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 Verhaltnsinterferer 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

Malware of different families may not like each other. For example, different malware binaries might try to uninstall each other before infecting a system. This is an interesting case of 'environment-sensitive malware'. There are some anecdotal evidences (blogs). To the best of our knowledge, there is no prior research addressing this problem in a systematic way. In this research we systematically try explore the scene in the wild. We ran multiple malware samples from different families at the same time in the Anubis environment (a dynamic full-system-emulation-based malware analysis environment). We later analyzed the results of the emulation and detect such behaviors. We used clustering algorithm to cluster the malware based on its resources activities such as file,registry,section,syncs. With this apporach we could find 30 paris of malware that were indeed battling each other.

Contents

1 Introduction

Malware Dynamic Analysis and Static Analysis Anubis [Anu] Malware Family some blogs on battlefield our intension on finding one

2 Literature Review

To the best of our knowledge there has not been any research on finding the behavioral interaction between the malware families. There are some anecdotal evidence with some blogs but not any systematic research proving that the change in behavior of a malware on the presence of another do exist. However, there has been some work on clustering the malware with regard to its behavior. We will discuss in detail about it later.

2.1 Malware Families

We have already discussed about Malware and its types in previous Chapter. Let us look at some of the famous widely spread malware families.

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* [Micl, Worm:Win32/Conficker.B] the detection of *Confiker* worm increased by more that 225 percent since start of 2009. It is capable of infecting and spreading across network without any human interaction. It exploits weak password use by trying some common passwords,in order to drop a copy of itself in a network share directory, if current user does not has admin privilege. The worm first tries to copy itself in the Windows system folder as a hidden DLL using some random name. When unsuccessful it tries to copy 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>"
```

Not only that it can load itself as service whenever *netsvcs* group is loaded by system *svchost.exe*, it is also capable of loading itself as fake service under 'HKLM\SYSTEM\CurrentControlSet\Services'.

2.1.2 Zeus

Zeus is another malware of *trojan* type, that affects the *Microsoft Windows* Operating System, that 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. Attacker can used this to change the function of Trojan to change the target value. The Trojan is generated by a toolkit which is also available in underground criminal market, and distribute 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 it changes the registry as in Listing 2.2 [Mick, 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 *Microsfot Windows* system with extensions *.EXE* or *.SCR*. It spreads by infecting executatble files and replicating itself across network shares. Each infected host becomes the part of peer to peer network used to propagate the malware [Nic, Sality]
Sality targets all files in *C:*, and tries to delete files related to anti-virus, stops security related process, and steals sensitive information like cached password and logged keystrokes. It also changes Windows registry keys in order to lower the computer security. One of the symptoms to find out if a machine is infected by *Sality* family malware is presence of Listing 2.3files in the PC [Micj, Win32/Sality].

Listing 2.3: Files created by Sality in the infected machine

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

2.2 Battle between Malware Families

There has been some evidence of battle between the malware families. We will look upon some of those references in past.

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*. 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]. Messages like *"dont'r ruine (sic) our business, wanna start a war?"* was seen inside the *Bagle.J*, where as the response message inside *Netsky.F* was *"Bagle you are a looser!!!! (sic)"*. Similar messages with profanity was 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*, also seems to admit that he had written the malware in order to remove infection with *Mydoom* and *Bagle* worms from victim's computer [Wikb].
All three malware family were computer worms spreading through email as an attachment and making victim curious enough to download and open it.

2.2.2 Kill Zeus

This was another war between the two *Trojan* malware families *Spy Eye* and *Zeus*. *SpyEye* is a 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 [Fed]
SpyEye came with the feature called *Kill Zeus* that was successful in removing large varieties of *Zeus* family. The battle between these two family was pretty much dynamic and many releases of the Trojan toolkit were made from both the family in order to negate each others affect [Har].

2.2.3 Shifu

This highly evasive malware family, **Shifu**, targeting mainly Japanese (82%), Austria-Germany (12%), and mix Europe (6%) banks for sensitive data, was detected recently in the year 2015 [Lim]. It uses *Domain Name Generation* to generated random domain names for covert botnet communication. It terminates itself it it finds that it is being run inside a virtual machine or is being debugged, which it finds out by checking system

files such as *pos.exe*, *vmmouse.sys*, *sanboxstarter.exe* or *IsDebuggerPresent* API [Diw].
Shifu had antivirus kind of feature that prevented any other malware from infecting its victim by stopping the installation or making the file look suspicious. It kept the track of all the files being downloaded by hooking *URLDownloadToFile*[1] function. For any unsigned executables coming from unsecure network it renamed the file to *infected.exx* and also send the copy of file to its *C&C* server, probably for further analysis. *Shifu* did not find or delete if the system is prior affected by other malware, but once it had the control of victim's, it managed to stop infection from further malware or any updates to the existing malware by disconnecting it with its botmaster [Lim].

2.3 Malware Analysis

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 types of malware analysis system, *Static* and *Dynamic*.
Static Anaylsis studies the binary by disassembling it with use of disassembler, decompilers, source code analyzers to find the control flow graph of all the code segment and path that the program might take under different given conditions. The analysis is done without running the program, but studying the assembly code of binary to detect the benign or malicious nature of program. *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 of the malware while it runs in host. *Dynamic analysis* is faster than static analysis and gives clear insight of the program behavior and nature. However it might not be feasible to traverse all the execution path that a program might take.
With the increase in the rate of new malware with polymorphic and metamorphic nature, it has become much difficult to detect a new malware from *static analysis*. The syntactic signatures are ignorant of the semantics of the instruction and can be easily evaded with code obfuscation and making program control flow obscure [MKK].
Also, many environment sensitive malware evade the *dynamic analysis* by changing their behavior, and not performing any malicious activities, once they find out they are being run inside analysis environment. This makes detection of evasive malware much more manual process [KVK14].

[1] Downloads bit from Internet. `https://msdn.microsoft.com/en-us/library/ms775123%28v=vs.85%29.aspx`

2.4 Malware Clustering

With the advent of dynamic analysis of malware, there has also been rise in different approach to build a system that could analyze the malware in large numbers. New malware are on the rise at an exponential rate with millions of them being introduced each day. 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. Even dynamic analysis required manual inspection because of diversity in malware variants and its rapid increase. A good clustering system would be able to detect the similarity between malware to find if the new binary is a variant of previous malware and classify them accordingly. There have been many approaches to use automated malware clustering in order to classify binaries based on their behavioral similarity. This can be used to analyze any new binary to find out if it's novel or similar to previously known malware.
Let us discuss about some previous works done with classification of malware based on the behavioral activities of malware.
Bayer, Comparetti, Hlauschek, et al. used execution traces of malware in order to create a behavioral profile which they later used as an input to their clustering algorithm. This was the first approach to use clustering of malware on a large scale. Their system was able to recognize and group similar malware with better precision and scalable manner. About 75 thousands malware were clustered within three hours.
Rieck, Trinius, Willems, and Holz also used clustering and classification using *machine learning* to analyze behavior of malware. Their work was inspired by [Bay+09] and used incremental approach to decrease the runtime overhead, as malware were processed in a daily basis with batch of 1000 [Rie+09].
Rafique and Caballero developed *FIRMA*, a tool that clusters the malware binary into families, based on the network traffic of unlabeled malware binaries. It also produces network signature for each of the network behavior of family. The tool was evaluated against the datasets of *16,000* unique malware binaries [RC13]. Pirscoveanu, Hansen, Larsen, et al. also presented a system to analyze and classify high amount of malware in a scalable and automatized manner, which could be used pre-filter novel from known malware. They used [Cuc, Cuckoo] to generate behavioral profile of a malware and *Random Forest* for *supevised* machine learning. The total number of samples used for experiment were *42,068*, out of which 67% were used for training and remaining 33% were used as testing set [Pir+15].
Yavvari, Tokhtabayev, Rangwala, and Stavrou uses behavioral mapping approach to find the commonalities between malware behavior grouped component wise. They define the term "soft cluster" which represents malware relationship with respect to all behavioral similarities, however small. Using soft clustering approach they revealed

component sharing that belonged to different families according to traditional grouping. The system was evaluated on the set of *1,727* unique malware samples [Yav+12]. Firdausi, Lim, Erwin, and Nugroho in their paper performed different classification using 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. They conclude that machine learning techniques based on behavioral profile could detect malware quite efficiently and effectively [Fir+10].

We too will also be using machine learning techniques in our research to cluster the malware, based on the behavioral profiles generated from dynamic analysis, to classify malware into different probable families. We believe in the notion that malware belonging to same families will have similar behavioral activities, thus more common resource activity. The work done on clustering so far has been done on smaller set of thousands of malware as described above. We use unsupervised machine learning algorithm to perform the clustering of those malware on very large data set of millions of malware. We will discuss in details what we chose for clustering and we perform it in section 3.4.

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 section 3.4 to cluster the malware based on their dynamic behavioral activities into different topic families. We used approach as discussed in section 3.5 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.

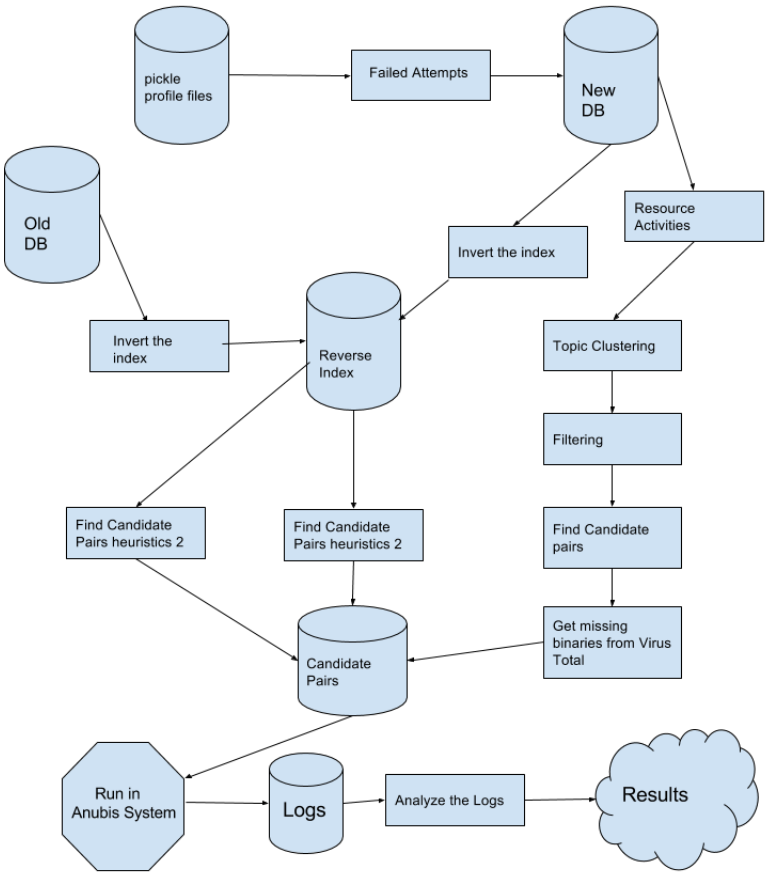

Figure 3.1: Overview of the research and experiment

3.2 Behavioral Profile

Previously, Bayer, Comparetti, Hlauschek, et al. used Anubis system for the dynamic analysis of malware to gets it execution traces [Bay+09]. According to the authors, Bayer, Comparetti, Hlauschek, et al., "**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 [Pyt] 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, Comparetti, Hlauschek, 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
```

3.2.2 OS Operations

Broadly, OS Operation is the generalization of a system call. Bayer, Comparetti, Hlauschek, et al. 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.[Micd, 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* [Micb].
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 [Micf].
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) [Mich].

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 [Micg].
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 [Mice]

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 [Micc].

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 [Mici].
The sync object type covers all the synchronization activities. [Mici].

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 [Wika, 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.

3.4 Topic modeling

We started to explore different clustering approach of malware samples based on their activities, into different Topics/families. To cluster the malware we had multiple options. We looked into the behavioral-clustering paper by [Bay+09, Bayer]. The clustering results were for tens of thousands of malware samples and not millions, and the

approach still does not look like scalable to (16M) millions of samples. 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 75K 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 alternative was VirusTotal labels, but unfortunately are not very accurate.
A third approach was to use a clustering algorithm that is scalable, but maybe less accurate than the behavioral clustering, but able to cluster millions of samples. We decided to map the problem to document clustering (considering each malware as a document, and the its resource activities as its words). As tf-idf 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.

3.4.1 Latent Dirichlet Allocation

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 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.

3.4.2 Gensim

Gensim (**generate similar**) [ŘS10] is an efficient python library developed by Řehůřek and Sojka. We used it to perform the LDA on our resource activities data. We chose it because of its simplicity, well documented API, and ability to work on large corpus.
We used the *ldamulticore*[Gen] 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. Also the training algorithm ran 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 machine, as size of training corpus did not affect the memory footprint. Our corpora data had maximum size of 81 Gigabytes.

3.5 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 [MyS] 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,*

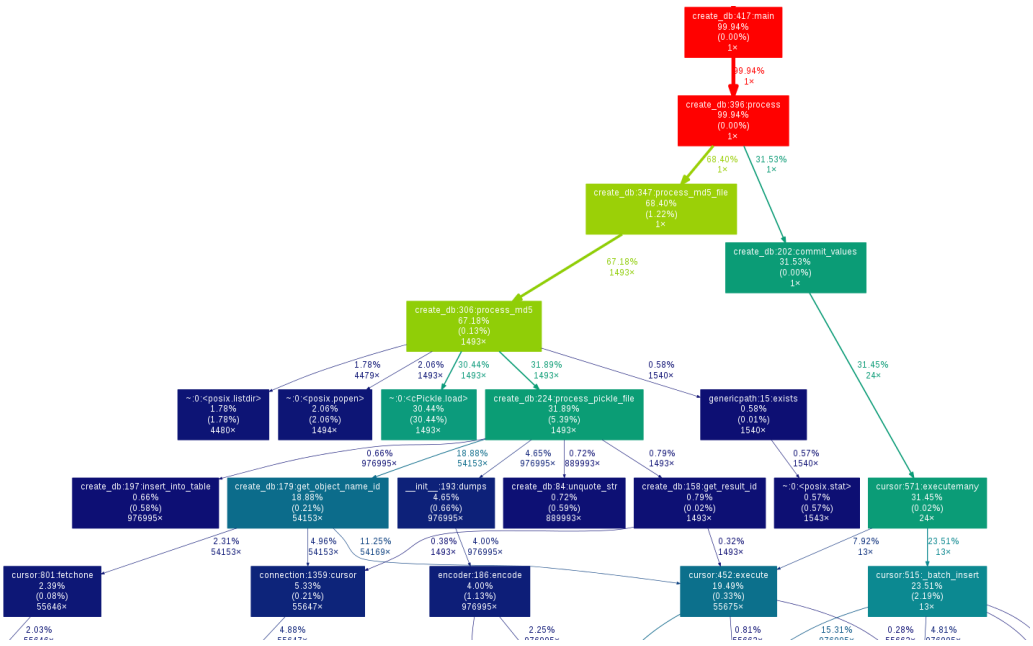

Figure 4.2: profiling of database creation program

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 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 [ŘS10, 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

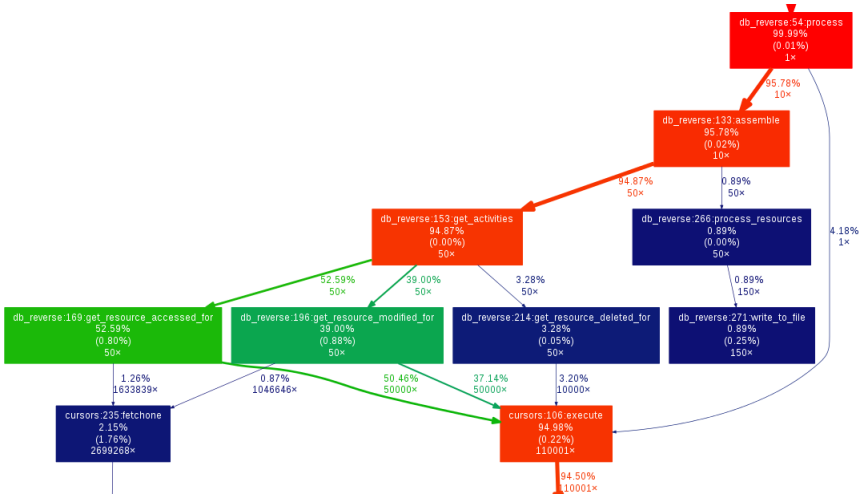

Figure 4.3: profiling for creation of resource activities of malware

Listing 4.6: Script to run Gensim LDA

```
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

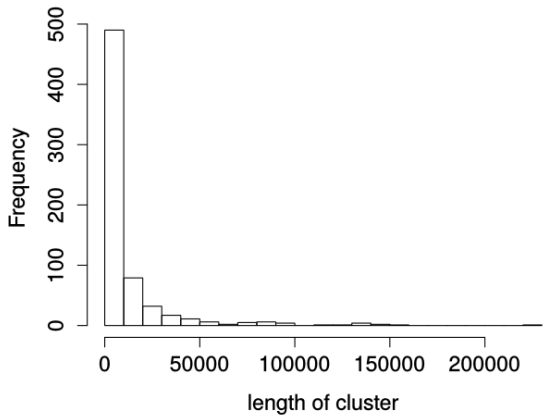

Figure 4.4: Histogram showing the distribution of Cluster size

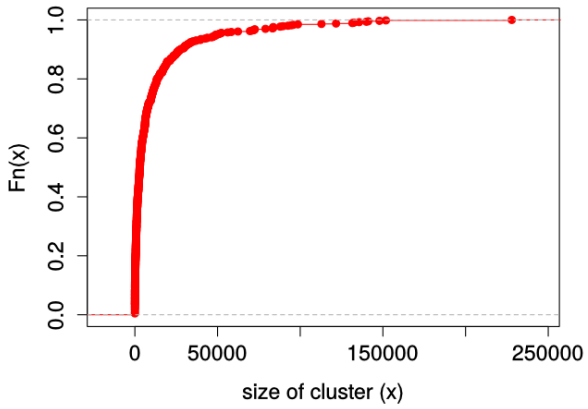

Figure 4.5: Graph showing cdf between size of cluster and topic fraction

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

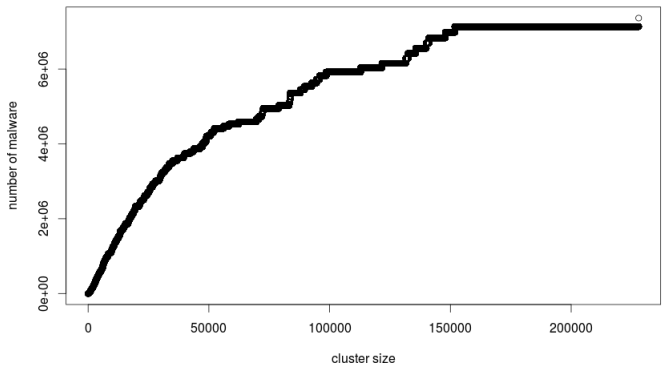

Figure 4.6: Graph showing cdf between size of cluster and total number of malware

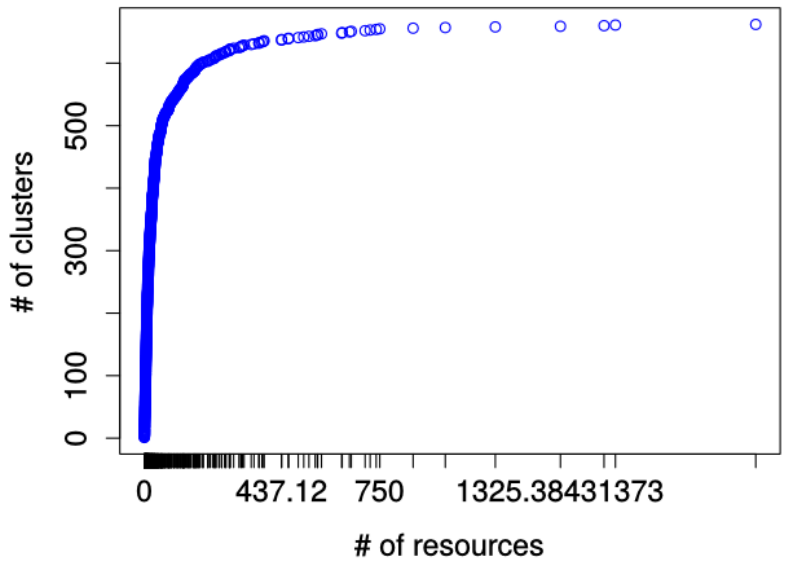

Figure 4.7: Graph showing cdf distribution of common resource between same family topic

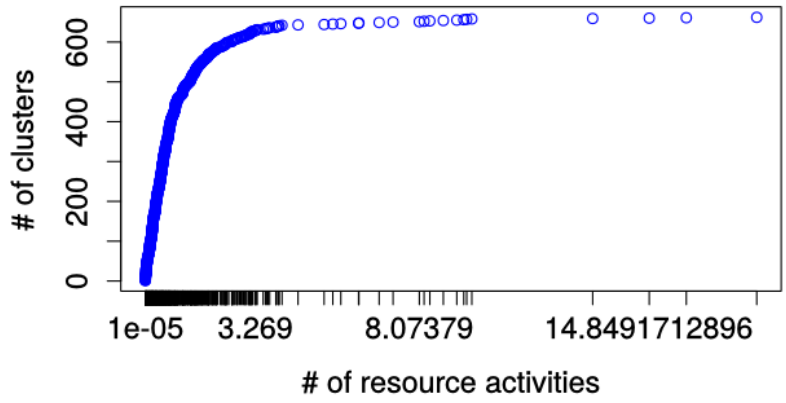

Figure 4.8: Graph showing cdf distribution of common resource between different family topic

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.

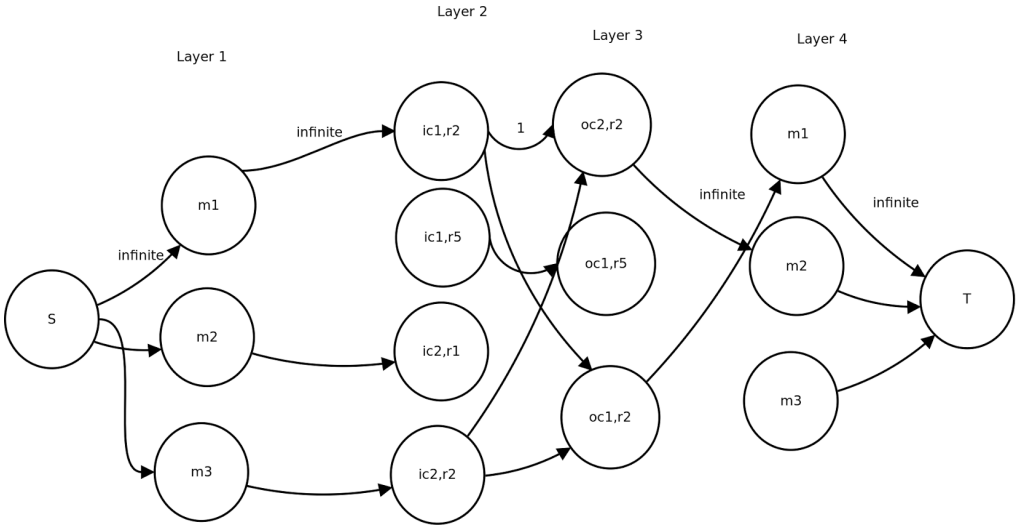

Figure 4.9: Graph representing the max flow implementation

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 3.5*.

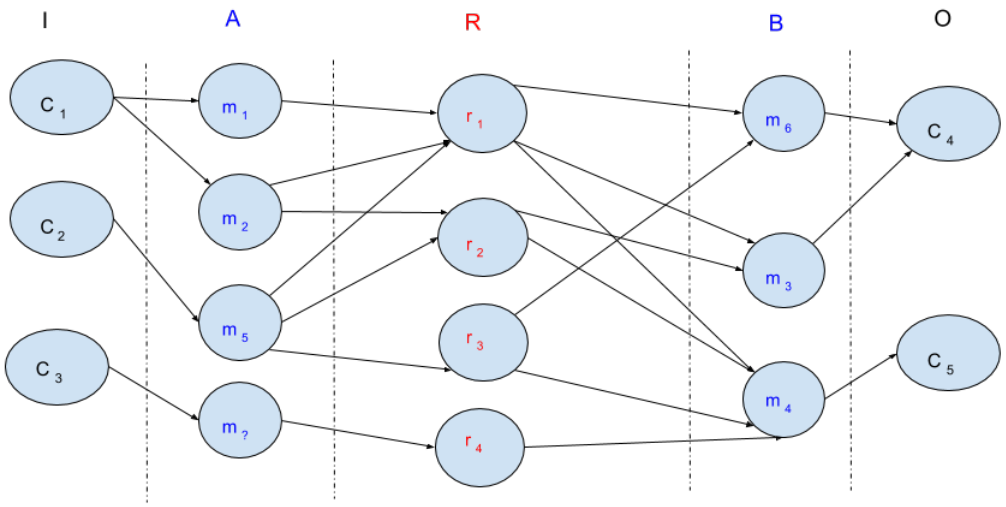

Figure 4.10: Graph showing the interconnection between Cluster topics, Malware, and Resources.

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 Future Work

6 Conclusion

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

Bibliography

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

[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.

[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.

[Cuc] Cuckoo Foundation. *Automated Malware Analysis - Cuckoo Sandbox*. www.cuckoosandbox.org. Last Accessed: 2015-12-30.

[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.

[Gen] 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.

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

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

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

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

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

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

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

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

[Micl] 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.

[MKK] A. Moser, C. Kruegel, and E. Kirda. *Limits of Static Analysis for Malware Detection*.

[MyS] 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.

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

[RC13] M. Z. Rafique and J. Caballero. "Firma: Malware clustering and network signature generation with mixed network behaviors." In: *Research in Attacks, Intrusions, and Defenses*. Springer, 2013, pp. 144–163.

[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.

[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.

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

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

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

[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.