as resource name created by exactly one single malware and delete by another single malware, to get the candidate pair and run the candidate pair in Anubis system. We analyzed the result and found some interesting case of dropper malware, self reading malware, and most importantly a logical flaw in our current approach.
The flaw in our current approach was we did not had failed attempt of malware activity such as read and delete logged in our database. This was because every sample binary submitted to anubis system for analysis were run in complete isolation. Hence any other malware binary which would try to access the resource created by malware other than itself would result into failed access or delete. We needed this failed operation we needed to create the new database. We used behavioral profile described in 3.2to recreate the database.
We also used document clustering algorithm, discussed in **??** to cluster the malware based on their dynamic behavioral activities into different topic families. We used approach as discussed in section 3.4 to find the probable candidate pairs. After we had minimal number of candidate pairs covering all the family topic and interesting

resources, we run those pairs in the Anubis system and analyzed the result to find battle activities between the two candidates.

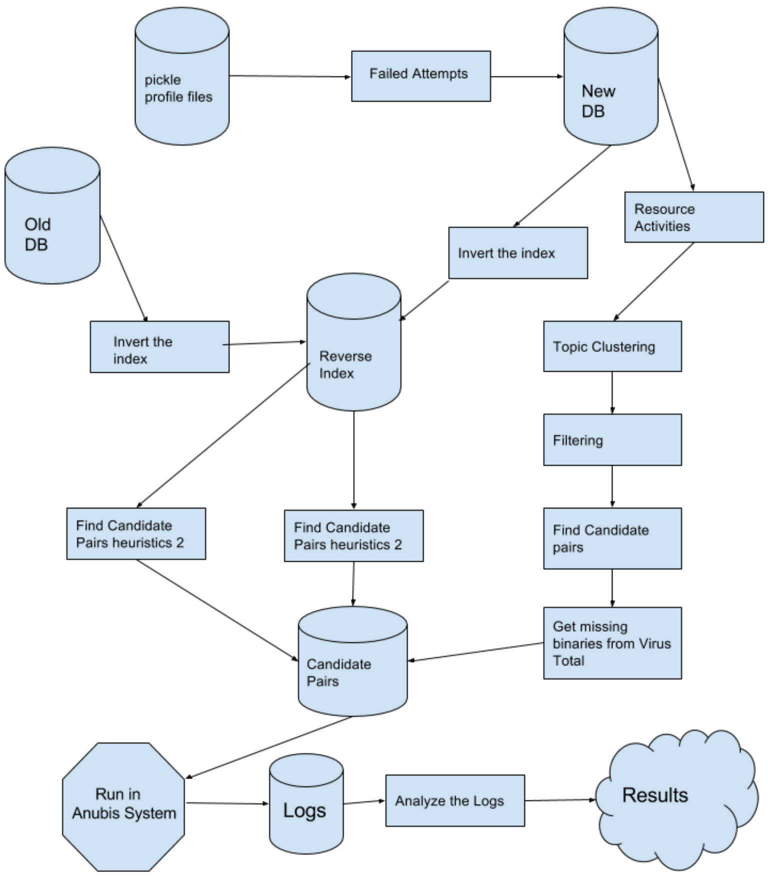

Figure 3.1: Overview of the research and experiment

3.2 Behavioral Profile

Previously, **bayer** used Anubis system for the dynamic analysis of malware to gets it execution traces [**bayer**]. According to the authors, **bayer** "**behavioral profile**", is defined 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. As 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.
They created a behavioral profile based on the execution traces of programs irrespective of order execution. 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. We had these behavioral profiles saved as python pickle [**pythonpickle**] file in our system and we used this to recreate new database that had the record of failed attempts of delete or read.
Our primarily concern was list of *OS Objects* and *Operations* associated with the behavioral profile of the malware. We needed those to recreate the database with the system calls as malware activities Lets describe these two in a brief.

3.2.1 OS Objects

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

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

3.2.2 OS Operations

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

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

The sample of behavioral profile is shown in Listing 4.3 in chapter 4 where we also describe how we used it for generating resource activities and text corpora for clustering.

3.3 Resource Types

As discussed before in subsection 3.2.1, an OS Object, are representation of resource types. For our research, we took into consideration following 8 resource types among those list. We will give a short description of each resource type according to Microsoft

Developers Network documentation. All the definition are taken into account from Microsoft Developers Network Official documentation website.[**msdn**].

3.3.1 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* [**msfile**].
File type in the behavioral profile encompasses not only general file, but named pipe and mailslot resources. File is an important resource type for us to focus as our hypothesis for research is that malware of certain family creates or deletes a certain file to infect a system and this also could be used by malware of another family to remove its nemesis from system.

3.3.2 Registry

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 [**msregistry**].
We take the registry keys associated with the malware into consideration for experiment as it provides vital information on the behavior of a malware. Malware with same family might have similar registry key activity and also malware from different family might look for the particular registry key in the system in order to detect the presence of its anti family.

3.3.3 Service

A user can start a *service* automatically 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 Manater (SCM) [**msservice**].

3.3.4 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 [**mssection**].
In case of behavioral profile, it broadly represents memory mapped files.

3.3.5 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. One or more threads can be run in the context of the process [**msprocess**]

3.3.6 Job

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 [**msjob**].

3.3.7 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 [**mssync**].
The sync object type covers all the synchronization activities. [**mssync**].

3.3.8 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 [**devicedriver**]
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 [**msdevice**].
We capture those loading and unloading of Windows Device Driver recorded in the behavioral profile.

3.4 Running Experiment

- Say, R is 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. Combine 'A' and 'B' and cluster them to cluster ids $[c_1, c_2, \ldots. c_n]$ (n is family/topic count) such that any malware sample x in (A union B) can be tagged/mapped to cluster id $C(x)$, where $C(x)$ belongs to $[c_1, c_2, \ldots c_n]$.
- Now, for each "r" in R, we generate a set of candidate pairs p_r for experiment. p_r is a set of malware pairs (x_r, y_r) such that x_r belongs to A_r and y_r belongs to B_r and $C(x_r)$ not equal to $C(y_r)$, not belonging to same cluster.
- We generate such (x_r, y_r) pairs for all possible cluster pairs $(C(x_r), C(y_r))$ corresponding to a resource "r".
- The final experiment set E is a set of such (x_r, y_r) for all resources r in R.
- Finally for any resource "r", if the size of the set $|C(x) : x \in A_r| > 10 or |C(x) : x \in B_r| > 10$, we discard r and its corresponding experiment pairs from E, since the resource created/read by too many families is less interesting.
- Here (x, y) and (y, x) will be different experiments because we run one sample, wait, and run another sample. Result can be different based on which one runs first. In rare case, both samples may be trying to detect each other.

4 Implementation

In this chapter we give a detail explanation of how we implemented and conducted our research as described in chapter 3

4.1 Study of current Database

Our main resource for the research was direct access the millions of malware samples from Anubis collected over the years. Not only was this resource our main strength but it was also a challenge to effectively and efficiently analyze those data for our research work.
We started with studying the Anubis system and current database. We primarily dealt with two database of Anubis backend, the **db_report** and **web_analysis**. The web analysis was the first entry of any sample malware submitted for analysis. It consisted of tables such as result, file, file_task. Each sample was given a unique file_id along with md5 and sha hashes. A new file_task *id* was created and the analysis of the sample would be done for different behavioral activities related to resources such as File, Registry, Mutex activities.
These activities were saved in db_report database table. The resource activities of the malware in **db_report** database was associated with the **web_analysis** database by the constraint key *result_id* of *result* table in web_analysis database.

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

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

We looked upon 3 resource type for the beginning. The resources type were *File, Registry, Mutex*. For each resource type we took into account their *create, read, delete* activities.
The total number of malware samples that we had in our test database were **22,154,180**. Our first step was to create a reverse index from the database so that we could get the

list of malware that is associated with the resource activities. We started with writing a python script to do the task and save it in the file. However we found the first hurdle of our project. Since the number of malware were too large trying to save all their activities in data structure would cause our machine to run out of memory and hence crash the system. To solve this we took the approach of map reduce.
We ran our script in batch 50K malware at a time and saved the reverse index of activity to the file with numbering. Around 420 files were created for each resource type. The result files were in the format where resource name and list of result ids were separated by commas (,) as delimiter. This was one of the time consuming operation and it took almost 5 days for the script to finish. After the reverse index was generated in multiple numbered output, we sorted the file based on resource name, alphabetically, and then joined the different parts with regard to the resource activity as key to merged it into one single file.
Since, this was a merge sort, we considered two sorted files at a time until single file was left, the reduce part was pretty fast ($O(n * \log(n))$).

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

The snippet of resultant output:

```
Application Data,87623151,87079727,87034095
AutoHotkey,87623151,87079727,87034095
AutoScriptWriter,87623151,87079727,87034095
B:\mbr.exe,121858971
BIN,177509111,103858187
Base Images,189524063,184501719,87504631,86763863
Buttons,111448211
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/Adobe Reader
    8,178046895,174206059,183601891,89650247
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/_MEI1192/,161552035,116241803
```

4.2 Packer & Unpacker

In order to run the pair of malware together inside the Anubis environment we made a Win32 console application, named Unpacker, as Anubis VM was based on Windows (XP). We used the fact that we can append any data to *exe*, and it would work fine. So we created a meta binary, that will read itself, and extract the other two binaries, that will be attached to it. It will then create it and spawn two independent process with certain time delay. We called this Unpacker binary.

We also wrote a packer program that would add the candidate binary pair at the end of Unpacker binary and then further append the time delay and file size of each candidate binary as meta information.
The unpacker binary when executed would read itself to fetch the meta information and then the candidate binary bytes.
The structure of unpacker binary is shown in Figure 4.1.

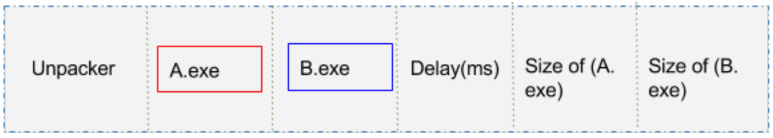

Figure 4.1: [Packer structure] Structure of the Packer binary that would create the candidate pair and run them with delay.

We used a struct of 3 integer size to read and hold the meta information. The size of last three *unsigned int* byte had the binary pair sizes information and delay, know as meta information. The offset would be deduction of $3 * size\ of\ uint$ from the total size of itself.
When the file size of packed binary was known, we used `fseek` function appropriately to start reading form the correct position until the exact size of binary were read. Then these bytes were saved to the new file pointer. Once the binary pairs were extracted by the unpacker new binary files were created in the Anubis environment and those were executed one after another with the delay of time as given in meta information. We used `windows.h` standard libraries *CreateProcess* and *Sleep* function for the purpose.

Listing 4.2: snippet of unpacker.c file

```
/* stuct for storing meta information */
typedef struct {
  unsigned int delay;
  unsigned int fsize1;
  unsigned int fsize2;
}meta_info;

/* reading the meta information and first binary */
rfp = fopen(argv[0], "rb");
wfp1 = fopen(fileName1, "wb");
```

```
fseek(rfp,0,SEEK_END);
size = ftell(rfp);
offset = size - sizeof(meta_info);
fseek(rfp, offset, SEEK_SET);
fread(&info, 1, sizeof(info), rfp);

/* calculate the unpackersize from the offset and files size. */
unpackersize = offset - (info.fsize1 + info.fsize2);

/* rewind back and to the point of the start of file1 */
fseek(rfp,0,SEEK_SET);
fseek(rfp, unpackersize, SEEK_SET);

nread_sofar = 0;
while (nread_sofar < info.fsize1) {
   nread = fread(buf, 1, min(info.fsize1 - nread_sofar, sizeof(buf)), rfp
       );
   nread_sofar += nread;
   fwrite(buf, 1, nread, wfp1);
}
fclose(wfp1);
```

4.3 Initial Experiment

From the reverse index of the resource activities of the Malware, we created a mapping between the created activities against the deleted or read activities. We created a list of malware based on the common resource name as the key, where a list of malware which created the resource and another list of malware which delete/read the same resource. We started looking for one to one interaction of malware to a single resource. Any resource type that was created by exactly single malware and deleted by exactly another single malware. We looked for a set (a, b), where malware 'a' creates the resource r, and malware 'b' deletes the same resource, r, and no other malware has create or delete activities on that resource r.
After finding such pairs, when we ran those malware pairs in the Anubis system using our unpacker.
In our initial run, we found lots of dropper malware causing this interaction. 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 with different level of interaction and analyzing the results manually, not only were we able to understand the Anubis report in depth but also found an 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 since Anubis ran each submitted binary in an isolated environment, there was no way that a malware '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 unsuccessfully tried to access or delete a resource created by another malware, but the current database that we had did not had record of such failed attempt, but only recorded the successful operations. This lead us to look for the alternatives were we could find log of such failed attempt activities during the execution of binary.
We used the behavioral profiles as described in section 3.2 to find such failed activities of deletion and access. A snippet of behavioral profile in shown in Listing 4.3.

Listing 4.3: Behvaioral Profile sample

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

op | registry | 'HKLM\\SOFTWARE\\CLASSES\\CLSID\\{00021401-0000-0000-C000-000000000046}'
 open:1
 query:1
 query_value(''):1
 query_value('InprocServer32'):0
 query_value('ThreadingModel'):1

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

If we look at the listing of operation in Listing 4.3, we can see it records all the OS operation executed on a OS Object such as file, registry, section. If the operation was successful the result value is *1*, else it is *0*. We discussed the structure of *OS Object* and *OS Operations* in subsection 3.2.1 and subsection 3.2.2 respectively.
For an instance, *'create'* operation for file '`C:\ProgramFiles\CommonFiles\sumbh.exe`' was successful, where as *'query_value'* operation with argument value *'InprocServer32'* for registry '`HKLM\SOFTWARE\CLASSES\CLSID\{00021401-0000-0000-C000-000000000046}`' failed.

4.4 Creation of Database

Recreating the database was one of the bottleneck in our project and consumed much time. The profile files were needed to be accessed via network file system, walking through list of directories and file to find the profile pickle of malware. Not all of the profile pickle were found. We found *16,031,518* out of *22,154,180* malware.
We took only 8 of the resource type into account while parsing the profile files; those were File, Registry, Sync, Section, Process, Service, Job, and Driver. We went through the most common used system calls defined as OS Operation as described in section 3.2 and then mapped them into the broad three categories *Modify, Access, and Delete*. Windows Native and Windows API calls were already generalized during the creation of behavioral profile.

Listing 4.4: Mapping of generalize system calls with regard to behavioral profile

```
MODIFY_LIST = {
       file: ["create_named_pipe", "create_mailslot", "create", "rename",
            "lock", "set_information", "write", "unlock", "flush_buffer",
           "suspend", "map", "resume"],
       registry: ["create", "restore_key", "save_key", "map", "set_value",
            "set_information", "compress_key", "lock", "resume", "suspend"
           , "mem_write"],
       process: ["create", "set_information", "suspend", "resume", "unmap
           ", "map"],
       job: ["assign", "set_information"],
       driver: ["unload"],
       section: ["create", "map", "unmap", "mem_write", "extend", "
           suspend", "resume", "set_information", "release"],
       sync: ["create", "release", "map", "set_information", "mem_write"],

       service: ["create", "start", "control"]
       }

ACCESS_LIST = {
       file: ["query_file", "query", "open", "query_directory", "
           query_information", "read", "monitor_dir", "control", "
           device_control", "fs_control", "query_value"],
       registry: ["enumerate", "enumerate_value", "flush", "monitor_key",
            "open", "query", "query_value", "mem_read"],
       process: ["open", "query"],
        job: ["open", "query"],
        driver: ["load"],
        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"]
        }
```

We used **MySql Version 5.5.46** as our database engine.MySQL [**mysql**] is highly scalable and flexible relational database system which complies with the ACID model design principle. We followed the previous table structure to design our new database so that we could reuse parts of our previous program for data processing. The final database size was 1.2 Terabyte.
The basic CRUD operations were time expensive and made the overall operation slow. We also hit the integer overflow for the primary index of registry access activities table. The total registry access activities were more than 4 billion of records. To solve this, we had to create another registry access table and start inserting the registry access records in new table. When accessing the data for later use, we had to change our program so that we checked both the tables in order to find the registry access activities of a malware.
We had to spent much time then expected on database tuning to make the program run faster so that it would complete in reasonable time. Most part of the execution time was taken to read the profile gzip pickle and then normal sql CRUD operation as shown in Figure 4.2.
These were for 16,031,518 binary pickle profiles and their resource activities.
After the creation of new database, we made a reverse index from the new refined data for the *resource_name* to malware ids modifying, accessing, and deleting the resource. Getting all the resource activities related to all the resource types *file, registry, section,*

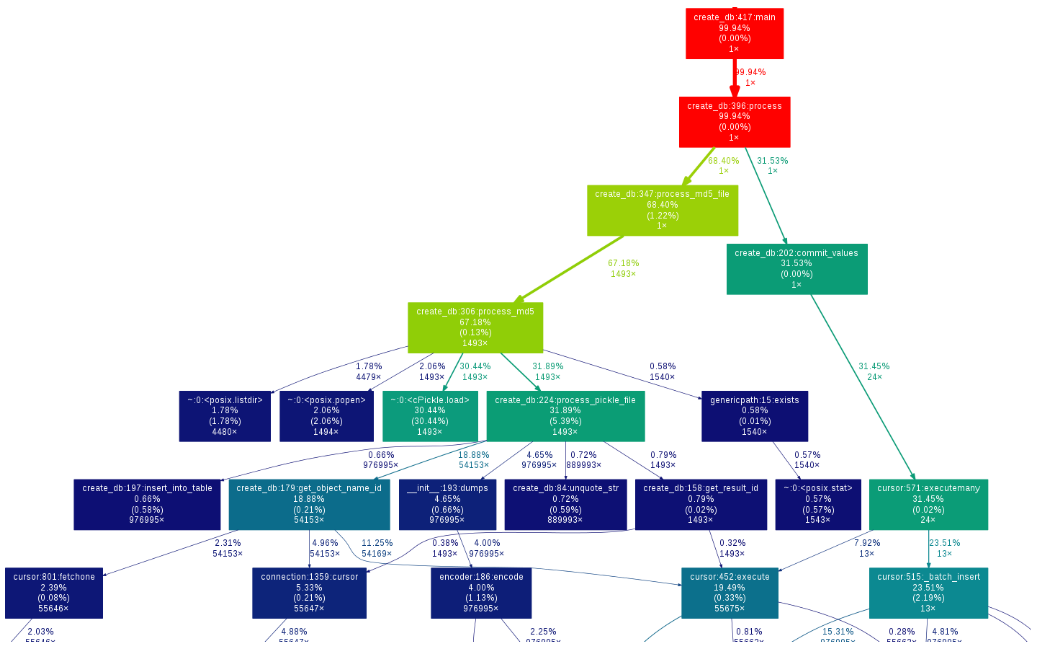
create_db:417:main 99.94% (0.00%) 1×
create_db:396:process 99.94% (0.00%) 1×
create_db:347:process_md5_file 68.40% (1.22%) 1×
create_db:202:commit_values 31.53% (0.00%) 1×
create_db:306:process_md5 67.18% (0.13%) 1493×
~:0:<posix.listdir> 1.78% (1.78%) 4480×
~:0:<posix.popen> 2.06% (2.06%) 1494×
~:0:<cPickle.load> 30.44% (30.44%) 1493×
create_db:224:process_pickle_file 31.89% (5.39%) 1493×
genericpath:15:exists 0.58% (0.01%) 1540×
create_db:197:insert_into_table 0.66% (0.58%) 976995×
create_db:179:get_object_name_id 18.88% (0.21%) 54153×
__init__:193:dumps 4.65% (0.66%) 976995×
create_db:84:unquote_str 0.72% (0.59%) 889993×
create_db:158:get_result_id 0.79% (0.02%) 1493×
~:0:<posix.stat> 0.57% (0.57%) 1543×
cursor:571:executemany 31.45% (0.02%) 24×
cursor:801:fetchone 2.39% (0.08%) 55646×
connection:1359:cursor 5.33% (0.21%) 55647×
encoder:186:encode 4.00% (1.13%) 976995×
cursor:452:execute 19.49% (0.33%) 55675×
cursor:515:_batch_insert 23.51% (2.19%) 13×

Figure 4.2: profiling of database creation program

driver, sync,service, process, and job was also time expensive. It took almost 5 days for us to get the reverse index of resource name to the malware ids with respective modify, access or delete operation. While creating the resource mapping this time, we took into account those activity that had successful creation with those activities with failed access and another mapping between successful creation and delete activity.
We made this mapping for every resource types. We made a mapping between successful file creation and failed file deletion and mapping between successful file creation and failed file access. Same mapping was done for resource types registry, sync, section,service, job, and driver. The resource name was always the common key for the mapping between the malware.
Once we had a mapping of resource activity with respect to resource name, we started to run the candidate pairs to test the behavioral interaction between those pairs. But, since the number of pairs were too many we hit the *n-combination* problem. A single resource `'r'` would have been created by more than thousands of malware and also same resource `'r'` would have been tried to access or delete with a failed attempt by another thousands of malware. If we made a pair out of every malware as possible candidate there would be too many candidates for us to run and analyze result.
We looked into clustering of the malware behavior to address this problem so that we

could cluster malware on different topic and only choose one sample from one topic among the list of topics.

4.5 Document Clustering

We researched on many types of clustering algorithm that we could use to cluster our malware. Our idea of clustering was based on malware resource activities. We had all the resource activities such as modification, access and deletion related to different resource types as discussed above. We represented each resource type and activities as numeric value. And each resource name would be represented by it's unique resource name id on our database system.

Listing 4.5: 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"}
```

So 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"*. Each such resource activities related to one single malware would be a bag of words representing a single document representing that particular malware.
We needed to convert all the resource activities related to modify, access and delete of the our test binaries into the corpora text. After 6 days of running the script, we could gather all the resource activity of 16,031,518 malware as the corpus for algorithm. The profiling of the program is shown in Figure 4.3. This was about 81 Gigabytes of data.
We used python module Gensim [**gensim**] for the topic clustering and used the LDA model as described in Methodology section. We filtered our corpora by discarding any resource activity that occurred in less than 1000 or more than 1 million for clustering. This was based on heuristic after studying the cdf graph of activities to number of malware. The number of topic was chosen to be 1000, however our malware samples were distributed among only 752 unique topic.
We calculated the distance between different clusters. We had quite a good results with small distance between the documents (malware) belonging to same cluster, intra topic distance, and larger distance between documents of different cluster, inter topic topic distance.
For the next clustering run, we wanted to decrease the number of malware (documents). To do this we tried the manual approach. We sorted the reverse index of resource name to malware with respect to the number of malware associated with the resource

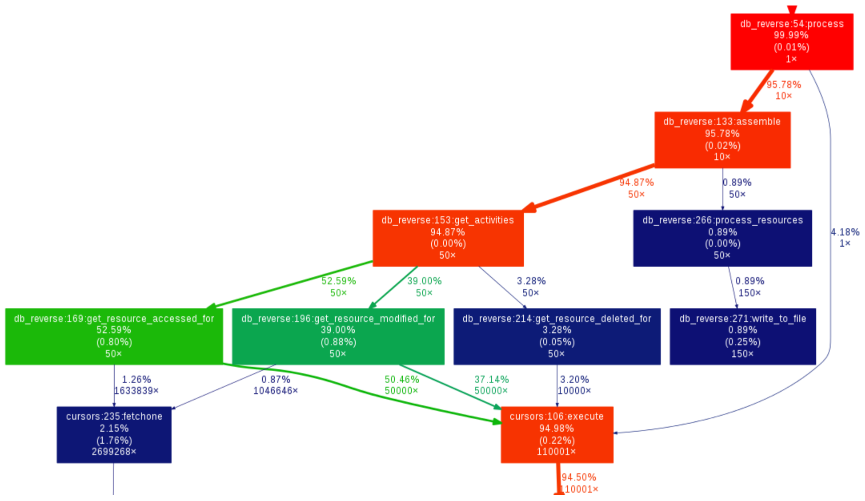

Figure 4.3: profiling for creation of resource activities of malware

Listing 4.6: Script to run Gensim LDA

```python
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(1000,0.6)
# 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)
```

name. Once we had it sorted in descending order, we got most commonly modified/accessed/created resource at the top of the list, and went through all the mapping line by line until we found some interesting resource name. Once we found that, we made a threshold to that resource name, and excluded any resource to malware mapping before that as it was not of substantial importance.
We did not had any resource activity mapping for resource type Service, Job, and Driver. Hence, we looked upon the remaining five resource type from our original resource types considered.
For instance, resource with name such as *Dr. Watson.exe* and *Sample.exe* were the one with highest count of malware associated with it. The former one is created whenever the program crashes and the later one is the name that anubis uses to run the sample binary.
We looked for such benign resource names and weeded out for all the resource activities related to resource type file, registry, section, sync, and process. We were able to lower the total number of malware from 16,031,518 to 7,362,635.
We created the bag of words again for those 7,362,635 malware, and started the document clustering LDA algorithm. This time we had even better results with respect to inter distance and intra distance of the clusters.
The malware were clustered into 662 clusters. 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 above graph we can see that distribution of cluster size was mostly below

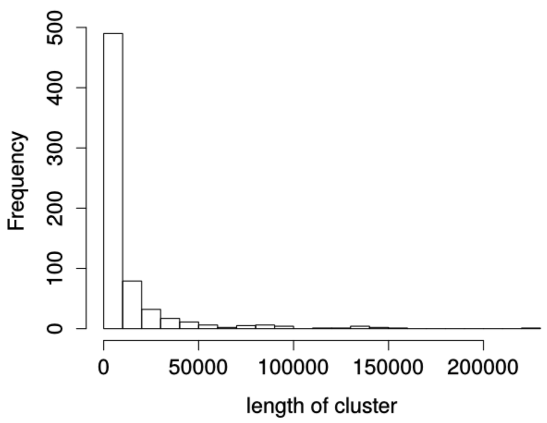

Figure 4.4: Histogram showing the distribution of Cluster size

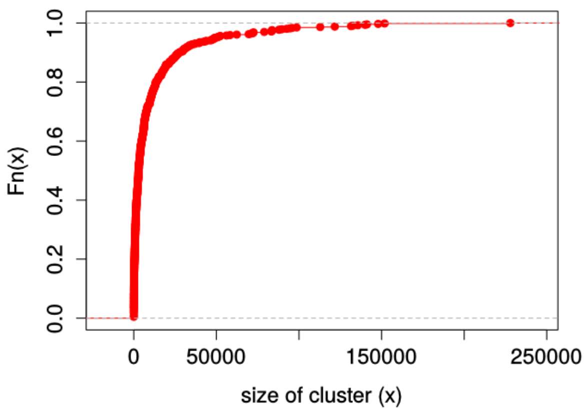

Figure 4.5: Graph showing cdf between size of cluster and topic fraction

50,000. 90% of the cluster topic, about *629*, falls under the cluster size of less than 50,000, where as about *4 million* malware, which is the 65% of total malware, are in the clsuter size of less than 50K.
To show the quality of our clustering we present 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 each topic family, we sampled at maximum 1000 malware, if the topic had that many number of malware. We calculated the average number of resource activities*(words)* present among those sampled mawlare*(document)* We then calculated the average common resources between the combinations of sample. We could see that the average common words between the intra malware sample were quite good with intra-distance of at least *500* covering 80% of family topics. For the inter-distance calculation, we took a random combination pair of topic *'x'* with other topics, and randomly sampled 1000 malware sample, if present, from those topic pair. We calculated the average common words between those the sampled malware for each pair to find the inter distance between the malware of different family topics. We can see that the inter-distance of only 10 was covering almost 90% of family topic.

4.6 Finding out the candidate pairs

After the clustering of the malware was done, we used the result and the our mapping of resource to malware to find the candidate pairs for experiment. In order to find the optimal set of candidate we interpreted the interaction of malware and resource with

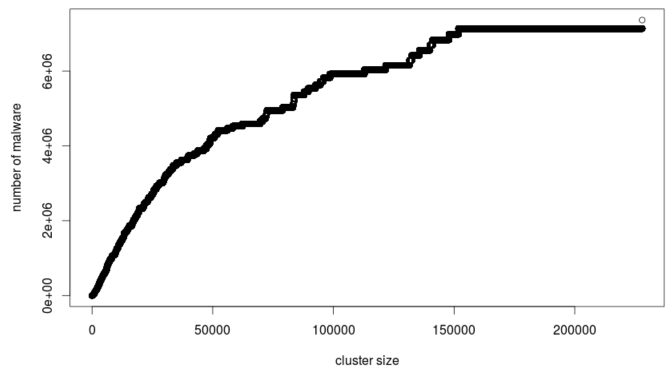

Figure 4.6: Graph showing cdf between size of cluster and total number of malware

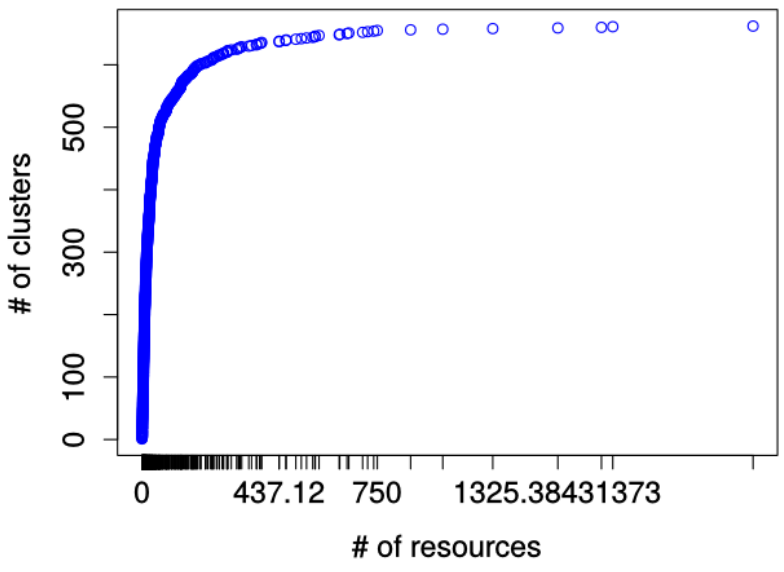

Figure 4.7: Graph showing cdf distribution of common resource between same family topic

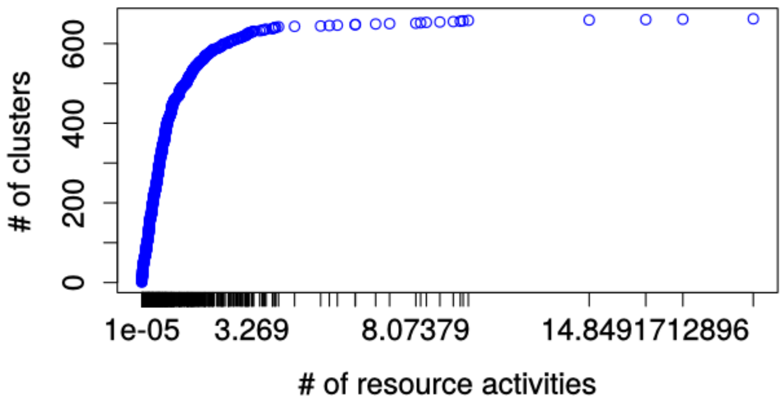

Figure 4.8: Graph showing cdf distribution of common resource between different family topic

respect to cluster into Flow Network.

4.6.1 Max Flow apporach

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 shown in Figure 4.9, to yield the optimal solution.
The network is made of 4 layers, other than sink and source. Layer one is reading malware, the 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 interesting resources 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 interesting resources that they could successfully modify. Layer four is the malware that successfully created/modified the resource.
The flow rules were:

- All malware from layer one is connected from source and also connected to the input cluster and resource combination it belongs to.
- All input cluster resource combination in layer two is connected to all output

cluster resource combination in layer 3 that has a matching resource.

- All output cluster resource 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.
- The maximum flow in this network is corresponding to an optimum match up.

Combinations and nodes that do not have a path from source to sink were not be put in the graph.

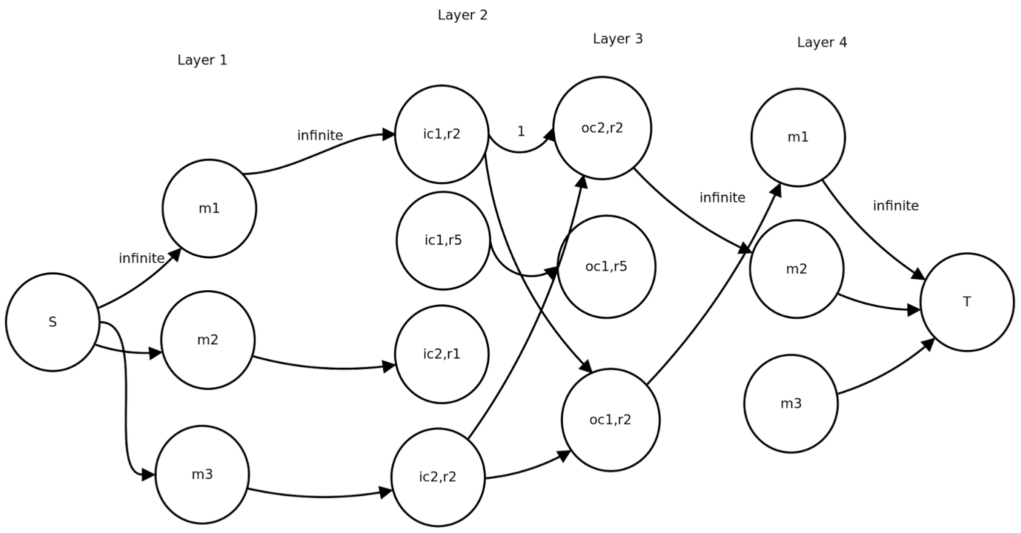

Figure 4.9: Graph representing the max flow implementation

We ran the max flow algorithm and found the optimal set of candidate pairs corresponding to every family topic, and interesting resource activity. However, the total number of candidate pairs to test were many for us to complete in time. We wanted to decrease this number in such a way that there were no repetition of malware pair if we already chose it regardless of different resource name. In order to achieve that, we tried Heuristic approach.

4.6.2 Heuristics Approach

The problem was to find the "minimal set" of md5 pairs that covers all unique 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 md5_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 output clusters from set *'B'*. Set *'A'*, *'B'* and *'R'* has usual meanings as described in the *section 3.4*.

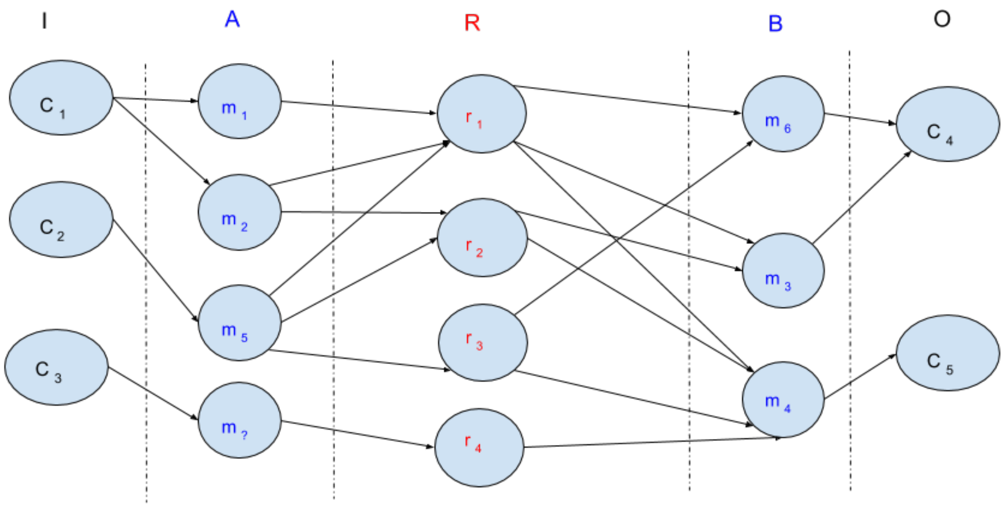

Figure 4.10: Graph showing the interconnection between Cluster topics, Malware, and Resources.

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.
The problem is 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 following 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 md5 pairs (all md5 pairs) and values are the list of corresponding resource ids.

Listing 4.8: Alogrithm to get minimal set of candidates for all resource

```
candidate_set = set()
for c_pair, v in db.iteritems():
    # reverse sort md5 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)
```

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 algorithm 4.8. It's a greedy algorithm starting from the md5 pair that corresponds most number of resource interactions. For each md5 pair it checks if that pair has been already chosen, and if not, adds the pair to the candidate list.

4.7 Running the Experiment

In total we had *263,701* candidate pairs to run the experiment. The breakdown of the number is given below. Modified has the usual meaning of successful operation, and access and delete are the failed attempt.

- process_modified_vs_process_deleted: 54
- file_modified_vs_file_deleted: 589
- section_modified_vs_section_accessed: 2786
- registry_modified_vs_registry_deleted: 4781

- sync_modified_vs_sync_accessed: 7791
- registry_modified_vs_registry_accessed: 35118
- file_modified_vs_file_accessed: 212582

4.7.1 Anubis Worker

We were provided with 7 instances of Anubis worker for conducting the experiment. For each candidate pairs we packed them together using the method as described in 4.2. Some of the missing binary from the candidate pairs were downloaded from *VirusTotal*. The Anubis worker were remotely accessed and operated. The time of execution for each worker was set to 6 minutes, when we ran the experiment such that malware 'a' of pair (a, b), run first for 3 minutes, and after that malware 'b' of the same pair is run in the same anubis worker. The time of execution were changed accordingly with respect to experiment. When we ran the malware 'a' first, then wait for some minutes for malware 'b' to run, and again wait for some minutes before malware 'a' was run again, the anubis worker run time was set to 9 minutes.
The result of the Anubis run were then copied to local machine to analyze further.

5 Results

6 Conclusion and Future Work

Glossary

computer is a machine that. . . .

Acronyms

ACID	Atomicity, Consistency, Integrity, and Durability.
API	Application Programming Interface.
C&C	Command and Control.
CRUD	Create, Read, Update, and Delete.
SQL	Structured Query Language.
TUM	Technische Universität München.

List of Figures

List of Tables