國立交通大學

網路工程研究所

碩 士 論 文

對混淆後之殭屍網路及惡意軟體自動化分析與分類

Automatic Analysis and Classification of Obfuscated Bots and Malware Binaries

研 究 生：江易達

指導教授：林盈達　教授

中 華 民 國 九 十 九 年 六 月

對混淆後之殭屍網路及惡意軟體自動化分析與分類

Automatic Analysis and Classification of Obfuscated Bots and Malware Binaries

研 究 生：江易達　　Student：Yi-Ta Chiang

指導教授：林盈達　　Advisor：Ying-Dar Lin

國 立 交 通 大 學

網 路 工 程 研 究 所

碩 士 論 文

A Thesis
Submitted to Institutes 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

June 2010

Hsinchu, Taiwan

中華民國九十九年六月

對混淆後之殭屍網路及惡意軟體自動化分析與分類

學生：江易達　　指導教授：林盈達

國立交通大學網路工程研究所碩士班

摘　　要

在網際網路中，殭屍網路是一個很嚴重的威脅。為了要偵測殭屍網路，我們需要一個有效率的方法來分析他的行為。然而殭屍可以用混淆程式，輕易的改變其二進位程式碼，因此重複分析同種類的程式會浪費許多時間。目前已有人提出分類演算法來解決此問題，但這些方法大都不能正確分類混淆後的程式。因此我們提出一套方法來正確的分類。首先收集其呼叫之系統函數序列，之後依據此序列計算最常共同子字串及間隔分布計算相似度。同時利用片段辨識的方法增加辨識率。實驗顯示在分別不同樣本時，可以達到 94% 的正確率，而對同一種樣本偽裝後，也有 90%能正確辨識為同一種樣本。

關鍵字：殭屍網路、系統函數、最長共同子字串演算法

Automatic Analysis and Classification of Obfuscated Bots and Malware Binaries

Student: Yi-Ta Chiang Advisor: Dr. Ying-Dar Lin

Institutes of Network Engineering
National Chiao Tung University

Abstract

Botnet is a serious threat on the Internet. In order to find a way to defect botnet, we need an efficient method to analysis its behavior. However, bots can easily transform its binary code by obfuscation, and waste the time to analysis many different bots obfuscated from the same origin. Some classifying algorithms are proposed to solve this problem, but many of them cannot classify obfuscated bots well. We propose a method to classify them. First we collect the system call sequence of malware, then we calculating LCS and Gap shift distribution to decide the similarity of two samples. We also use Segment identification for improving the correctness. Experiment shows our algorithm can achieve 94% correctness rate on distinguish different samples, and 90% correctness rate on identifying class of bot variants.

Keywords: Botnet, System Call, LCS Algorithm

Contents

List of Figures

List of Tables

Chapter 1 Introduction

The Internet faces many security threats ranging from low-level attacks such as IP packet spoofing to high-level malicious activities such as Botnet. Botnet is an autonomous network that consists of compromised computers running software agents, commonly referred as robots or bots, under the control of an attacker. A bot-network is typically formed to conduct nefarious activities such as DDoS attack, e-mail spamming, stealing of information, and etc. These attacks cause concerns on the use of Internet and may lead to financial losses, so the detection and removal of Botnet is an urgent issue that we need to solve.

Botnet Detection

For the detection of botnet, one approach is to look at the aggregated traffic on a subnet and identify traffic patterns that characterize a botnet[1-4]. This is based on the assumption that bot agents of a same type usually bear similar traffic patterns in communicating with the command and control (C&C) thereof. The attack traffic (e.g. sending e-mail spam) from these bot agents shall also be similar if they follow the same design. As a result, one can identify the existence of a botnet by observing repeated patterns of suspicious traffic, which is indeed an important approach in botnet detection. However, network traffic analysis faces challenges from countermeasures such as traffic randomization, protocol obfuscation, the use of step stones, etc. Besides, it is not always possible to gain access to the traffic for the whole network due to privacy concern. In addition, the amount of the collected traffic can also place a huge burden on the analysis.

Analysis of Bot Agent Binaries

The other approach for detecting botnet is to identify the bot agent binaries. One popular method is static binary analysis[5-8], which is widely employed in most anti-virus scanners. Here, static analysis can be as simple as comparing the hash values of the binaries or as complex as scanning for suspicious instruction sequences in a binary. For bot agents with well-known signatures, static analysis can be an efficient and effective way for their detection. However, obfuscation tools can easily fool static analysis. For example, ASProtect[9], Themida[10] and UPX[11] can replace the instructions of a binary to prevent any static analysis.

For identifying bot agent binaries, one can also rely on dynamic analysis of binary[12-17], namely executing a binary and monitoring for suspicious activities such as calling privileged APIs or modifying system registry. Dynamic analysis can capture the runtime behavior of a bot agent even in the case when it is obfuscated. This is because obfuscation tool cannot rip or disrupt the runtime behavior without affecting the original functions of the obfuscated binary (the bot agent). However, dynamic analysis cannot guarantee that every possible execution path in a binary will be fully explored, and behaviors buried in those untaken paths will be ignored by the analysis.

It is common to see variants of bot agents. In fact, the source code of many popular bot agents can be found and downloaded from the Internet. This fuels the creation of even more bot variants by people who are not skillful enough to write a bot agent from scratch. To deal with bot variants, a common practice is the use of classification techniques to group together bot binaries that bear similar characteristics / features. This not only helps identify a new variant quickly, by knowing which characteristics one should look for, but also leads to more effective countermeasures

in responding to bot variants.

Classification of Bot Binaries

To address the problem of obfuscated bot agents and bot variants, we propose a new similarity-matching algorithm based on longest common subsequence (LCS). This method contains two steps: (1) system call sequence extraction by dynamic analysis; (2) system call sequence similarity metric using longest common subsequence. Because most bots still rely on system calls to interact with the underlying operating system in order to maintain binary portability and achieve pervasive infection, we believe the observed system call sequence will serve as a good feature for the follow-on classification process that is based on longest common subsequence similarity matching. Even though obfuscation tools can easily transform a program into different appearances for avoiding static analysis, they cannot hide the system calls triggered by the program. Therefore system call trace with dynamic analysis is a good method to classify unknown malwares. Experiment shows our proposed algorithm can get higher correctness rate on obfuscated bot binaries.

The rest of this work is organized as follows. Chapter 2 mentions the evolution of Botnet and the analysis methods. Chapter 3 details the proposed similarity algorithm along with techniques to improve the classification accuracy. In chapter 4, we present an implementation based on our proposed algorithms. Chapter 5 describes the experiment environment, result, and discussion. Finally, chapter 6 concludes this work with some future research directions.

Chapter 2 Background

Botnet pulls bots together to initiate large-scale attacks. In this section, we give a

brief overview of the life cycle of botnet, including the injection of bot agents into victim computers, establishing connection to C&C server, and the attack stage. We also discuss related works on analysis and classification of bot binaries.

2.1 Taxonomy of Botnet

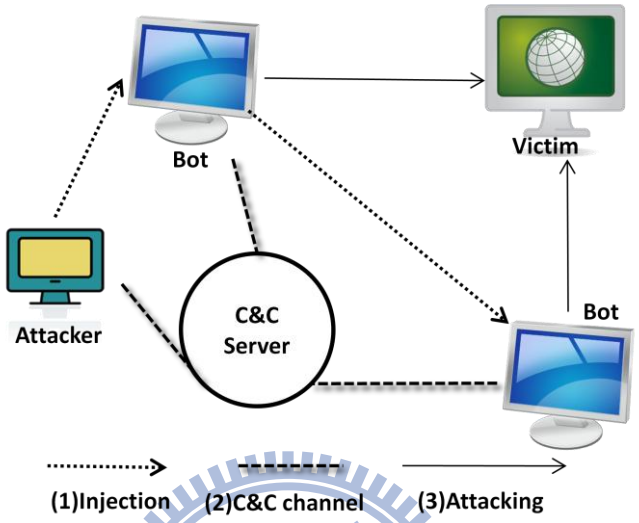

Fig. 1 Botnet Lifecycle

A botnet typically consists of three operations as shown in Fig. 1: (1) injection, (2) establishing connections to C&C server and waiting for commands, and (3) launching attacks. During injection, bots are injected to computers on the Internet. The injection of a bot into a target computer can be achieved via exploiting a remote vulnerability of the computer, disguising the bot as a harmless e-mail attachment, including the bot in some software package that is likely to be downloaded to the target computer via P2P file sharing, and etc. After a bot is injected into a computer, it can soon start seeking for other vulnerable computers and infect those computers as well. In this case, the growth rate of a botnet is exponential.

After a bot injects itself into a computer, it will attempt to establish a channel with the C&C server, which is often an IRC server. The attacker (bot herder) can then issues commands to or receives messages from the bots via the C&C. Fig. 1 shows how a botnet is formed from the C&C servers and the bots. To hide the location of the

bot herder, proxies (or step stones) can be used in-between the bot herder and the C&C server. Sometimes, a botnet will have more than one C&C servers. This prevents the botnet from failing in case the single C&C crashes or powers off.

Once a botnet is formed, depending on the types of bots in use, different types of attacks such as e-mail spam, DDoS attack, click fraud, etc can be carried out by the bots on the botnet collaboratively. Unlike a traditional network attack, in which attack traffic originates from only a few hosts, botnet-induced attacks can involve attack traffic from thousands of sources, and that makes tracking and blocking the attack traffic much more difficult. Furthermore, the attack traffic from a botnet can be huge when the participating bots all launch attacks around the same time. For instance, the botnet formed by MyDoom[18] employs 160,000 computers to generate DDoS traffic targeting the web site of SCO.

2.2 Overview of Bot Analysis

As botnet is formed by bots, one way to look at the botnet is by analyzing its constituent bot. By understanding the internals of a bot, we can have a clear picture of not only how the botnet is formed but also the attack vectors associated with the botnet.

For the analysis of bots, there are two different approaches: static analysis and dynamic analysis. Static analysis analyzes a bot binary without executing it. It typically involves dissembling the binary code, and then depicts its function call graph statically. E. Carrera, and G. Erdelyi[5] use graph isomorphism techniques to compare the call graphs from collected malwares, then use the comparing result to determines the similarity of collected malwares. Z. Liang, T. Wei, Y. Chen et al.[6] merge function calls into modules, these modules perform specific types of jobs, such as file modifying, registry modifying, and command handling. Q. Zhang, and D.

Reeves[7] look at specific patterns of assembly code and use the patterns to measure the similarity between malwares. The above works cannot handle obfuscated samples correctly, so S. Cesare, and Y. Xiang[8] design an unpacker that dump the original program from memory, after obfuscation tools restore it in memory for executing. Then they analyze this program for avoiding obfuscation.

Static analysis is typically very efficient. They may explore all execution paths in a malware. However, this means that binary obfuscation[19] can easily fool the static analysis by incurring additional execution paths, and shuffling and twisting the execution paths in an obfuscated malware. They can also restructure the data variables and tables in a binary to confuse the static analysis further. The experiments in 5.2 shows that some bots use obfuscation to hide themselves, and static analysis cannot identify them well.

Dynamic analysis is the most effective solution in obfuscated malware analysis. U. Bayer, C. Kruegel, and E. Kirda[12] proposed a system named "TTAnalyze", which executes a malware sample inside a virtual machine and observes behaviors like file modification, registry modification and network access from the sample. A key challenge in dynamic analysis is that only those control paths that are actually executed will be analyzed. A. Moser, C. Kruegel, and E. Kirda[14] use speculative branch prediction to overcome this challenge. M. Bailey, J. Oberheide, J. Andersen et al.[13] measure the similarity between bot samples based on the result from dynamic analysis, but they only look at high-level information like file name, connected host, and registry, which is easy to be randomized in bot samples. C. Willems, T. Holz, and F. Freiling,[15] also use a virtual machine to conduct dynamic analysis on bot samples. They also provide a public web interface to their analysis environment, where people can upload suspicious malware samples for analysis. P. Trinius, J. Gobel, T. Holz *et*

al.[16] try to use a block diagram to present the system calls of a program, this diagram can help us to distinguish malware more easily. J. Li, M. Xu, N. Zheng *et al.*[17] collect system call sequence by hardware virtualization, then try to identify function blocks based on the patterns occur mostly. Their comparison is based on the blocks, but ours is based on the system calls and the arguments.

2.3 Research Goal

We propose a framework that unifies the analysis and the classification of obfuscated malware. We rely on dynamic analysis techniques to extract system call details of an obfuscated bot binary. We consider not only the system call IDs, as seen in previous work, but also the call arguments used in each system call to improve the resolution of our analysis. For the classification, we devise a similarity metric based on the longest common subsequences from the extracted system call information. Finally, we notice that obfuscation often relocates code segments in a binary. In evaluating the similarity metric, we adopt some heuristics to compensate for these relocations, which would otherwise decrease the classification accuracy dramatically.

Chapter 3 Classification of Bot Binaries

We found that many bot binaries are obfuscated at Sec. 5.2, the classification needs to rely on a similarity metric that is not sensitive to binary obfuscation. The similarity metric we employ is based on observing the run-time behaviors of bot binaries. Specifically, it relies on comparing the sequence of invoked system calls by the bot binaries during runtime.

As system calls are crucial to the operation for bot agents, obfuscation cannot alter the invocations of the system calls that are critical to a bot agent's operation. On the other hand, according to our observation, the invoked system call sequences from

multiple runs of the same bot agent bear very little variation. As a result, comparing the invoked system call sequences can be used a reliable similarity metric for the classification of bot binaries.

This chapter details the observations of system call and the similarity algorithm. We also proposed some heuristics for improving classification accuracy.

3.1 Feature for Classification: Bot System Call Sequence During Injection Phase

We capture the IDs of system calls invoked by a bot agent during its injection phase. The sequence of the system call IDs are sorted by the capture time. The sequence is treated as a feature for the bot agent and is used in determining the similarity between two bots.

System calls invoked by a bot during the injection phase typically follows stereotyped pattern and is quite distinguishable. For instance, in the injection stage shown in Fig. 1, a bot usually hides its binary under %WINDIR%\system32 or other system folders, where users do not check routinely, and therefore a bot can have a better chance to avoid detection. Besides, a bot often modifies the auto-start entry point (ASEP) in Windows registry so that it can start automatically when the computer boots. Most of the steps in the injection stage are carried out by invoking the corresponding system calls (e.g. copying files and modifying registry entries). These steps are typical for most bot agents. On the other hand, it is very difficult for obfuscation to alter the system calls invoked by a bot as this would affect the execution of the bot.

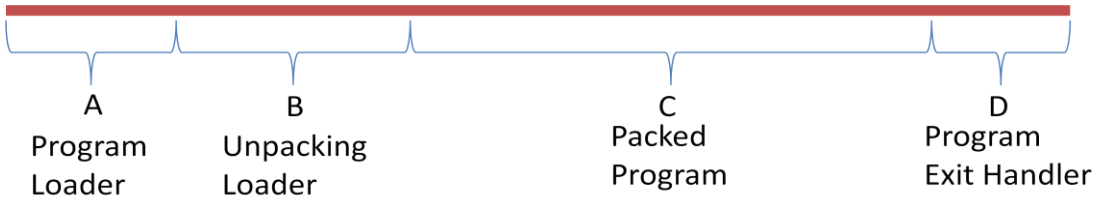

Fig. 2 Segment of System Calls in Bot

Typically, a bot's system call sequence during the injection phase can be split into the four parts as shown in Fig. 2. In part A, Windows OS loader loads necessary DLLs, allocates the memory space and etc. In Part B, unpacking loader prepares an environment to execute the original program, like unpacking the compressed binary into the text segment. In part C, system calls are made by the underlying program. In part D, there are some system calls that are used to release allocated resource and exit the program. Different obfuscation tools (packers) can introduce different system calls in part B, but in order to maintain the functionality of the original program, part C always includes system calls made by the original program.

It is very rare for a bot to employ multi-threading during the injection phase. In fact, none of the bots used in our experiment is seen to employ multi-threading during the injection phase. If multi-threading is observed, we choose the thread that has the most number of system calls and use the system calls in that thread as the feature for the bot.

3.2 LCS Similarity of System Call Sequences

For the classification of bot samples, the high level goal is to group known variants of a bot into a cluster (class), and when a unknown bot sample arrives, we can accurately identify the class it belongs to or, if no such class is found, confirm the sample as a whole new bot.

There are two ways to create a variant of a bot. The first is to use binary obfuscation to create a differently looking bot binary. As obfuscation only adds system calls but not alters existing system calls in the bot binary, the system call sequence can be used reliably as a feature for the classification of bot variants created by obfuscation.

The other way to create a bot variant is to take the source code of an existing (ancestor) bot and modify the code to create the variant. Typically, the modification does not dramatically change the code structure of the source. Most of the time, only slight changes such as adding a new injection method, adding couple of new control commands, and adjusting of C&C communication parameters. As a result, the system call sequences between the variant and the ancestor shall be similar. Therefore, we can also rely on system call sequence to identify class of bot variants created by source code modification.

The method in calculating the similarity between two system call sequences is based on their longest common subsequence. For two system call sequences X: $X_1, X_2, X_3, \ldots, X_m$ and Y: $Y_1, Y_2, Y_3, \ldots, Y_n$, where X_i and Y_j are the IDs of system calls, the longest common subsequence LCS(X,Y) is a subsequence of both X and Y, and the length of LCS(X,Y) is maximized.

The following formula is used to evaluate the similarity S(X,Y) between two call sequences *X* and *Y*:

$$S(X,Y) = \frac{|LCS(X,Y)|}{\min(|X|,|Y|)} \quad (1)$$

The similarity S(X,Y) is the ratio of the maximal length of the common system call sequence to the length of the shorter sequence between X and Y. Since $|LCS(X,Y)| \le \min(|X|,|Y|)$, the value of S(X,Y) is between 0 and 1, where 1 means either X is a variant of Y, or Y is a variant of X. A threshold T_S will be used to decide two samples are similar or not.Although a very short sequence is possibly a subsequence of a long sequence, we can ignore short sequence as the number of system call invoked by a bot is usually over 500.

3.3 Improve the Accuracy of Similarity Calculation

As shown in Fig. 2, obfuscation can introduce extra system calls and shift the system calls from the original program. If we simply take the system call sequence from an obfuscated bot, it is likely LCS similarity will incorrectly include those system calls introduced by the packer. Ideally, one should consider only system calls from segment C in Fig. 2 when calculating the LCS similarity, as the objective is to determine the similarity between the bots, which are packed within segment C. However, obfuscation has made accurate identification of segment C a non-trivial task. In the following, we present two heuristics that can be used to estimate the location of segment C roughly. We can then extract system calls from the estimated C segment and feed it to the LCS similarity calculation.

Gap Shift Compensation

When the LCS Similarity of two samples is high, there are another possibility that their system call sequence is interleaved. Because System call is a low-level view, if we insert many unrelated system calls between two system calls in the LCS sequence, the function will be very different, but the similarity is still high. We use the following method to solve this problem.

After obtaining the LCS $(S_1, S_2, S_3, \ldots S_l)$, we can find the original position of S_i in X and Y. Let the position in X and Y are $(p_1, p_2, p_3, \ldots, p_l)$ and $(q_1, q_2, q_3, \ldots, q_l)$, respectively. The gap shift sequence is the difference of the position $(p_1 - q_1, p_2 - q_2, p_3 - q_3, \ldots, p_l - q_l)$, if the values of this sequence are changed rapidly, it means the subsequence in X or Y is interleaved, whereas if the values are almost constant, the subsequence is almost continuous.

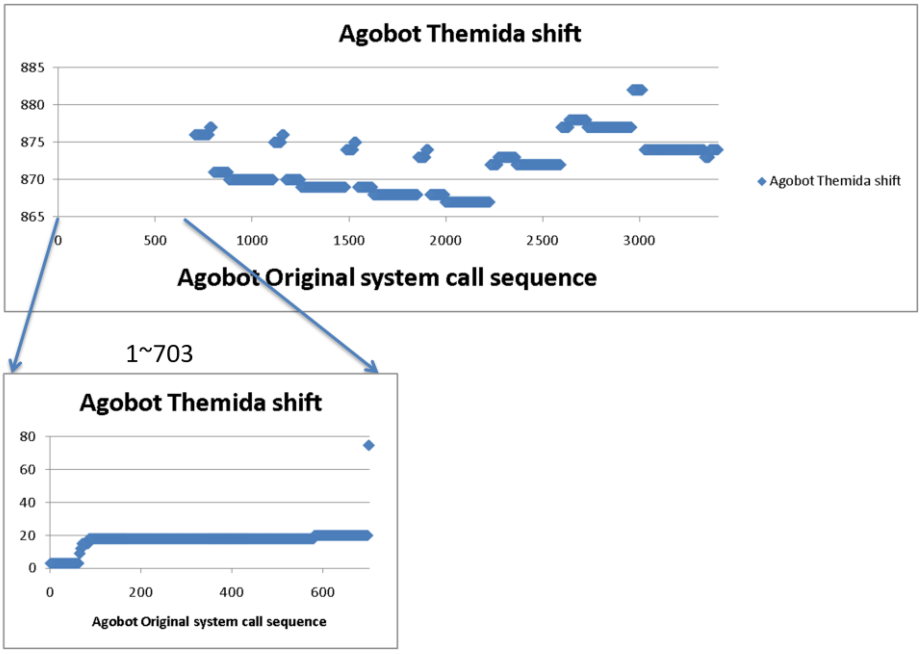

Fig. 3 The shift value chart of agobot original & agobot Themida

Fig. 3 shows the change of subsequence position for agobot original (unpacked) sequence(X) and agobot obfuscated by Themida sequence(Y). The x-axis is the index of agobot original sequence. If the system call X_i is also present at Y_j, we mark a point at $(i, j-i)$ on the chart. This figure shows almost all system calls in agobot original exist in Themida obfuscated agobot, and the shift center on 20 and 870. This means additional system calls added by obfuscation are centralized.

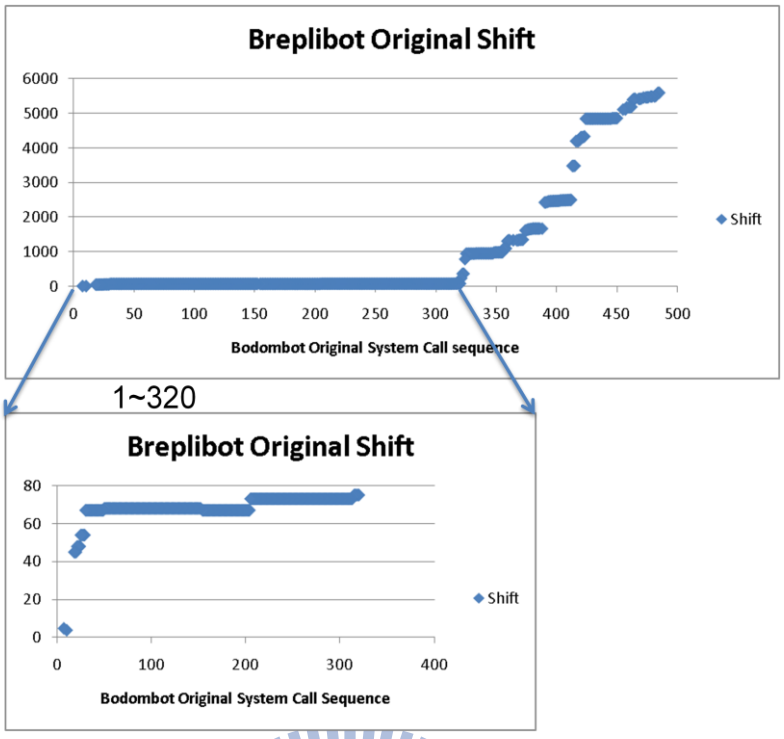

Fig. 4 The shift value chart of Bodombot original & Breplibot origional

Fig. 4 is two different bots, although almost all system calls in Bodombot also appear in Breplibot, but the shift value changes rapidly. We can use this property to conclude they are different bots.

Let N be how many unique numbers in $(p_1-q_1, p_2-q_2, p_3-q_3, \ldots, p_l-q_l)$, L is the length of subsequence, then we define the formula:

$$R = \frac{N}{L} \tag{2}$$

If R is higher than a threshold T_R, then the value in shift sequence is changed quickly, this two sample are likely different bots. With the formula (1) and (2) and the threshold, the follow criteria is used to decide two samples are similar or not:

$$\begin{cases} S \le T_S \cdots\cdots\cdots\cdots\cdots\cdots Different \\ S > T_S \ and \ R > T_R \ \cdots Different \\ S > T_S \ and \ R \le T_R \ \cdots Same \end{cases} \tag{3}$$

Segment Identification

In Fig. 2, we can see that only the central portion (segment C in Fig. 2.) of the system call sequence is of relevance for similarity computation. Fig. 3 shows this behavior clearly. At first, the shift is about 20, this is because the OS will load different resource for different program, so their segment A has a little difference. Then the shift jumps to 870, so after obfuscated by Themida, Themida add about 850 system calls before the segment of main program.

The system calls involved in other segments (A,B, and D) are common to almost all executable files. Hence we can cut those irrelevant segments and use only segment C in the similarity calculation. However, a small obstacle is that the system calls in segment B actually depend on the types of the obfuscation tools in use. So we have to build profiles for each different obfuscation tool in order to identify and remove segment B effectively from an obfuscated binary executable. For an obfuscated binary, its segment B typically consists of file and registry related system calls such as NTCreateFile, NTDeleteFile, NTOpenFile, NTCreateKey, NTOpenKey and NTQueryValueKey). The arguments to these system calls follow certain patterns depending on the obfuscation tool in use. For example, Themida always contains the follow system call (in bold) and argument in Fig. 5:

```
    NTOpenKey
\Registry\Machine\Software\Microsoft\Windows
NT\CurrentVersion\Image File Execution
Options\winmm.dll
    NTOpenKey
\Registry\Machine\Software\Microsoft\Windows
NT\CurrentVersion\DRIVERS32
    NTQueryValueKey wave
    NTQueryValueKey wave
    NTQueryValueKey wave1
    NTQueryValueKey wave2
    NTQueryValueKey wave3
    NTQueryValueKey wave4
    NTQueryValueKey wave5
    NTQueryValueKey wave6
    NTQueryValueKey wave7
    NTQueryValueKey wave8
```

Fig. 1 Part of the system calls of Themida

To build the profile, we again use LCS to identify the common subsequence across binaries obfuscated by a given packer. After running LCS over a certain number of obfuscated binaries, the common subsequence that is left will include only segment A,B, and D. Since segment A and D are due to the standard OS loader and un-initialization code, we can manually trim them away and build the profile that contains only the isolated segment B for the given packer. The profile can later be used to remove segment B from an obfuscated binary.

Chapter 4 System Implementation

We implemented a system to capture the system call sequence of a running bot program and perform classification based on the captured call sequence against a database of known bot agents. In this section, we give an overview of the system

architecture and the execution flows.

4.1 System Architecture

The system consists of 4 components: (1) Controller, which coordinates the execution and analysis environment; (2) Recorder, which extracts the system call sequence from dynamically instruments a running bot program via PIN tools[20] and captures the system calls from a running bot binary; (3) Classifier, which classifies a bot sample by identifying similar bot samples from the database; and (4) Database for storing the system call sequences of bot samples known to the system.

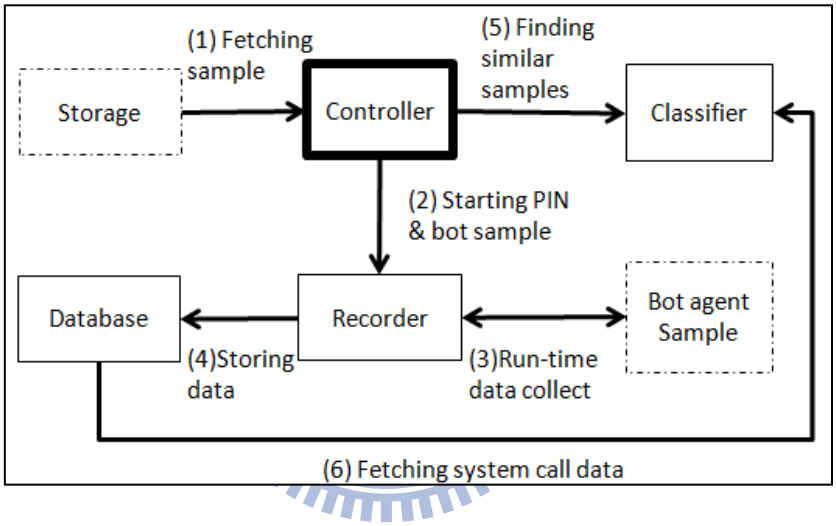

Fig. 6 Architecture Diagram

Fig. 6 shows the execution flow among the function blocks. First, the controller fetchs a bot sample from the storage. It then invokes the recorder, which executes the bot sample with PIN. The recorder records all the system calls invoked by the bot sample during its execution. The name of the bot sample and the invoked system calls are stored in the database for subsequent classification. Finally, the classifier identifies known bots from the database that are similar to the one just analyzed.

4.2 Recorder Implementation

Dynamic analysis of a binary executable can be achieved in three popular ways. One is API hooking, in which system calls are intercepted and recorded by the hook

function (the recorder). This method only handles the coarse system calls and does not provide fine-grained details such as the processor instructions executed. API hooking can be detected by inspecting the Export Address Table(EAT) and Import Address Table(IAT)[21]. Bots with anti-analysis mechanism may quit its execution early on when hooks from malware analysis tools are detected.

The other method is running the bot sample inside an emulator such as QEMU[22] and monitoring its behaviors via the emulator. Because the whole system is emulated, the monitoring can be done in fine-granularity at the instruction-level. However, some existing works[14] using this method needs to modify the source of an emulator. This limits them can only use open-sourced emulator like QEMU.

The approach we use is binary code instrumentation. It dynamically instruments the monitoring code into a running binary whenever a certain condition is reached. (e.g. a system call is about to be invoked.). This approach does not involve API-hooking and runs much faster than the emulator-based approach as most of the time the binary is executed directly on the processor hardware. After the injection stage, some bots will create another child process for waiting commands, and stop itself. We do not continue monitor the child process because the system calls in the injection stage is enough to classify.

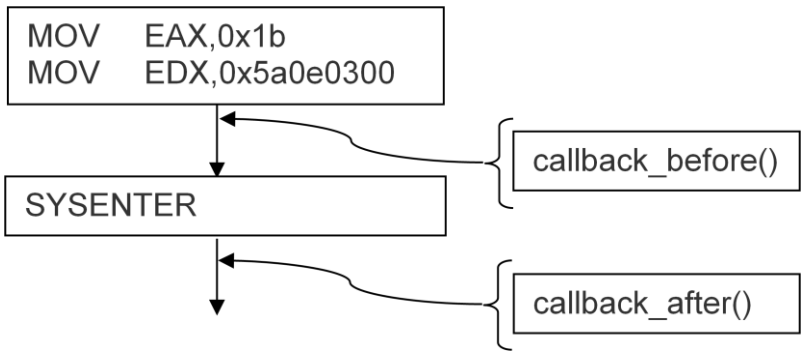

Fig. 7 How PIN records system calls

As shown in Fig. 7, when a program wants to invoke system call, it puts the

system call ID in the register EAX and the address of argument in register EDX. The program then uses SYSENTER instruction to invoke the syscall handler in the OS kernel.

The just-in-time compiler of PIN can monitor SYSENTER instruction. We register a function to PIN by PIN_AddSyscallEntryFunction() API at callback_before , so that the Recorder module can know the program will invoke a system call.

In the callback function, we use the following PIN APIs to get the information of the thread ID, system call ID, and system call arguments:

Table 1 The PIN APIs used in the recorder module

API name	Purpose
PIN_GetSyscallNumber	Get system call ID
PIN_GetTid	Get ID of the thread that invokes SYSENTER
PIN_GetSyscallArgument	Get the N^{th} variable at the argument

System call ID and thread ID can be directly stored in the database. However, the arguments need to be further processed according to the argument types (e.g. a pointer, an integer, a long integer, and etc.) Take NTQueryValueKey for example, its prototype declaration looks like:

```
NTSTATUS ZwQueryValueKey(
  __in  HANDLE KeyHandle,
  __in  PUNICODE_STRING ValueName,
  __in  KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass,
  __out_opt  PVOID KeyValueInformation,
  __in  ULONG Length,
  __out PULONG ResultLength
);
```

In this system call, some of the variables are pointers (e.g. KeyValueInformation ValueName, and ResultLength), and some are integer values (e.g. Length). We only process those arguments of interest to our analysis. For instance, in this example, we

extracts the string pointed ValueName and ignore all the other arguments. The processed arguments will be stored along with the system call ID into the database.

Chapter 5 Experimental Studies

We design three experiments to verify our proposed algorithm and system design. First we evaluate the correctness rate of grouping same bots by applying our algorithm to same bot with different obfuscation tools. Then we measure the correctness rate of distinguish different bots by applying our algorithm to different bots with the same obfuscation tool. Finally we shows the execution time with and without our recorder module.

To evaluate the correctness and efficiency of the proposed algorithm and overall system design, we implement a testbed and conduct the following two experiments: (1) 10 random bot agents are chosen and obfuscated with ASProtect, Themida and UPX. We then feed the obfuscated bot agents to our classification system and measure the classification accuracy by counting the number of correctly classified bot agents, (2) estimating the execution overhead of PIN by comparing the running time of a bot agent with and without PIN.

5.1 Experiment Environment

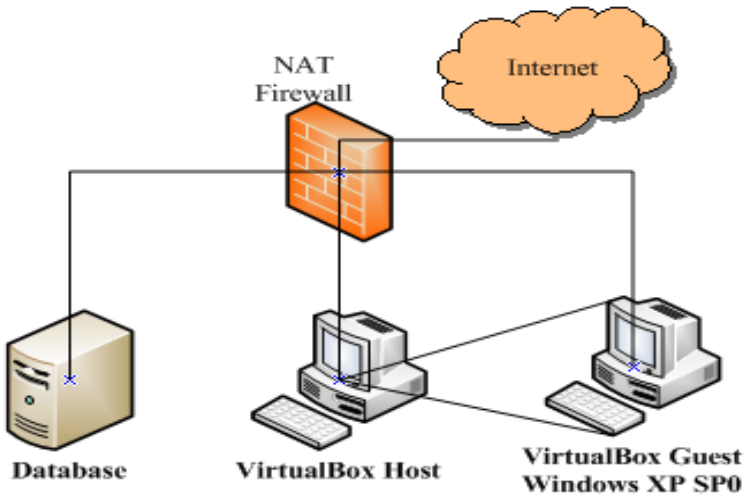

Fig. 8 Experiment environment

In the experiment environment shown in Fig. 8, a database is used for storing bot agent binary executables and their system call sequences, which are later collected by the recorder module in Sec. 4.2. The controller module loads a bot agent from the database and runs the bot in a virtual machine running Windows XP. The recorder module then collects the system call sequence from the bot agent while it is running. The virtual machine (VM) makes it easy to fall back to a clean system state before processing the next bot agent. For the virtual machine, we use VirtualBox, which is free and comes with a command line interface (CLI) for controlling virtual machine. The CLI facilitates the experiment process as the instantiation of a clean VM can be automated via scripts. Inside the virtual machine, we use the version of Windows XP prior to service pack 1 to ensure the maximal compatibility with the bot being analyzed (e.g. DEP in Windows XP might prevent some of the bots from being successfully executed.). All the machines are connected to the Internet through a NAT firewall to prevent malicious traffic from the Internet to interfere with the experiment testbed while allowing bot agents to make connections to the C&C server.

In Table 2, we presents 10 different randomly selected bot agents from 463 bot

samples. These 10 bot agents are packed with different obfuscation tools - ASProtect, Themida and UPX. This results in 40 test targets in total (10 original plus 10 from each obfuscation tool).

Table 2 List of Bots used in the experiment

Id	md5 hash	Kaspersky	Sophos
1	ea46b4606531d28474e06cb4cd060c71	Backdoor.Win32.Anibot.b	Mal/IRCBot-B
2	c1ed6261902ebc178f55159ca1b061b1	Backdoor.Win32.Afbot.a	Mal/IRCBot-C
3	d7b32cc7056f37eb8ccf0d1f472d8e5b	Backdoor.Win32.Rbot.gen	W32/Rbot-Gen
4	fa29f9048e3b57705e97583d70f00ba1	Backdoor.Win32.Agobot.gen	W32/Agobot-Gen
5	f1f9f762f899a24a2d71a35c4b825db8	Backdoor.Win32.Rohbot.a	Mal/Generic-A
6	69fd63dade7cd4f8878c6e80084069fb	Backdoor.Win32.Rbot.gen	W32/Rbot-Fam
7	4aac3724863070dc422ad0dc0a39a5af	Backdoor.IRC.Botva.b	Troj/Bckdr-MPJ
8	8a87d88714f2017e2cdd74912449e7cf	Backdoor.Win32.DevBot.b	Troj/DevBot-B
9	c3207feb5160c71227dbd92cc3fe4e53	Backdoor.Win32.DaSBot.12	Mal/Generic-A
10	0ce8ccbd76e6126ed10350fd70c37d98	Backdoor.Win32.PoeBot.a	W32/Poebot-Gen

5.2 Static Analysis Experiment

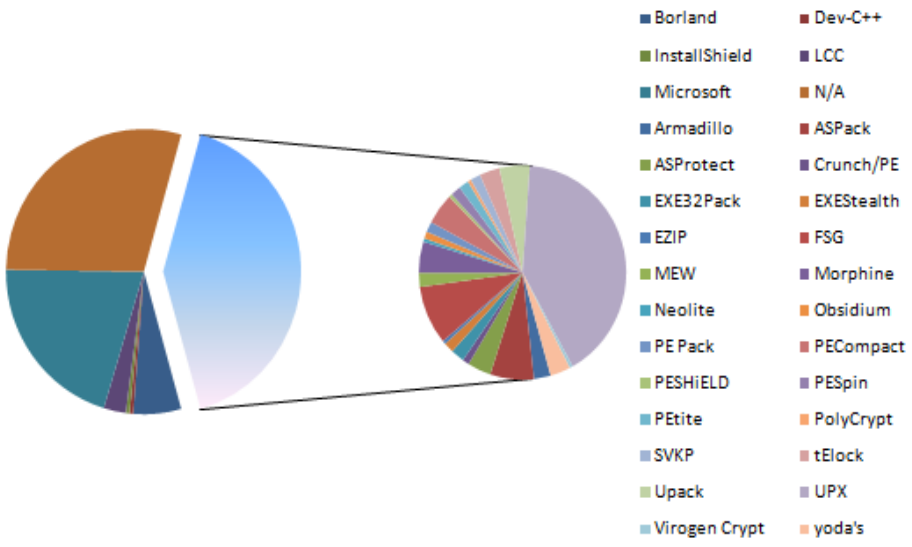

Fig. 9 PEiD analysis result

In Fig. 9, PEiD is used for analyzing 463 bot samples. PEiD is a static analysis

tool for detecting the existence of obfuscation tool. From the PEiD scan, it finds that 41.2% of the samples are obfuscated. This highlights the rampancy of obfuscated bot samples and justifies the need for better analysis and classification techniques in dealing with bots.

Table 3 Static analysis result (AV engines v.s. obfuscation tools)

ID	Anti Virus	Non-obfuscated	ASProtect	Themida	UPX
1	Antivir	TR/Dldr.Small.caf.3	**TR/Downloader.Gen**	**TR/Crypt.TPM.Gen**	TR/Dldr.Small.caf.3
	Kaspersky	Backdoor.Win32.Anibot.b	**N/D**[1]	**N/D**	Backdoor.Win32.Anibot.b
	NOD32	Win32/Genetik	Win32/Genetik	**N/D**	Win32/Genetik
	Sophos	Mal/IRCBot-B	**Sus/Behav-325**	**Mal/Behav-285**	**Mal/IRCBot-B**
2	Antivir	BDS/Backdoor.Gen	**N/D**	**TR/Crypt.TPM.Gen**	BDS/Backdoor.Gen
	Kaspersky	Backdoor.Win32.Afbot.a	**N/D**	**Packed.Win32.Black.a**	**Backdoor.Win32.Afbot.a**
	NOD32	Win32/IRCBot.OY	**Win32/IRCBot.OY**	**N/D**	Win32/IRCBot.OY
	Sophos	Mal/IRCBot-C	**Sus/Behav-325**	**Mal/Behav-285**	Mal/IRCBot-C
3	Antivir	EXP/DameWare.ggg	TR/Downloader.Gen	**TR/Crypt.TPM.Gen**	**WORM/Rbot.210944**
	Kaspersky	Backdoor.Win32.Rbot.gen	**TR/Crypt.TPM.Gen**	**N/D**	Backdoor.Win32.Rbot.gen
	NOD32	Win32/Rbot	Win32/Rbot	Win32/Rbot	Win32/Rbot
	sophos	W32/Rbot-Gen	**Sus/Behav-325**	**Mal/Behav-285**	W32/Rbot-Gen
4	Antivir	BDS/Agobot.3.200704	**N/D**	**TR/Crypt.TPM.Gen**	**BDS/Sdbot.Q.Plus**
	Kaspersky	Backdoor.Win32.Agobot.gen	**N/D**	**N/D**	Backdoor.Win32.Agobot.gen
	NOD32	Win32/Agobot	Win32/Agobot	**N/D**	Win32/Agobot
	sophos	W32/Agobot-Gen	**Sus/Behav-325**	**Mal/Behav-285**	W32/Agobot-Gen
5	Antivir	TR/Crypt.FKM.Gen	TR/Crypt.FKM.Gen	**TR/Black.Gen2**	TR/Crypt.FKM.Gen
	Kaspersky	Backdoor.Win32.Rohbot.a	**N/D**	**N/D**	Backdoor.Win32.Rohbot.a
	NOD32	unknown NewHeur_PE	**N/D**	**N/D**	unknown NewHeur_PE
	sophos	Mal/Generic-A	**Sus/Behav-1021**	**Mal/Behav-285**	Mal/Generic-A

[1] N/D means not detected

To show how obfuscation can easily fool state-of-the-art static analysis, we use Jotti's malware scan to scan the sample #1 to #5 at Table 2. Jotti integrates multiple scan engines. All of these files are scanned at May 5, 2010. Both the original unpacked bot binaries and the obfuscated versions are scanned by this scanner. From Table 3, all four anti-virus scanners correctly identify the five bots in the original forms. However, with obfuscation, both false positives and false negatives are observed, which are shown in bold text in Table 3. All four anti-virus tools can effectively decrypt UPX-packed bots because UPX simply compresses an executable without involving any obfuscation. ASProtect change the program structures substantially and fool the analysis from these anti-virus scanners quite effectively. For sample #6 to #10, they also show the same tendency that AV tools are affected by obfuscation.

5.3 LCS Similarity of Bot Variants Created by Obfuscation

This experiment shows the LCS similarities between bot variants created by obfuscating a bot sample with different packers. The test targets include the 10 bot samples without any obfuscation listed in Table 2 (denoted as group A). We obfuscate each of those 10 bot samples with ASProtect to create ASProtect-obfuscated test targets (denoted as group B). Similarly, we have Themida-obfuscated test targets (group C) and UPX-obfuscated test targets (group D).

Ideally, the similarities between two test targets from the same bot sample should be 1, which means that the two targets belong to the same class. Detailed results are shown as Table 4. In each cell, the value on the first line corresponds to the LCS similarity $S(X,Y)$ (Eq.1). Values on the second line correspond to the Gap Shift values R (Eq.2):

Table 4 Similarities between test targets derived from the same bot sample

Sample #1				Sample #2				Sample #3			
	B	C	D		B	C	D		B	C	D
A	0.95 0.01	0.97 0.01	0.99 0.01	A	1 0.03	0.98 0.04	1 0.01	A	0.99 0.01	0.95 0.01	1 0.01
B	-	0.54 0.01	0.95 0.01	B	-	0.78 0.05	1 0.01	B	-	0.99 0.02	0.99 0.01
C		-	0.97 0.01	C		-	1 0.01	C		-	0.95 0.01
Sample #4				**Sample #5**				**Sample #6**			
	B	C	D		B	C	D		B	C	D
A	0.81 0.01	0.99 0.03	0.99 0.01	A	0.96 0.01	0.98 0.01	0.99 0.01	A	0.93 0.01	0.99 0.01	0.99 0.01
B	-	0.57 0.01	0.81 0.01	B	-	0.91 0.02	0.99 0.01	B	-	0.77 0.05	0.93 0.01
C		-	0.99 0.03	C		-	0.98 0.01	C		-	0.94 0.01
Sample #7				**Sample #8**				**Sample #9**			
	B	C	D		B	C	D		B	C	D
A	0.98 0.01	0.94 0.01	0.98 0.01	A	0.98 0.01	0.98 0.01	0.99 0.01	A	0.98 0.01	0.91 0.01	0.99 0.01
B	-	0.21 0.03	0.98 0.03	B	-	0.55 0.01	0.99 0.01	B	-	0.59 0.07	0.98 0.01
C		-	0.94 0.03	C		-	0.98 0.01	C		-	0.91 0.01
Sample #10											
	B	C	D								
A	0.78 0.04	1 0.01	1 0.01								
B	-	0.35 0.04	0.78 0.04								
C		-	1 0.01								

Table 4 shows that on the same bots, the similarity values are consistently high and *R* values are consistently low. Segment Identification can improve the accuracy of LCS similarity greatly. For instance, if we turn off segment identification, the LCS similarity between B and C in Sample #5 will drop from 0.91 to 0.79.

5.4 LCS Similarity across Different Bot Samples

This experiment evaluates the LCS similarities across 10 different bot samples. It shows the range of *S* and *N* values is different than Sec.5.3. This is conducted on four batches of experiments. First, we calculate the pair-wise LCS similarities from the 10 bot samples. The result is presented in Table 5A. We then calculate the pair-wise LCS

similarities on ASProtect-obfuscated bot samples with the result shown in Table 5B. The results with Themida-obfuscated bot samples and UPX-obfuscated bot samples are presented in Table 5C and Table 5D respectively.

Table 5 Numeric result on the same obfuscation samples

A. Non-obfuscated bots

	2	3	4	5	6	7	8	9	10
1	0.61 0.33	0.58 0.18	0.74 0.17	0.68 0.61	0.40 0.16	0.31 0.30	0.65 0.18	0.36 0.27	0.46 0.5
2	-	0.75 0.08	0.91 0.32	0.77 0.63	0.69 0.27	0.24 0.31	0.40 0.17	0.56 0.25	0.93 0.17
3	-	-	0.91 0.16	0.74 0.71	0.62 0.17	0.25 0.16	0.43 0.02	0.45 0.26	0.89 0.06
4	-	-	-	0.65 0.63	0.94 0.14	0.83 0.10	0.95 0.13	0.72 0.23	0.60 0.20
5	-	-	-	-	0.69 0.62	0.64 0.54	0.65 0.60	0.72 0.54	0.56 0.34
6	-	-	-	-	-	0.41 0.13	0.65 0.20	0.83 0.02	0.86 0.21
7	-	-	-	-	-	-	0.23 0.22	0.27 0.27	0.77 0.12
8	-	-	-	-	-	-	-	0.50 0.27	0.81 0.26
9	-	-	-	-	-	-	-	-	0.20 0.28

B. ASProtect obfuscated bots

	2	3	4	5	6	7	8	9	10
1	0.47 0.29	0.70 0.11	0.77 0.05	0.94 0.06	0.58 0.08	0.50 0.13	0.76 0.11	0.56 0.13	0.80 0.09
2	-	0.62 0.09	0.51 0.25	0.34 0.38	0.48 0.27	0.23 0.27	0.39 0.18	0.43 0.24	0.32 0.25
3	-	-	0.92 0.03	0.92 0.1	0.68 0.14	0.42 0.08	0.94 0.01	0.58 0.16	0.79 0.10
4	-	-	-	0.90 0.10	0.77 0.04	0.92 0.04	0.89 0.08	0.76 0.04	0.22 0.34
5	-	-	-	-	0.94 0.07	0.90 0.07	0.94 0.11	0.95 0.08	0.28 0.17
6	-	-	-	-	-	0.50 0.08	0.70 0.14	0.90 0.02	0.17 0.33
7	-	-	-	-	-	-	0.36 0.12	0.40 0.04	0.14 0.32
8	-	-	-	-	-	-	-	0.61 0.16	0.47 0.23
9	-	-	-	-	-	-	-	-	0.20 0.28

C. Themida obfuscated bots

	2	3	4	5	6	7	8	9	10
1	0.41 0.16	0.67 0.16	0.76 0.17	0.98 0.01	0.53 0.11	0.43 0.17	0.72 0.17	0.49 0.23	0.88 0.22
2	-	0.75	0.68	0.77	0.68	0.24	0.40	0.56	0.27

		0.08	0.22	0.63	0.27	0.31	0.17	0.25	0.25
3	-	-	0.90 0.07	0.74 0.71	0.62 0.17	0.25 0.16	0.93 0.02	0.45 0.26	0.26 0.15
4	-	-	-	0.72 0.13	0.91 0.1	0.87 0.03	0.88 0.07	0.76 0.24	0.76 0.16
5	-	-	-	-	0.41 0.13	0.65 0.20	0.75 0.60	0.83 0.02	0.20 0.38
6	-	-	-	-	-	0.41 0.13	0.65 0.20	0.83 0.02	0.20 0.38
7	-	-	-	-	-	-	0.23 0.22	0.27 0.27	0.10 0.52
8	-	-	-	-	-	-	-	0.50 0.27	0.47 0.23
9	-	-	-	-	-	-	-	-	0.20 0.33

D. UPX obfuscated bots

	2	3	4	5	6	7	8	9	10
1	0.61 0.33	0.58 0.18	0.46 0.5	0.68 0.61	0.40 0.16	0.31 0.30	0.65 0.18	0.36 0.27	0.28 0.18
2	-	0.75 0.08	0.93 0.17	0.77 0.63	0.69 0.27	0.24 0.31	0.40 0.17	0.56 0.25	0.27 0.25
3	-	-	0.89 0.06	0.74 0.71	0.62 0.17	0.25 0.16	0.43 0.02	0.45 0.26	0.27 0.15
4	-	-	-	0.57 0.31	0.86 0.2	0.78 0.10	0.51 0.25	0.86 0.21	0.35 0.42
5	-	-	-	-	0.69 0.62	0.64 0.54	0.65 0.60	0.72 0.54	0.57 0.61
6	-	-	-	-	-	0.41 0.13	0.65 0.20	0.83 0.02	0.20 0.38
7	-	-	-	-	-	-	0.23 0.22	0.27 0.27	0.10 0.52
8	-	-	-	-	-	-	-	0.50 0.27	0.47 0.23
9	-	-	-	-	-	-	-	-	0.20 0.33

This result shows the LCS similarities between different samples are indeed lower than the LCS similarities between variants of the same sample (give reference to the previous section). Some of the pairs have high LCS similarity values. However, their Gap Shift values (R) are also high (see Sample 2 v.s. 4 in Table 5A). Therefore the formula (3) is a good criteria for deciding whether two sample are similar.

5.5 Choosing the threshold values for *S* and *R*

In order to determining the threshold values T_S and T_R, we plot the curve of classification correctness vs. T_S and also the curve of classification correctness vs. T_R.

For the classification correctness, we consider both the correctness on identifying

class of bot variants (#_of_pairs_above_T_S_in_Table 4 / #_of_pairs_in_Table 4), and the correctness on distinguishing different samples (#_of_pairs_below_T_S_in Table 5/ #_of_pairs_in_Table 5).

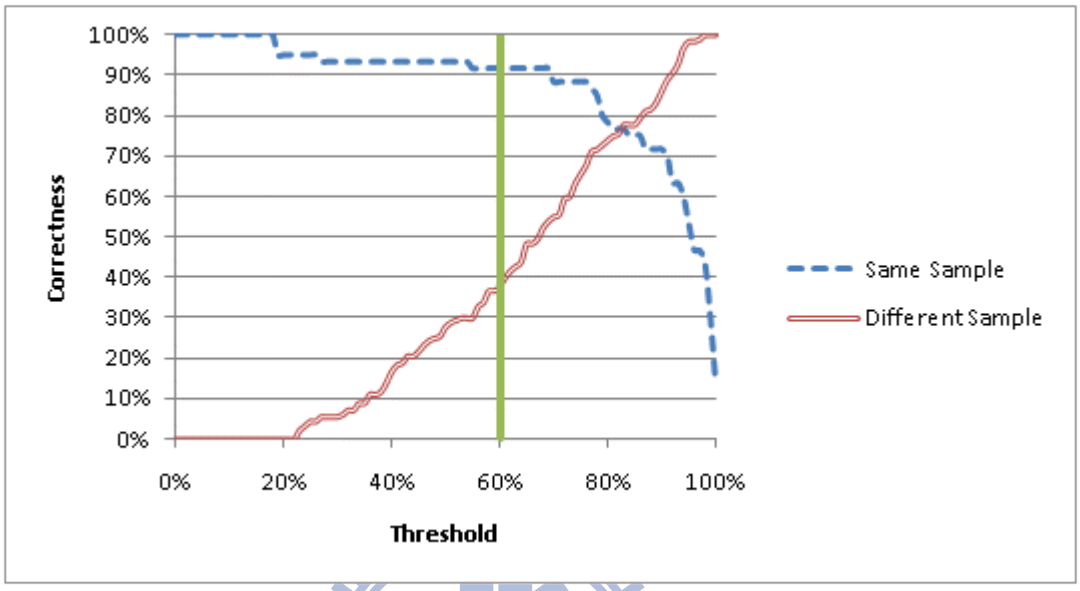

Fig. 10 LCS correctness threshold

We consider 60% as an appropriate threshold value for LCS similarity S because for identifying class of bot variants created by obfuscating a source bot sample with different packer tools, this achieves 90% correctness (i.e. for about 6 out of the 60 pairs of the bot variants used in Table 4, the pair-wise LCS similarity is incorrectly reported to be below 60%).

For distinguishing different bot samples, the classification correctness (distinguishing them as unrelated bots) rises linearly when increasing the threshold value for S. With the threshold value of 60%, the classification correctness on distinguishing different bot samples is only about 40%.

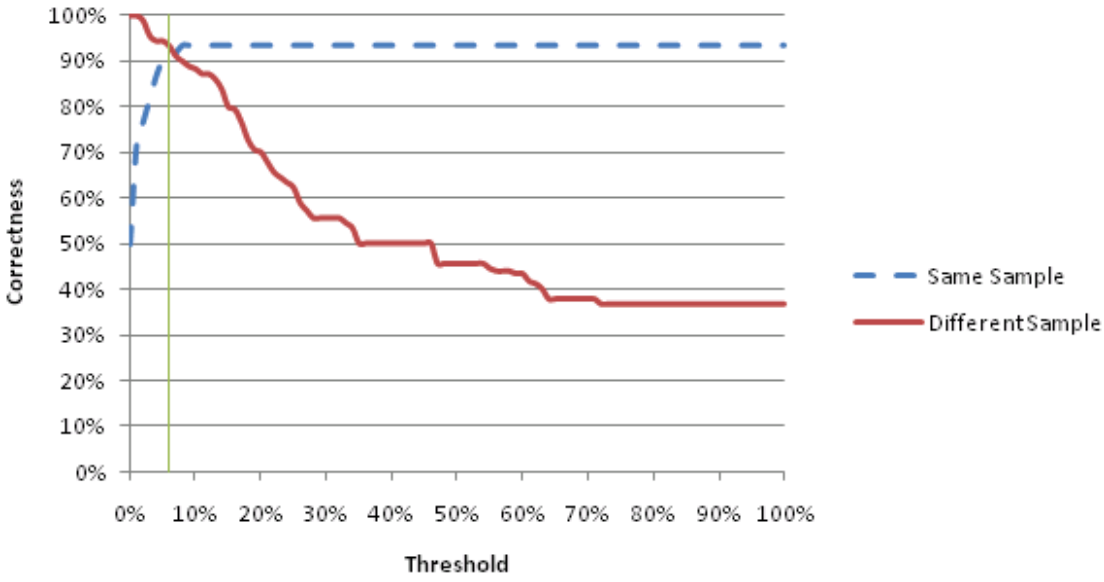

Fig. 11 Gap Shift correctness threshold

To improve the correctness on distinguishing different bot samples, we need to consider the gap shift value R as well (Eq. 2). As shown in Fig. 11, if we choose 6% as the threshold value T_R for R, we can correctly distinguish different samples with 94% probability while maintaining the correctness on identifying class of bot variants at 90%.

5.6 Classification Result Comparison

Based on the threshold experiments in previous section, the samples in Table 2 can be classified into groups by our algorithms. Table 6 is the comparison of the number of samples that classify correctly based on our algorithms and Anti-Virus Engines. It shows our algorithms can classify more correctly.

Table 6 The rate of classified correctly in total samples

Our algorithms	**Antivir**	**Kaspersky**	**NOD32**	**Sophos**
90%	68%	70%	80%	48%

5.7 Efficiency Experiment

As mentioned before, PIN Tool uses JIT compiler to instrument the target program with check code dynamically. In this experiment, we want to observe the

overhead introduced by the instrumentation and the recorder module.

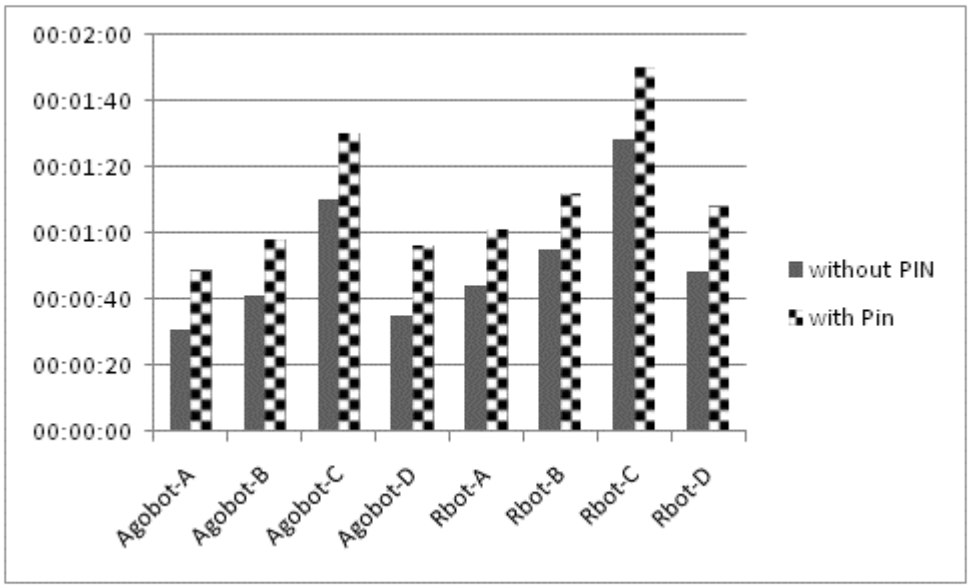

Fig. 12 Execution time with and without PIN

Fig. 12 shows the execution times for running bot samples directly (without PIN) and with our recorder module (together with PIN). We use two samples (Agobot and Rbot) for this experiment. We also examine the execution time for running the obfuscated versions of the bots. We employ three obfuscation tools (B: ASProtet, C: Themida, D:UPX, and A is without obfuscation). Overall, the overhead from using our system is around 55%.

Chapter 6 Conclusions and Future Works

Our work aims at the classification of bot binaries, many of which are obfuscated and immune to static analysis. We use a dynamic analysis tool called PIN to record the system call sequence of a running bot. Because system call sequence defines the interactions between a program (the bot binary) and the operating system, obfuscation can hardly alter the sequence without breaking the interactions. We rely on this invariant property of invoked system call sequence in obfuscated bot binaries as the

basis of our classification framework.

For the classification, we define a similarity metric between two bots based on the longest common subsequence of their system call sequences. Although obfuscation can hardly change the original system call sequence in a bot binary, it does introduce additional system calls into the obfuscated binary. Most of them are due to the obfuscation packer's stub code. The additional system calls can result in noises in the classification process. To reduce the noise, we inspect the system call arguments to identify those system calls introduced by the packer and have them filtered. Overall, our system can achieves classification correctness of more than 90%.

While some obfuscated programs have additional threads for detecting the existence of debugger, no bot has been seen to use multi-thread at injection stage. If multi-threading at the injection stage is observed, our system will take the system call sequence in the main thread (the thread with the most system calls) for the analysis and classification. In future work, the similarity metric can be extended to consider all the system call sequences in a multi-threaded bot program.

The current system is based on an off-line process. It records the system call sequence and then compares the sequence with sequences of known samples in the database. In future work, we will attempt with an on-line analysis process, where the system can work as an anti-virus tool that detects active running bots on a computer.

Reference

[1] Y. Xie, V. Sekar, D. Maltz *et al.*, “Worm Origin Identification Using Random Moonwalks,” in IEEE Symposium on Security and Privacy, 2005.

[2] M. Collins, and M. Reiter, “Hit-list worm detection and bot identification in large networks using protocol graphs,” *Lecture Notes in Computer Science*, vol. 4637, pp. 276-295, 2007.

[3] G. Gu, J. Zhang, and W. Lee, “BotSniffer: Detecting botnet command and control channels in network traffic,” in The 15th Annual Network & Distributed System Security Symposium, 2008.

[4] T. Yen, and M. Reiter, “Traffic aggregation for malware detection,” *Lecture Notes in Computer Science*, vol. 5137, pp. 207-227, 2008.

[5] E. Carrera, and G. Erdelyi, “Digital genome mapping - advanced binary malware analysis,” in Virus Bulletin Conference, 2004.

[6] Z. Liang, T. Wei, Y. Chen *et al.*, “Component Similarity Based Methods for Automatic Analysis of Malicious Executables,” in Virus Bulletin Conference, 2007.

[7] Q. Zhang, and D. Reeves, “Metaaware: Identifying metamorphic malware,” in Twenty-Third Annual Computer Security Applications Conference, 2007.

[8] S. Cesare, and Y. Xiang, “A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost,” in 24th IEEE International Conference on Advanced Information Networking and Applications, 2010.

[9] "ASPACK SOFTWARE - Best Choice Compression and Protection Tools for Software Developers," [online], available from World Wide Web; http://www.aspack.com/asprotect.aspx.

[10] "Oreans Technology : Software Security Defined.," [online], available from World Wide Web; http://www.oreans.com/themida.php.

[11] "UPX: the Ultimate Packer for eXecutables - Homepage," [online], available from World Wide Web; http://upx.sourceforge.net/.

[12] U. Bayer, C. Kruegel, and E. Kirda, “TTAnalyze: A tool for analyzing malware,” in the 15th Annual Conference of the European Institute for Computer Antivirus Research, 2006.

[13] M. Bailey, J. Oberheide, J. Andersen *et al.*, “Automated classification and analysis of internet malware,” *Lecture Notes in Computer Science*, vol. 4637, pp. 178-197, 2007.

[14] A. Moser, C. Kruegel, and E. Kirda, “Exploring multiple execution paths for malware analysis,” in IEEE Symposium on Security and Privacy, 2007.

[15] C. Willems, T. Holz, and F. Freiling, “Toward automated dynamic malware analysis using cwsandbox,” *IEEE Security & Privacy*, pp. 32-39, 2007.

[16] P. Trinius, J. Gobel, T. Holz *et al.*, “Visual Analysis of Malware Behavior Using Treemaps and Thread Graphs,” in 6th International Workshop on Visualization for Cyber Security, 2009.

[17] J. Li, M. Xu, N. Zheng *et al.*, “Malware Obfuscation Detection via Maximal Patterns,” in Third International Symposium on Intelligent Information Technology Application, 2009.

[18] "The secrets to MyDoom's success," [online], available from World Wide Web; http://antivirus.about.com/cs/allabout/a/mydoomddos.htm.

[19] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” *Technical Report 148, Department of Computer Science, University of Auckland*, 1997.

[20] "Pin - A Dynamic Binary Instrumentation Tool," [online], available from World Wide Web; http://www.pintool.org/.

[21] T. Keong. "ApiHookCheck Version 1.01," [online], available from World Wide Web; http://www.security.org.sg/code/apihookcheck.html.

[22] "QEMU," [online], available from World Wide Web; http://wiki.qemu.org/Main_Page.