國立交通大學

網路工程研究所

碩 士 論 文

基於內核函數呼叫模式之
惡意程式種類辨認方法

Recognizing Malware Families with
Invocation Pattern of Kernel Functions

研 究 生：劉芳瑜

指導教授：謝續平　教授

中 華 民 國　100 年 8 月

基於內核函數呼叫模式之
惡意程式種類辨認方法

Recognizing Malware Families with Invocation Pattern of Kernel Functions

研 究 生：劉芳瑜　　　Student：Fang-Yu Liu

指導教授：謝續平　博士　　　Advisor：Dr. Shiuhpyng Shieh

國 立 交 通 大 學

網 路 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Network Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

August 2011

Hsinchu, Taiwan, Republic of China

中華民國 100 年 8 月

基於內核函數呼叫模式之
惡意程式種類辨認方法

研 究 生：劉芳瑜　　　　　　　　指導教授：謝續平 博士

國 立 交 通 大 學

網 路 工 程 研 究 所

摘要

惡意程式種類辨認方法是用來判斷一隻被測試的惡意程式是不是屬於某特定種類的成員。任何一種辨認方法都必須有能力產生代表各種類的共同行為特徵。然而，現有的產生行為特徵的方式仍存在漏洞，例如：核心層次的 Rootkit 能夠繞過在分析系統內紀錄有哪些應用程式函式庫被使用的監測方法。在本篇論文中，我們設計了一個能夠產生代表整個種類之惡意程式行為特徵的方法。此方法利用將惡意程式置於虛擬機器中執行，以監視惡意程式的行為。為了讓惡意程式無法繞過本系統的分析，我們藉由在虛擬機器外部設置監控機制，記錄核心函式的呼叫情形。此外，也運用對於整個系統的污染資料流分析，可以得知有哪些被呼叫的核心函式之參數與被測試的惡意程式有關聯。再者，這樣的方式也能夠讓我們追蹤到有跨程序行為的惡意程式，這一個特點是之前與我們目標相同的研究都做不到的。最後將產生的核心函式呼叫記錄轉換成 HMM 的模型，作為表示惡意程式種類的行為特徵。由評鑑結果顯示，利用本系統產生的行為特徵於辨認惡意程式種類時，能夠達到非常低的漏報率。

Recognizing Malware Families with Invocation Pattern of Kernel Functions

Student: Fang-Yu Liu   Advisor: Dr. Shiuhpyng Shieh

Department of Network Engineering

National Chiao-Tung University

Abstract

Malware family recognition is the process of judging whether a malicious binary program belongs to certain family. In this process, a pattern representing a sequence of malicious behaviors shared among malware in the same family shall be automatically generated. Existing mechanisms such as in-system API profiling can be circumvented by some malware such as kernel-level rootkit. In this thesis, a novel scheme is proposed which generates a unique behavior pattern for each family of malware. In our scheme, malware are executed on a virtual machine. By hooking in-kernel functions underlying the VMM, invocation sequences of a malware program cannot be disguised and therefore are accurately profiled. Our scheme covers the whole-system taint analysis to identify the in-kernel function invocations where parameters are contaminated by the malware being tested. Our scheme also tracks cross-process malware, which is not covered by previous work. Profiled invocation sequences are further converted to HMM-based pattern. The evaluation result shows that our behavior patterns give extremely low false negative in the recognition phase.

誌　謝

一眨眼就在交大過了兩年，這個我從小就很嚮往的學府。首先，非常感謝爸爸、媽媽，辛苦賺錢並給我一個幸福的家庭，讓我的求學之路快快樂樂、沒有後顧之憂。以及非常疼愛我的在天堂的奶奶，很遺憾的是沒辦法跟您當面分享我穿了畢業袍還有找到工作的喜悅心情，希望您在天堂也能夠看見我的成長。

我以身為DSNS Lab為榮。喜歡大家在炎熱的天氣，仍然熱烈的討論中午要出去吃什麼，最刺激的是還會臨時改決定;喜歡大家不管在多大的課業重擔下，仍然會自己找樂子發洩壓力，研究好厲害也好會玩;喜歡每個月的實驗室慶生，這是一般實驗室所少有的溫馨場景。感謝謝續平老師，時時指引我的研究方向，並且教導我們正確的研究態度。感謝辛苦的助理，明華是一個好媽媽，關心我們的生活與教導我們待人處世的態度;慧雯經常與我分享生活趣事，還有分享便當，太貼心了。感謝辛勤的工程師們，Michael、Sky、Pokai、大佩、小佩，經由與你們相處，得知許多關於業界的文化，開闊我的視野。感謝實驗室的學長姐，繼偉、Vic、Mashi、梅子綠、花生、Fallman、阿不拉、Alan、嘉偉，無私的指教，每次問學長們問題，總是非常熱心的回答我。特別感謝繼偉與嘉偉，花了很多的時間指導我的論文，一次又一次很有耐心地與我討論，讓我終於能夠完成這篇論文。感謝與我同屆的峯哥、江江、修醇，大家一起努力加油打氣終於走到這一天。感謝碩一的天天開心、昀旻、凱任、凱峰，還有碩零的學弟妹們，實驗室因為有你們更有活力了。

我要感謝師大資工98級新竹幫，怡欣、阿貞、佳慧、阿揚、強強、小黑、KAKA、猴爺、軟春、良叡，還有在交大認識的小飛俠，很開心有你們這群朋友的陪伴，當我的精神支柱，時常聽我分享生活中大大小小的事情，難過的時候給我安慰，給我好多溫暖，好喜歡你們。

感謝在交大陪伴我的所有人，我要將這份畢業的喜悅與成就歸功於你們。

Table of Content

List of Figures

List of Tables

1. Introduction

Malware (malicious software) remain a serious problem in spite of the wide use of various anti-virus applications. For the time being, thousands of new malware are being generated per day. According to the reports [1] [2], there are 1,017,208 instances of new malware were detected in the first half of 2010, approximately 10% more than the previous half year. The malware writers continuously develop new methods of polymorphism and metamorphism such as obfuscation, encryption, or packing to evade signature-based detection. Furthermore, metamorphism enables malware to change its appearance when every time it propagates. To deal with such large numbers of malware instances efficiently, automatically deriving representative malware behavior patterns, which are used to recognize a whole malware family, is necessary. Fortunately, the observation that numerous malware share common behaviors enables us to derive a generalized signature for each group of them. In doing so, testing whether a malicious program belongs to an existing well-known group of malware can be determined efficiently. In this chapter, we give a brief introduction to existing related schemes, proposed methods, and our contribution.

1.1. Background

For recognizing malware families with behavior pattern, in this section, we indicate that why not use signatures but behavior patterns, the existing monitoring mechanisms and their drawback, and the malicious behaviors generally focus.

1.1.1. Behavior Patterns

Signature-based recognition is the most widely used approach, but one signature could not identify other malware. Due to the continuously development of malware program, it is no longer valid to deal with the large number of mutant malware programs.

Using behavior patterns to recognize malware family is efficient. As indicated by recent studies [3], each malware instance in the same family shows similar behavior patterns. Because most of original malware are created by the same authors, they also have several different versions though many times upgrade. In addition, other authors often rewrite the existing malware programs. According to these reasons, one behavior pattern is useful to recognize lots of malware that in the same family.

Checking arguments of API is effective. When the malware is executed, it must invoke APIs with several arguments. Hence, the API invocation sequences are adapted to represent the malware behavior. Moreover, each API's name and argument represents with meaningful word, so that it is easy to use when analyzers want to functionality of malware programs. To profile arguments of known malware and the frequently used arguments of each family could be apply as behavior pattern for future recognition.

1.1.2. Monitoring mechanism

In order to observe malware behavior, based on considering the system call workflow from user level to kernel level, we separate the monitoring mechanisms into two perspectives for discussion. The former mention that what kind of the object we monitor, the latter is about where to monitor. In addition, we define" **in-kernel function**" as the low level kernel functions that system call must invoke. The reference of the section is depicted in Figure 1.

Objectives for monitoring: Monitoring on user-level library APIs, attacker cloud invoke system call directly without using the user-level library APIs, therefore, the monitor mechanism is bypassed. Otherwise, monitoring on system call, Rootkit could enter kernel level directly instead of invoking system call, hence the monitor is invalid.

In-system monitoring: As long as the monitor and malware exist in the same circumstance, the monitor mechanism could be overridden by in-kernel level Rootkit. No matter monitoring on kernel level APIs or user-level library APIs, such us the approaches in the previous mention, the result are no different.

1.1.3. Malware Behavior

In this paper, we monitor process, registry, file, network as recent studies and the famous malware analysis website [4] [5] [6] , because all malware have the subset of these four types object's behaviors. Out research is integrity and sufficient that not less than other related works. Running malware under our monitor system, the outcome is a human readable report which profile malware behaviors. The report contains sufficient information, including the cross-process malware interaction, the contents of malware communication over network, registry modification, dynamic API loading, etc.

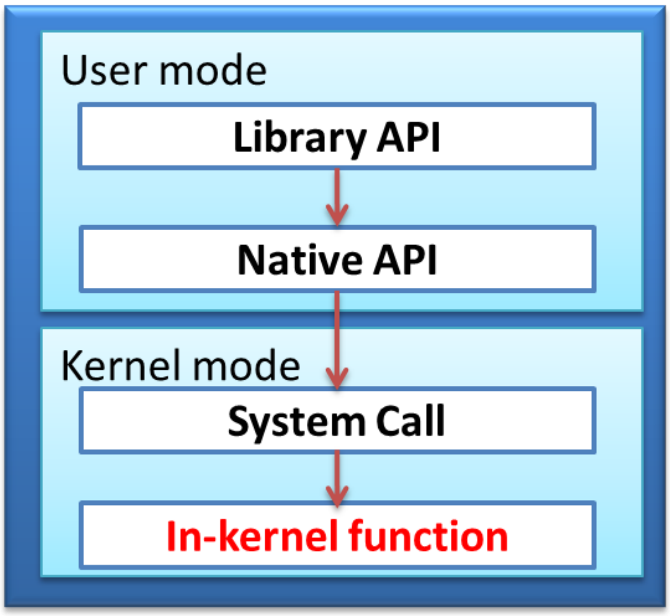


Figure1. System calls workflow

1.2. Requirement

We believed that an ideal recognizing malware family system should provide following features: **Automatic behavior pattern generation**, in order to cope with malware efficiently; **Accuracy**, means that using behavior patterns to recognize malware family with low false positives and low false negatives; **Non-circumventable**, no malware is liable to bypass the monitor, which is to ensure that the system could get the malware behavior completely.

1.3. Concept

In order to achieve non-circumventable monitor, we analyze malware behavior by collecting in-kernel function calls and arguments from the underlying emulator. All of above monitor mechanisms are too easy to bypass and cannot capture the malware behavior information completely. Because of no matter how malware program avoids using user-level API, it must invoke in-kernel function finally. For the reason, we monitor in-kernel functions even arguments. Also, when monitoring in-kernel level functions, in-kernel level Rootkit could override the monitor mechanism. To overcome this problem, we use out-of-box hooking technique, to build our monitor on the underlying emulator, so monitor and malware are not in the same space that the monitor mechanism works well.

For the purpose of recognizing malware family with high accuracy, we use tainting to precise the monitor result. Taint could track which arguments are related with malware. When a monitor system working without taint, it cloud only distinguish process between tested malware and other program by the help of the CR3 processor register, nevertheless, not know which arguments have high relationship with malicious behavior. Especially, taint could monitor relations between data across multiple processes, even in kernel. Using taint help us to get the malware information more completely, thus, improving the accuracy of recognize malware family.

We produce an automatic pattern generation system, the basic function of malware analysis. We extract invocation sequence to dilute unrepresentative information, in order to precise the behavior traces before generate pattern. For example, when in-kernel function arguments involve meaningless string such as hashed filenames, this information must be dilute. Finally, the system describes in-kernel function transitions with Hidden Markov Model (HMM). The HMM is easy used to recognize malware family, so is suitable for our system.

1.4. Contribution

In this paper, we proposed a novel approach to generate a behavior pattern for a family of malware. In addition, two important features distinguish our work from existing researches. Firstly, unlike previous approaches, which can be circumvented by lower-level hooking or overwriting, our out-of-box in-kernel function hooking is inescapable for malware being tested. Secondly, taint-based argument checking gives more accurate behavior profiling because the taint status help us differentiate between arguments fed by malware and those by benign programs running in background. Thirdly, the taint propagation is done system-wide, and it can hence deal with cross-process malware, which are not covered by previous work. Obviously, our system produces more complete malware behavior patterns than other approaches. An experiment on 511 malicious samples originating from 15 different families was performed. The evaluation result shows that our behavior patterns give zero false positive and low false negative (less than 5.8%) at recognition phase.

1.5. Synopsis

The paper is organized as follow. Chapter 2 gives introduction to related works. Chapter 3 gives the detailed description of our system. Implementation and evaluation are in Chapter 4 and Chapter 5. At the end of the paper, we make an overall conclusion in Chapter 6.

2. Related Work

Automatically grouping malware of the analysis results is a necessary procedure in some malware research [7] [8] [9] [10]. The crucial condition to get accuracy grouping result depends on the ideal analysis results. For this purpose, researchers have to design an analysis method.

The figure of similarity pattern is the key point for every system that aims to group malware to identify a malware family. To this end, the systems have to consider how to represent each malware analysis result is suitable; moreover, which evaluation model is able to fit the analysis result and appropriate to compute the distance between malware. These approaches to generate malware family pattern can be divided into two parts: static analysis and dynamic analysis techniques.

2.1. Static analysis

Static analysis [24] [30] [36] is the technique of analyzing executable program without executing it. These propose work by disassembling the binary first. Existing method use control flow and dataflow analysis techniques to describe the analyzed program. The most significant advantage of static analysis is usually faster than dynamic analysis. On the other side, the main weakness is ineffective against the polymorphic and metamorphic malware. There exist the self-modifying programs that packed executable instructions which often related to malicious behaviors and unpack themselves during run-time. Static analysis is useless in this situation because it cloud not get the complete instructions without execute the program. Following are related works that classified by the focus objective.

Opcode. The approach was presented by Bilar et al. [11]. The system decomposes malware code through statistical analysis of opcode distributions which used to distinguish malware samples and non-malicious samples. However, it also has weakness in general static analysis method.

Function or String. Tian et al. [12] measure the function length by the number of bytes that in the code. In addition, considering the malware function length frequency appear in a malware sample. In [13], the author proposes an automated malware classification system. Classification of malware based on the function length and the printable strings information. To examine the feature of which string often be used as an argument. As long as, malware writer insert junk instructions, replace instructions or registers, these low-level assembler mnemonics would change the code size and appearance, these malware signature is no use.

API calls. Sathyanarayan et al. [14] suggest generating signatures and detection of malware families based on the semantics and API calls. The system extracts and statistics the frequency of the critical API calls from the executable program to evaluate the likelihood test then identify the malware family. Unlike the previous static analysis systems, this approach is immune to common obfuscations that only affect the code pattern but do not change the behavior. Because of considering the API calls, such obfuscations have no effect on this signature generation approach. But still invalid against analysis the packed malware.

2.2. Dynamic analysis

Dynamic analysis is the technique of supervising executable program during run-time. This technique is a solution to break the limitations of static analysis. While dynamic analysis has no problem with polymorphic and metamorphic malware, and is effective to deal with self-modifying programs. The method in this paper is closely related to these works. This section discusses different approaches for generating malware family pattern with dynamic analysis, and compares them with our approach.

Instructions. In [15] [16], grouping instructions together as the malware behavior information to identify whether the tested program is benign or malicious. First, the system captures runtime instruction sequences from an executable program and organizes instruction sequences into basic block. Then, extract the instruction groups that frequently used within basic blocks. Compared with our approach, using instruction information to deals with the malware is probable to evade by metamorphic malware. Other researchers [17] have proposed to detect malware through checking instruction semantics. The weakness of the approaches is that it has no ability to handle the reorder instructions situation. Nevertheless, the obscure technique would not change the behavior of malware, has no effect on our behavior pattern.

Network trace. These systems use the extracted features such as HTTP requests [18], domain names [19], and similar communication patterns [20] for clustering. Our system monitor the network communication completely, we not only analysis network-based malware, but also other malicious programs.

System call graph. These researches [22] [23] capture the system call invocation traces and construct to the system-call graphs, then design a method to extract the sub-graph as the behavior pattern. Using an algorithm to compute the distance between the patterns, and identify the malware family. The difference with our approach is that they monitor behavior on the system calls but our system monitor on in-kernel functions. As mentioned above, monitoring on system calls would be bypassed by Rootkit, but non-circumventable on the in-kernel functions.

Arguments. [4] [24] [26] [27] [28] [29] extract patterns from malware invoked system call and arguments to identify malware family. Park et al. [21] use of one representative common behavioral graph that is created from individual behavior graphs for all malware instance in the same family. The approach that present by Bayer et al. [4] use out-of-box hooking and focus behavior on the file, process, network, registry as our system. But the system has not use taint technique, could not monitor the behavior between processes and know the malware related argument explicitly. The system [27] in the related works is closest to ours, not only use out-of-box hooking but also taint analysis. It taint all the system call‘s arguments and return values. Every time system call is invoked, check whether any argument is tainted, if the answer is yes, log the system call and arguments on the track record. All system calls serve as its taint sources, but we have a whole-system taint, so that our monitor result must be more accuracy. All the related works in this section use system call level monitor mechanism, as we have seen, it is insufficient.

3. System Overview

Our system captures invocation sequence of in-kernel functions with arguments and generates behavior patterns that use to recognize malware families. We can roughly divide the procedures into three components, behavior monitor, pattern generator, and family matcher. Firstly, in the training phase, behavior monitor plays an important role of supervising malware and produces a behavior trace. Secondly, pattern generator train the same family's malware behavior traces to a family's pattern. Finally, the testing phase has a component called family matcher, the system state can recognize a tested malware in or not in the each family. Furthermore, the system focus of the object is standalone malware instead of infecting malware and VM-Aware malware is not in the scope of our discussion. The reference of the chapter is depicted in Figure 2. We would describe them in following.

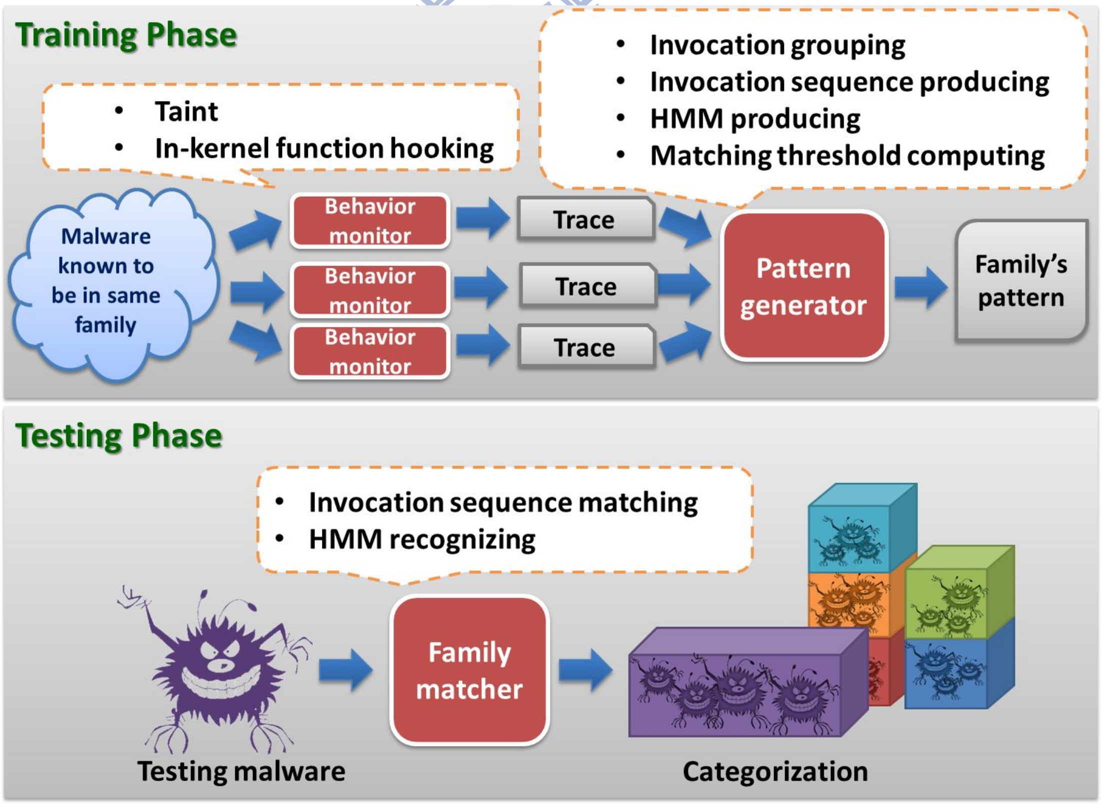


Figure2. System overview

3.1. Behavior monitor

The malware behavior monitor component of our system is based on QEMU, a whole-system emulator. We extended QEMU [30] with out-of-box in-kernel functions hooking and whole-system taint tracking that used the tainting system in previous work [32]. Putting a malware into emulator, then analyzing malware behavior by collecting in-kernel function calls and arguments from the underlying emulator, we obtain the malware behavior trace finally.

Taint. We use a system-level emulator (QUMU) with taint capability which covers registers, memory, and HD. Using tainting is helpful to know which invoked in-kernel functions' arguments are related to the monitored malware. To start tainting, we import a malware to HD sectors, and these sectors sever as taint source. After that, to execute the malware for five minutes, at the same time, check whether any argument of the invoked in-kernel function is tainted.

In-kernel functions hooking. Hooking on the in-kernel functions, rather than user-level APIs or system calls make malware hard to bypass our monitor. QEMU utilize dynamic translator doing a runtime transform the target CPU instructions into the host instructions. During the transform procedure, we check if any hooked in-kernel function is invoked. Our hooking focus on the four types of the in-kernel function, such as process, registry, file, network, and choose the functions that necessary for malicious activities, moreover, care the arguments that include the related object's specific properties. For instance, malware could not access network without invoke the sending packet function in the net card driver, and we analysis argument involve the packet content of the sending packet function.

3.2. Pattern generator

Pattern generator use the traces of the family being trained as the materials and produce the family typical pattern, that consisting of a HMM, representative string for each state in the HMM, and a threshold for matching the probability. As illustrated in Figure 3, generating malware family pattern is a multi-step process. It consists of an initial, log clustering, invocation sequence producing, HMM producing, and a final matching threshold computing.

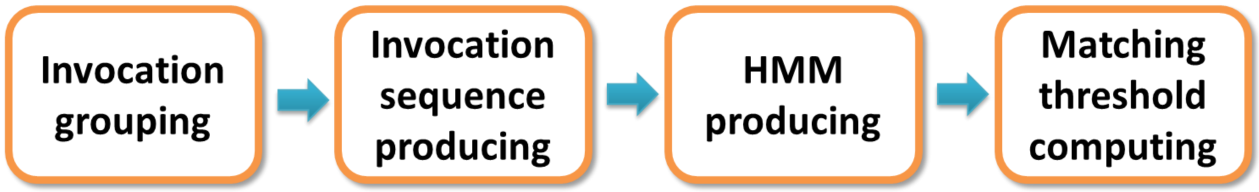


Figure3. Procedure of Pattern generator

Invocation grouping. Disposing the arguments that profile in the trace report and then using clustering method to group similar arguments with distances under certain threshold into same clusters. Begin with the arguments disposing, we dilute the unrepresentative strings and expand the influence of the meaningful information, in order to make sense of the traces then cluster argument more reasonable. The first one is that we remove common log items appearing in many families, the log items mean each invoked in-kernel function with its arguments record in the trace. The reason is that common log not only could not represent a particular family's behavior but also appear in each trace too many times to reduce the degree of difference between each trace. Secondly, remove meaningless items such as Windows random number seed that the monitor system often performs this procedure even in benign programs. Also, capturing invocation sequences with relaxed string matching is means that we replace long digital-alphabetic sequence with repeated characters of equal length, the long digital-alphabetic sequence for example from a hashed string is meaningless but make the log items look divergence. The last one, replace continues identical log items with one single item. In addition, these is exist integer arguments in the in-kernel functions; each assigned value is corresponding to a particular purpose. We replace the integer with the particular purpose name to enhance the meaningful information. After disposing arguments, for clustering the log items with their distance, we use existing method “Complete-Link Clustering,” it tends to find compact clusters of approximately equal diameters. Before clustering, we assign the **distance threshold** in the beginning. Complete-Link Clustering map each log item to a coordinate, and grouping two clusters when **distance threshold** that we had assign is bigger than the maximum of the distance between any two points in the two clusters(**distance threshold**>$D(X,Y)$), as can been seen in Figure 4. Finally, we pick most representative string for each cluster for the purpose of recognizes tested malware efficiently later. In this paper, the clusters that profile in this step called argument clusters.

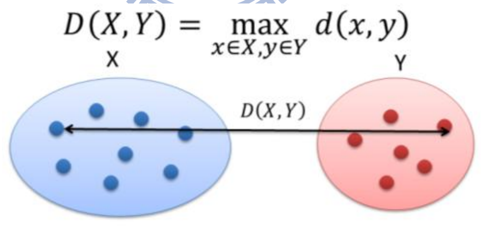
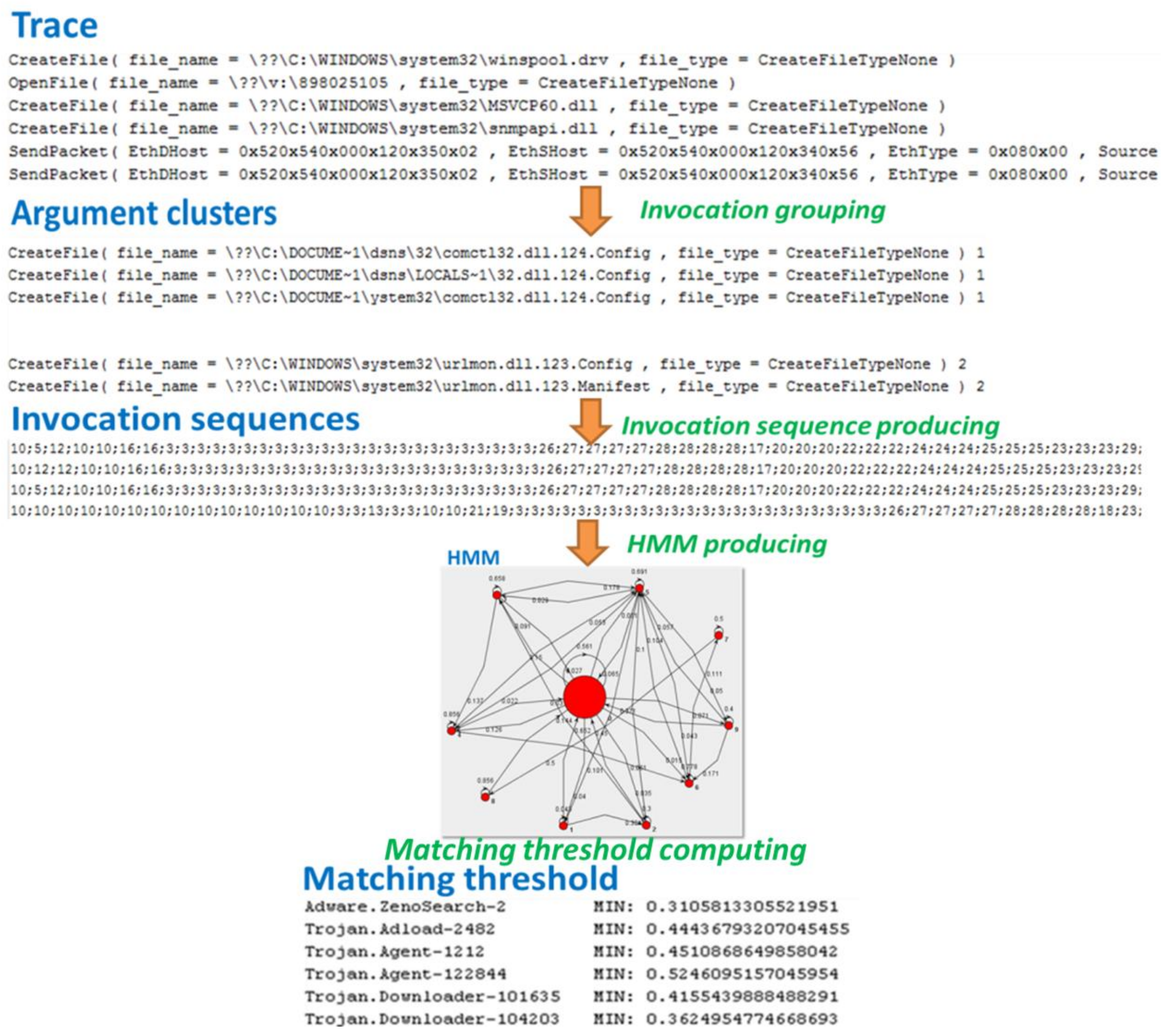


Figure4. Complete-Link Clustering

Invocation sequence producing. When the previous step finished clustering log items, the next task is to produce invocation sequences. For ease of comparison similarity between traces, we map each log item to cluster number, and each log shall be transformed to a sequence of clusters.

HMM producing. In this step, we use the same family’s malware traces to train a HMM as the family’s behavior pattern. HMM is an existing approach. It is a sequence-based correlation model and able to evaluate the deviation between a sequence and the model, that is suitable for our sequence traces and required features for our goal, we use HMM to this end. Every integer within the invocation sequences is a cluster number; as a consequence, each argument cluster shall be an HMM state. Then, to train HMM of the family through the sequences that acquired in previous step. In the end, we add a null state with transition probability is equal to zero, that the transition probability for HMM is a possibility of transition between states, however, for the reason why we would explain in the Family matcher section.

Matching threshold computing. At last, the output of this step is a family matching threshold. We collect all the invocation sequences that in the same family and compute under a particular family HMM, each invocation sequence would get a value, and decide the range from the minimum value is the family's matching threshold. In this paper, we present related value, a value that comes from a HMM computed result. In particular, we emphasize that each family has its own matching threshold and each invocation sequence compute under each family's HMM would get the different related value.



```
CreateFile( file_name = \??\C:\WINDOWS\system32\winspool.drv , file_type = CreateFileTypeNone )
OpenFile( file_name = \??\v:\898025105 , file_type = CreateFileTypeNone )
CreateFile( file_name = \??\C:\WINDOWS\system32\MSVCP60.dll , file_type = CreateFileTypeNone )
CreateFile( file_name = \??\C:\WINDOWS\system32\snmpapi.dll , file_type = CreateFileTypeNone )
SendPacket( EthDHost = 0x520x540x000x120x350x02 , EthSHost = 0x520x540x000x120x340x56 , EthType = 0x080x00 , Source
SendPacket( EthDHost = 0x520x540x000x120x350x02 , EthSHost = 0x520x540x000x120x340x56 , EthType = 0x080x00 , Source
```

```
CreateFile( file_name = \??\C:\DOCUME~1\dsns\32\comctl32.dll.124.Config , file_type = CreateFileTypeNone ) 1
CreateFile( file_name = \??\C:\DOCUME~1\dsns\LOCALS~1\32.dll.124.Config , file_type = CreateFileTypeNone ) 1
CreateFile( file_name = \??\C:\DOCUME~1\ystem32\comctl32.dll.124.Config , file_type = CreateFileTypeNone ) 1


CreateFile( file_name = \??\C:\WINDOWS\system32\urlmon.dll.123.Config , file_type = CreateFileTypeNone ) 2
CreateFile( file_name = \??\C:\WINDOWS\system32\urlmon.dll.123.Manifest , file_type = CreateFileTypeNone ) 2
```

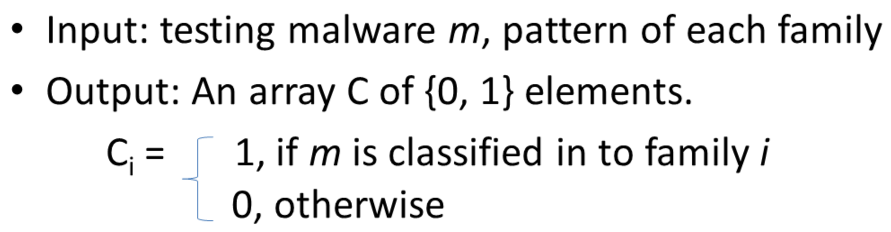
```
10;5;12;10;10;16;16;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;26;27;27;27;27;28;28;28;28;17;20;20;20;22;22;22;24;24;24;25;25;25;23;23;23;29;
10;12;12;10;10;16;16;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;26;27;27;27;27;28;28;28;28;17;20;20;20;22;22;22;24;24;24;25;25;25;23;23;23;23;2
10;5;12;10;10;16;16;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;26;27;27;27;27;28;28;28;28;17;20;20;20;22;22;22;24;24;24;25;25;25;23;23;23;29;
10;10;10;10;10;10;10;10;10;10;10;10;10;10;3;3;13;3;3;10;10;21;19;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;3;26;27;27;27;27;28;28;28;28;18;23;
```

```
Adware.ZenoSearch-2          MIN: 0.3105813305521951
Trojan.Adload-2482           MIN: 0.44436793207045455
Trojan.Agent-1212            MIN: 0.4510868649858042
Trojan.Agent-122844          MIN: 0.5246095157045954
Trojan.Downloader-101635     MIN: 0.4155439888488291
Trojan.Downloader-104203     MIN: 0.3624954774668693
```

Figure5. Pattern generator’s output

3.3. Family matcher

In our system testing phase, we use family patterns that created previously to recognize whether a tested malware is the member of the particular family or not for each family, and the outcome is an array that consists of zero and one. If the *i*’th element of the array value is one, means the tested malware is the classified in to family *i*.



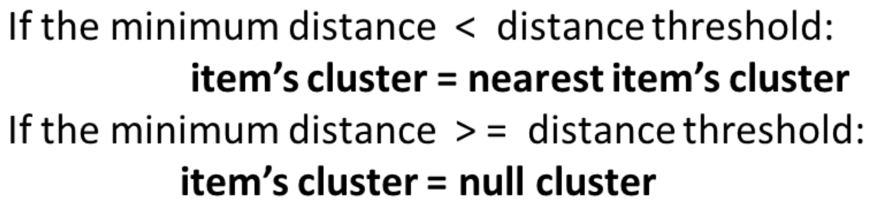
- Input: testing malware *m*, pattern of each family
- Output: An array C of {0, 1} elements.

$$C_i = \begin{cases} 1, \text{ if } m \text{ is classified in to family } i \\ 0, \text{ otherwise} \end{cases}$$

Figure6. Family Matcher’s Algorithm – I/O section

Before doing the matching procedure, we have to monitor the tested malware in order to produce the testing trace. Next, perform the invocation sequence mapping and the HMM recognizing for each family.

Invocation sequence mapping. In this step, we translate the testing trace into the family’s tested invocation sequence. For the purpose of mapping tested malware each behavior log items with the family’s argument clusters, we evaluate the distance between a tested log items and representative string of each cluster. In this way, we keep track of the nearest item that has the minimum distance from the transforming tested log item; moreover, compare the value of minimum distance and the **distance threshold** that has assigned when doing the family’s log clustering.



If the minimum distance < distance threshold:
item’s cluster = nearest item’s cluster
If the minimum distance >= distance threshold:
item’s cluster = null cluster

Figure7. Invocation sequence mapping – setting cluster

To illustrate in detail, if the minimum distance is bigger than the **distance threshold**, the log item’s cluster is equal to the nearest item’s cluster, otherwise, the log item’s cluster would assign to a null cluster. The null cluster corresponds to a uniform value and do not conflict with the other argument clusters. Then, we obtain a tested argument sequence finally.

HMM recognizing. The last step of our system, we evaluate the tested invocation sequence under the family HMM, and identify whether the tested malware is the family's member or not. As mentioned above, an argument cluster corresponds to a HMM state. In addition, the null state in the HMM we have added before corresponds to the null cluster. Null cluster is mapped when the tested log items that could not map to the family's cluster, so it does not exist in the training malware traces that no state would transition to null state, we had set the null state transition probability [33] to zero. For the purpose of increase the identify flexibility, our system cut the tested malware invocation sequence into small fragments of equal length for evaluate, and we have to consider the length of a fragment (**fragment length**). The principle is that HMM evaluate the probability according to the state transition sequence. The longer length of the tested sequence the evaluation would be less flexibility, for instance, when only the argument sequence equal to the training malware's could get a high probability, the other malware that even in the same family maybe have a very low probability. After cutting, all of the state has the probability to be the beginning state of the small fragments, so we set each state with the same initial probabilities. Evaluating each fragment against with HMM, and get its probability. Then, compare the average probabilities of all fragments to the family matching threshold; if the average is upper than the family matching threshold, the malware is a member of the malware family and mark "1" at the family's position in the outcome array, otherwise, is not and mark "0".

```
sum = 0;
n = len(seq) / fragment_len; //cutting pattern into pieces
for i = 1 to HMM.nbstates()
        hmm.setPi( i, 1);      //setting the initial probability
for j = 0 to n-1    //evaluating each piece's probability
        subseq = seq.sublist(j*fragment_len, fragment_len);
        sum += hmm.probability(subseq);
avg = sum/n;
```

Figure8. HMM recognizing's Algorithm

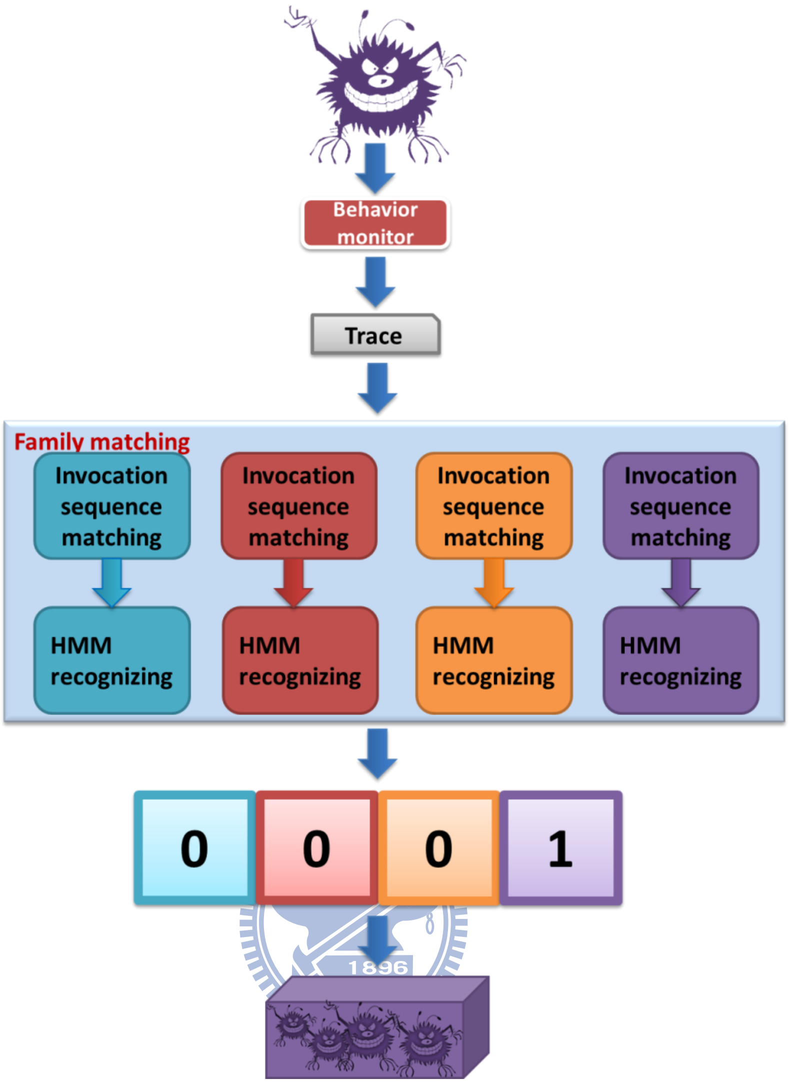

Behavior monitor
Trace
Family matching
Invocation sequence matching
Invocation sequence matching
Invocation sequence matching
Invocation sequence matching
HMM recognizing
HMM recognizing
HMM recognizing
HMM recognizing
0
0
0
1

Figure9.Family matching procedure

4. Implementation

Based on our approach, we design a system that has three components, there are a monitor has ability to hooking the in-kernel functions and tainting the whole-system, a clustering system, and a HMM tool. The whole-system tainting mechanism using the existing system in our lab machine that built upon the research [32]. We implemented other functionality of the system. Following are our descriptions in detail.

4.1. In-kernel function hooking

Our monitor is based on QEMU, an open-source system-wide emulator, using dynamic binary translation in order to run an unmodified operating system with high execution speed. The dynamic binary translation works on a basic-block at one time, where a basic block is an instruction sequences that ends with a branch or jump instruction. Because of translate several constructions at once is more efficiency than only one. The binary translator procedure is to translate a basic block which had not been translated, then execute it, and continue to translate the next basic block until the process stop executing.

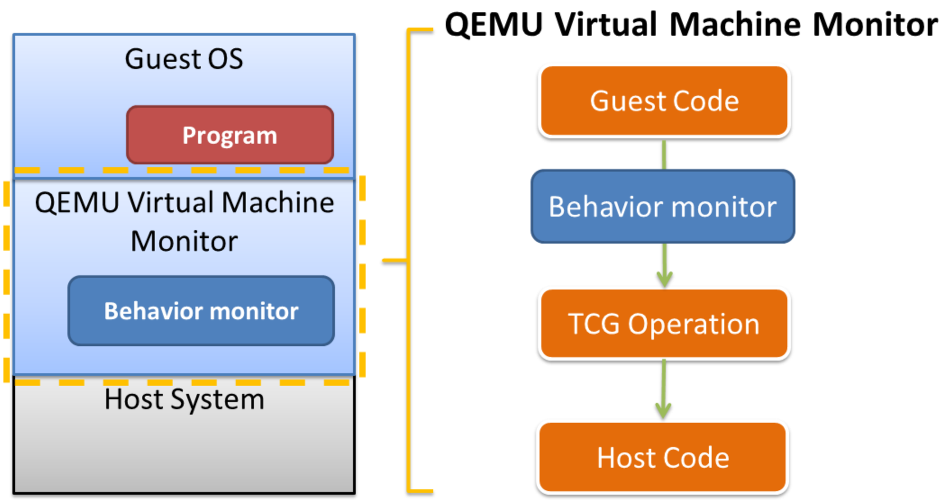


Figure10.Modified QEMU dynamic binary translation procedure

Modify QEMU for in-kernel function hooking. Each time QEMU translate a basic block it must be invoke our **match functions** that used to know whether the translating basic block is our monitoring in-kernel function or not. If the answer is yes, we would call the **helper functions** that we implemented to check the arguments' taint condition and profile the invoke record, otherwise, do nothing. Match function identify the in-kernel functions according to compare the several instructions that at the beginning of the basic block with the in-kernel function patterns where a pattern is a several previous instructions from an in-kernel function. When the hooked in-kernel function is invoked, the monitor get the memory location of the arguments that we care as discussed in chapter1.3 to do the tainted check. If any argument that in the same function has tainted, we would add the record on the trace.

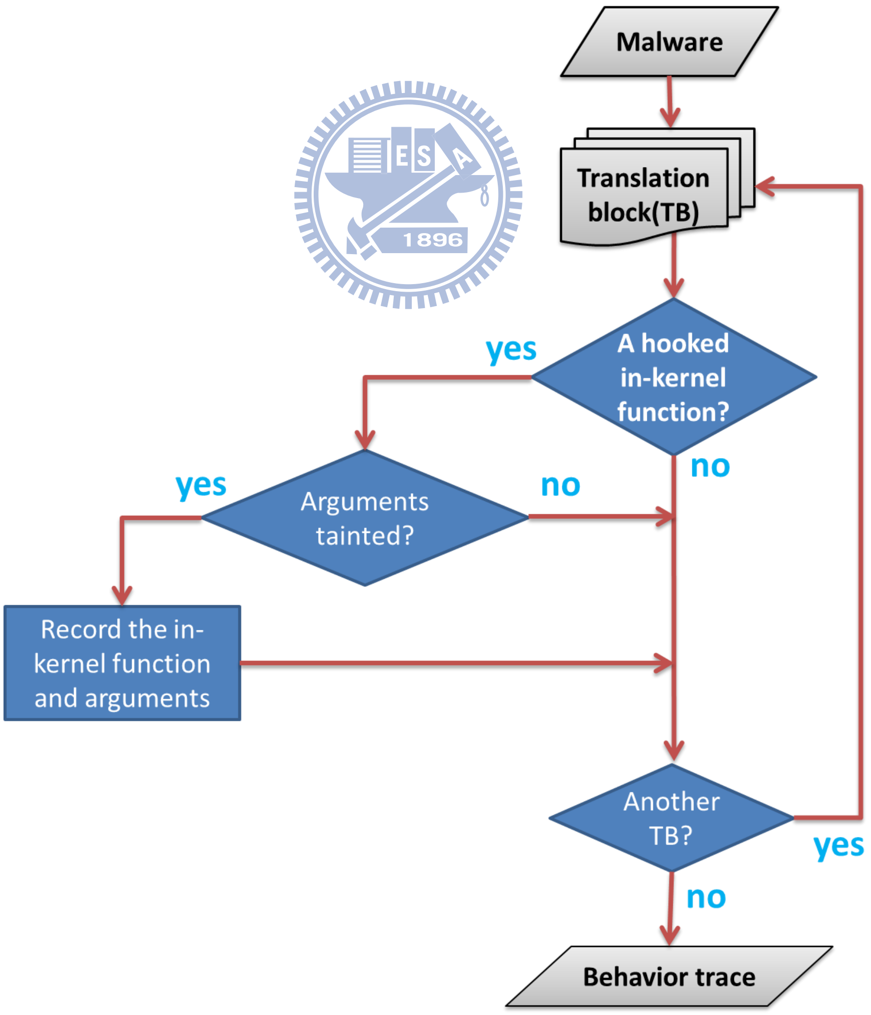


Figure11.In-kernel function hooking workflow

Table1.Hooked in-kernel functions and arguments

Object	Interaction	Function & Monitor arguments	Explain
Process	Create	**MmCreatePeb**	
		PEPROCESS TargetProcess	Pointer to the process structure that involved with the process CR3 and name.
	Delete	**MmDeleteProcessAddressSpace**	
		PEPROCESS Process	As the former
Registry	Create	**CmSetValueKey**	
		PUNICODE_STRING ValueName	The name of the value entry.
		PVOID Data	The data for the value entry.
	Delete	**CmDeleteValueKey**	
		UNICODE_STRING ValueName	The name of the value entry.
File	Create	**IopCreateFile**	
	/Open	*POBJECT_ATTRIBUTES ObjectAttributes*	The name of the file.
	/Write	*ULONG Disposition*	Indicate Create, Open, Write
		CREATE_FILE_TYPE CreateFileType	The type of the file.
	Delete	**IopDeleteFile**	
		PVOID Object	The name of the file.
Network	Send packet	**MiniportSend**	
		PNDIS_PACKET Packet	Pointer to the packet structure that involved with the Src/Dst mac, ip, port

4.2. Clustering system

We use the “Complete-Link Clustering” java library to implement the system.

4.3. HMM tool

To realize our design, we implement the HMM pattern generation with “Jahmm” [34], which is a HMM library implemented in Java environment. It supports a learning algorithm, which generates automatically a matrix consisting of transition probabilities and observation probabilities from several observation sequences. We add the “null state” into a generated HMM by augmenting the two-dimensional probability matrix with an additional row and column of zeros.

5. Evaluation

To verify the capability of our approach to recognize the malware family with high accuracy, we use our system to run the dynamic tests on real-world malware data sets. In the following section, we show the experimental setup. Then in Section5.2, we present our evaluate result. Finally, we discuss some particular case and the limit of our system.

5.1. Experimental Setup

We implement our system on Linux, and set up the Windows XP SP3 in the modified QEMU. To evaluate the propose method, we collected 511 instances of malware from the websites and classified them into 15 different malware families by ClamAV [35], a famous antivirus tool, as show in Table1. In addition, we assume that ClamAV classified result is valid and every malware reveal its behavior in QEMU directly. Each of the family consist more than 15 instances and the most have 66 instances, we need several samples for HMM training just like speech recognition training. For the propose of evaluation, half of the malware instances in each family to be used as training data, all of the malware instances as tested data. As mentioned in proposed scheme, the **distance threshold** for clustering and the **fragment length** are the parameters that must be considered in our system. We evaluate the recognizing accuracy by adjust the two parameters, and show the result in next section.

Table2.Family list with invoked in-kernel function objects and instance number

Malware Name	Process	File	Registry	Network	Number
Adware.ZenoSearch-2	√	√	√	√	**32**
Trojan.Adload-2482	√	√	√	√	**38**
Trojan.Agent-1212		√	√		**34**

Trojan.Agent-122844		√	√		**21**
Trojan.Downloader-101635		√	√	√	**16**
Trojan.Downloader-104203		√	√		**38**
Trojan.GenericFF		√	√	√	**66**
Trojan.Lineage-286	√	√	√	√	**23**
Trojan.Ripnip-1		√	√		**68**
Trojan.Udr	√	√	√	√	**23**
W32.Philis-60	√	√	√	√	**27**
W32.Philis-138	√	√	√	√	**35**
Worm.Gavir.A	√	√	√	√	**26**
Worm.Mydoom.M	√	√	√	√	**24**
Worm.VB-1761		√	√		**40**

5.2. Result of evaluations

Figure 9 shows that the relationship between fragment length and accuracy. To observe the trend of the lines in generally, the longer length has the lower false positive but higher false negative. Because the other family malware have different behavior with the training family, there are difficult to match a long behavior sequence for the training family. Also, the lines in the Figure 10 present that setting stricter limit to cluster the log items would get the significant diversity of lower false negative and a little higher false positive. According to our result, when using small distance and suitable length, our false positive is 0%.

Table3. Profiling result and Computing average

Distance	Length	False positive	False negative
20	20	0.004753	0.084149
20	40	0	0.105675
20	60	0	0.107632
20	80	0	0.119374
20	100	0	0.123288
Distance20 Average		0.000951	0.108023
40	20	0.020268	0.060665
40	40	0.000839	0.068493
40	60	0.00028	0.080235
40	80	0.00028	0.082192
40	100	0.00028	0.088063
Distance40 Average		0.004389	0.07593
60	20	0.011183	0.058708
60	40	0.005871	0.072407
60	60	0.000559	0.09002
60	80	0.000419	0.086106
60	100	0.000419	0.088063
Distance60 Average		0.00369	0.079061
80	20	0.011602	0.058708
80	40	0.005731	0.074364
80	60	0.010484	0.078278
80	80	0.00615	0.076321
80	100	0.00629	0.076321
Distance80 Average		0.008051	0.072798

Distance	Length	False positive	False negative
20	20	0.004753	0.084149
40	20	0.020268	0.060665
60	20	0.011183	0.058708
80	20	0.011602	0.058708
Length20 Average		0.011951	0.065558
20	40	0	0.105675
40	40	0.000839	0.068493
60	40	0.005871	0.072407
80	40	0.005731	0.074364
Length40 Average		0.00311	0.080235
20	60	0	0.107632
40	60	0.00028	0.080235
60	60	0.000559	0.09002
80	60	0.010484	0.078278
Length60 Average		0.002831	0.089041
20	80	0	0.119374
40	80	0.00028	0.082192
60	80	0.000419	0.086106
80	80	0.00615	0.076321
Length80 Average		0.001712	0.090998
20	100	0	0.123288
40	100	0.00028	0.088063
60	100	0.000419	0.088063
80	100	0.00629	0.076321
Length100 Average		0.001747	0.077221

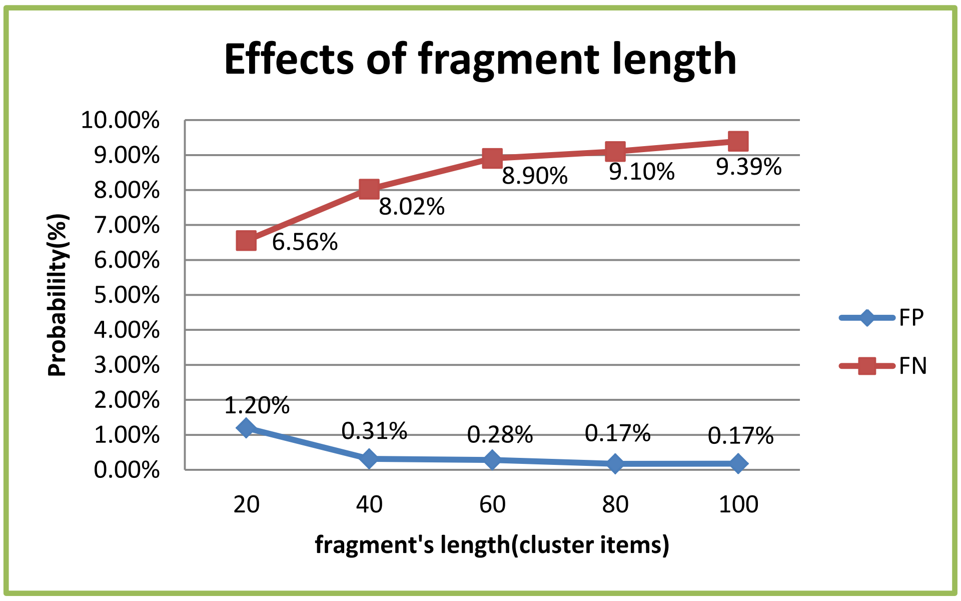


Figure12.Effects of fragment length

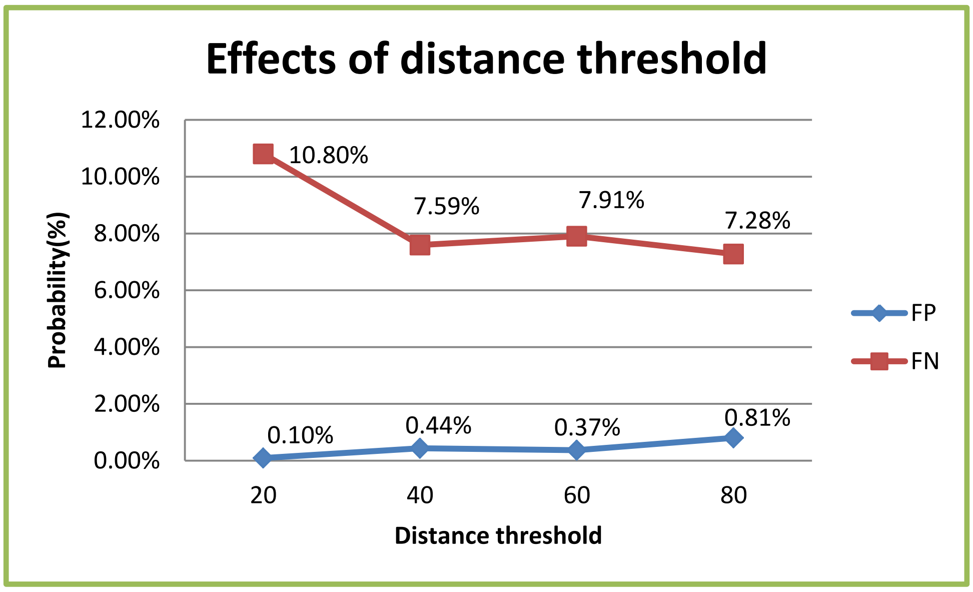


Figure13. Effects of Distance Threshold

5.3. Discussion

According to figure12, the larger fragment length has the lower false positive. A long fragment corresponds to a long term behavior observation, it's difficult to match the behavior pattern, and therefore, other family's malware is hard to be recognized to the family. Figure13 shows that the shorter distance has the lower false positive. The shorter distance means the two log items must more similar that could be clustered together. For malware not in the family, its log items are easier to assign to the null state and harder to match this hmm.

We describe the two possible outcomes in detail. First, the tested malware has not match any family pattern, namely, it is a new malware for our training system. Collecting these several malware together to cluster later, if any cluster has more than 15 malware instances then we create a new family for this cluster. Otherwise, the tested malware match one or more malware behavior patterns. Its many families' member just like a people have many identities.

Our system's every malware instances must reveal its behavior in QEMU directly. Otherwise, the instances such as VM-Aware malware would not display their malicious behavior when it detect that it is running in a virtualized environment.

6. Conclusion

Recognizing malware by family makes malware signature generation easier. In addition, it helps us distinguish malware newly emerge from those which existed already, and therefore a more efficient malware analysis becomes possible. In this dissertation, a novel approach is proposed to generate one behavior pattern for a family of malware. The proposed method relies on executing malware inside an emulated environment. Through monitoring invocations of in-kernel function and checking argument taintness at VMM-level, our scheme assures that malware cannot circumvent the monitoring mechanism. Moreover, the use of whole-system tainting guarantees that cross-process and kernel-space invading, which were not covered by previous work, are tracked. These features broaden the scope of our system while concentrating on information directly "contaminated" by the program being examined. Behaviors of the same malware family shall be then converted to an HMM-based pattern. According to our evaluation, our system produces no false positives and < 5.8% false negatives when matching 511 samples against 15 families.

7. Reference

[1] G Data Malware Report Half-year report January-June 2010. http://www.gdatasoftware.co.uk/uploads/media/GData_MalwareReport_2010_1_6_EN.pdf

[2] D. Perry. Here comes the flood or end of the pattern file. In Virus Bulletin, Ottawa,2008.

[3] E. Stinson, and J. C. Mitchell, "Characterizing Bots' Remote Control Behavior," in Proceedings of the 4th international conference on Detection of Intrusions and Malware and Vulnerability (DIMVA), Lucerne, Switzerland, July 12-13, 2007.

[4] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A Tool for Analyzing Malware," in Proceedings of the 15th European Institute for Computer Antivirus Research Annual Conference (EICAR), April 2006.

[5] J. Dai, R. Guha, and J. Lee, "Efficient Virus Detection Using Dynamic Instruction Sequences," Journal of computers 4(5), 405–414, May 2009.

[6] ThreatExpert http://www.threatexpert.com/

[7] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic spyware analysis," in Proceedings of the USENIX Annual Technical Conference (ATC), June 2007.

[8] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis" in Proceedings of the 2007 IEEE Symposium on Security and Privacy, p.231-245, May 20-23, 2007.

[9] H. Yin, Z. Liang, and D. Song, "HookFinder: Identifying and understanding malware hooking behaviors," in Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS), February 2008.

[10] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in Proceedings of the 14th ACM conference on Computer and communications security (CCS), 2007.

[11] D. Bilar, "Fingerprinting malicious code through statistical opcode analysis," in Proceedings of the 3rd International Conference on Global E-Security (ICGeS), London, UK, April 2007.

[12] R. Tian, L. M. Batten, S. C. Versteeg, "Function Length as a Tool for Malware Classification," in Proceedings of the 3rd International Conference on Malicious and Unwanted Software (MALWARE), pp.69-76, 2008.

[13] R. Islam, R. Tian , L. Batten, and S. Versteeg, "Classification of Malware Based on String and Function Feature Selection," in Proceedings of the 2010 Cybercrime and Trustworthy Computing Workshop (CTC), 19-20, July 2010.

[14] S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar, “Signature Generation and Detection of Malware Families,” in Proceedings of the 13th Australasian conference on Information Security and Privacy, Wollongong, pp. 336-349.

[15] J. Dai, R. Guha, and J. Lee, “Efficient Virus Detection Using Dynamic Instruction Sequences,” in Proceedings of the Workshop on Security and High Performance Computing Systems (SHPC), 2008.

[16] J. Dai, R. Guha, and J. Lee, “Dynamic Instruction Sequences Monitor for Virus Detection,” in Proceedings of the 4th annual workshop on Cyber Security and Information Intelligence (CSIIRW), ACM New York, NY, USA, 2008.

[17] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, “Semantics-aware malware detection,” in Proceedings of the 2005 IEEE Symposium on Security and Privacy, Oakland, May 2005.

[18] R. Perdisci, W. Lee, and N. Feamster, “Behavioral clustering of http-based malware and signature generation using malicious network traces,” in Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation (NSDI), April 2010.

[19] D. Plonka and P. Barford, “Context-Aware Clustering of DNS Query Traffic,” in Proceedings of the 8th ACM SIGCOMM conference on Internet Measurement, October 2008.

[20] G. Gu, R. Perdisci, J. Zhang, and W. Lee, “BotMiner: Clustering Analysis of Network Traffic for Protocol and Structure Independent Botnet Detection,” in Proceedings of the 17th USENIX Security Symposium, July 2008.

[21] Y. Park, and D. Reeves, “Deriving Common Malware Behavior through Graph Clustering,” in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (CCS), New York, NY, USA, 2011.

[22] M. Christodorescu, S. Jha, and C. Kruegel, “Mining specifications of malicious behavior,” in Proceedings of the ACM SIGSOFT symposium on The foundations of software engineering (FSE), pages 5–14, New York, NY, USA, 2007.

[23] X. Wang, W. Yu, A. Champion, X. Fu, and D. Xuan, “Detecting Worms via Mining Dynamic Program Execution,” in Proceedings of the Third International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm, pages412-421, Nice, 2007

[24] X. Hu, T. C. Chiueh, and K. G. Shin. “Large-scale malware indexing using function-call graphs,” in Proceedings of the 16th ACM conference on Computer and communications security (CCS), pages 611–620, Chicago, Illinois, USA, 2009.

[25] C. Willems, T. Holz, and F. Freiling, “Toward Automated Dynamic Malware Analysis Using CWSandbox,” IEEE Security and Privacy, 5(2), 2007

[26] F. Maggi, M. Matteucci, and S. Zanero, “Detecting intrusions through system call sequence and argument analysis,” IEEE Transactions on Dependable and Secure

Computing (TDSC), October 2010.

[27] U. Bayer, P. M. Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda, “Scalable, Behavior-Based Malware Clustering,” in Proceedings of the 16th Symposium on Network and Distributed System Security (NDSS), 2009.

[28] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel, “Fast malware classification by automated behavioral graph matching,” in Proceedings of the Workshop on Cyber Security and Information Intelligence Research (CSIIRW), 2010.

[29] G. Jacob, H. Debar, and E. Filiol, “Malware detection using attribute-automata to parse abstract behavioral descriptions,” CoRR, abs/0902.0322, 2009.

[30] C. Krugel, E. Kirda, D. Mutz, W. K. Robertson, and G. Vigna, “Polymorphic worm detection using structural information of executables,” In RAID, pages 207–226, 2005.

[31] F. Bellard, “Qemu, a Fast and Portable Dynamic Translator,” in Usenix Annual Technical Conference (ATC), 2005.

[32] C. W. Wang, and S. Shieh, “SWIFT: Decoupling System-Wide Information Flow Tracking for Malware Analysis”, 2011

[33] Hidden Markov model http://en.wikipedia.org/wiki/Hidden_Markov_model

[34] Jahmm - An HMM library http://www.run.montefiore.ulg.ac.be/~francois/software/jahmm/jahmmViz/

[35] ClamAV http://www.clamav.net/lang/en/

[36] Q. Zhang and D. S. Reeves, “Metaaware: Identifying metamorphic malware,” in Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC), pages 411--420, 2007.