Automatic Analysis and Classification of Obfuscated Bot Binaries

Ying-Dar Lin[1], Yi-Ta Chiang[1], Yu-Sung Wu[1], and Yuan-Cheng Lai[2]
(Corresponding author: Yi-Ta Chiang)

Department of Computer Science, National Chiao Tung University[1]
1001 University Road, Hsinchu, 300, Taiwan
Department of Information Management at National Taiwan University of Science and Technology[2]
43,Sec.4,Keelung Rd.,Taipei,106,Taiwan
(Email: yida@cs.nctu.edu.tw)

(Received Dec. 17, 2012; revised and accepted May 16, 2013)

Abstract

Botnets is a serious threat to Internet security. Popular defense strategies such as traffic filtering and malware detection all require a good understanding of the constituent bot binaries for creating the corresponding filter rules or signatures. This means that an effective analysis and classification process for bot binaries is needed for dealing with the threat of botnets. Unfortunately, the rampant usage of binary obfuscation these days has made the analysis and classification rather difficult. A simple string pattern matching or disassembly of the binary no longer suffices as the exact instruction sequence can be easily altered by obfuscation. In this work, we propose a new framework for automatic analysis and classification of bot binaries. The framework analyzes a bot binary's runtime system call trace and uses the longest common subsequences between system call traces for the classification of bot binaries. The framework can effectively deal with obfuscated bot binaries. Experiment result shows that the framework can attain an overall 94% true positive rate and 93% true negative rate.

Keywords: Longest common subsequence algorithm, obfuscation, system call

1 Introduction

The Internet faces many security threats nowadays ranging from low-level attacks such as packet spoofing to large-scale malicious activities such as botnets. A botnet is an autonomous network that consists of compromised computers running software agents, commonly referred to as robots or bots, under the control of an attacker. A bot-network (botnet) is typically formed to conduct nefarious activities such as DDoS attack [18], e-mail spamming [17], stealing of personal information, etc. These attacks have raised concerns over Internet security and can have severe financial impact. For example, a DDoS attack caused by botnets in New Jersey had cost a loss of over $2.5 million dollars [5].

The threat of botnets is difficult to eradicate because new types of bots appear every day. The analysis and classification of bot binaries can no longer rely on manual analysis carried out by experts solely. The process has to be automated in order to match the high birth rate of new bots these days. On the other hand, the rampant usage of binary obfuscation also brings new challenge to traditional analysis and classification techniques that are based on string pattern matching or disassembly. These traditional techniques use the raw instruction sequence to characterize a binary, and the sequence can now be easily mutated through binary obfuscation.

In this work, we present a framework for the automatic analysis and classification of bot binaries. The framework uses dynamic analysis to extract system call sequences from bot binaries. The framework then classifies the binaries based on the LCS similarity of system call sequences. We notice that obfuscation can relocate instructions in a bot binary. On the other hand, obfuscation can also introduce extra system calls into a call sequence. Both of these can negatively affect the classification accuracy. We therefore come up with heuristics to compensate these effects. Another problem is that many bots contains anti-VM code to prevent being analyzed in a virtual machine (VM), we therefore use the PIN tool to observe their behaviors in real machines. Our experiment based on 564 distinct bot binaries and 1692 variants shows that the framework is able to achieve high classification accuracy (94% true positive rate and 93% true negative rate) even with obfuscated bot binaries. Overall, the framework offers a streamlined and effective process for the automatic analysis and classification of obfuscated bot binaries.

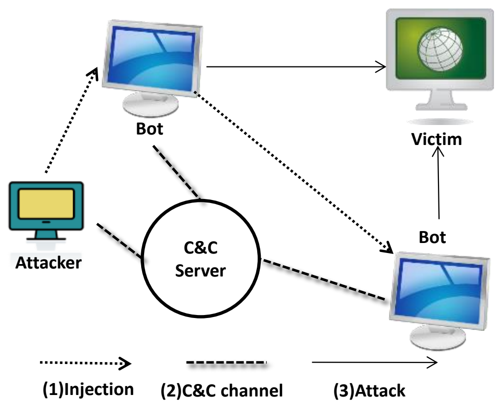


Figure 1: Architecture of a botnet

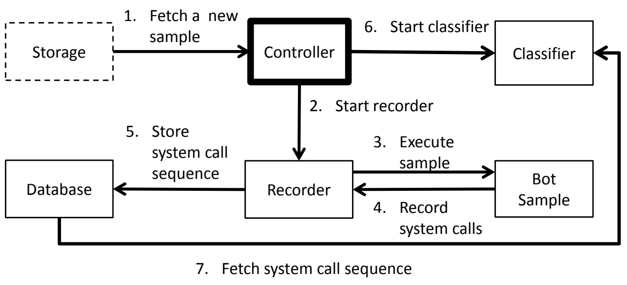


Figure 2: Architecture diagram

2 Background

2.1 Taxonomy of Botnet

A botnet is made of a bunch of bots, which are controlled by a command and control server (C&C server) as shown in Figure 1. A botnet typically follows the three-phase life-cycle, that includes: (1) the injection of bots onto vulnerable hosts, (2) the injected bots establishing connections back to a C&C server and waiting for its commands, and (3) C&C server issuing commands to the bots to order the launch of attack on a chosen victim.

The injection of bots can be achieved through many different ways such as exploiting vulnerability in network services, through e-mail attachment, via P2P file sharing, and so on. After a bot is injected into a computer, the bot will attempt to establish a communication channel with a C&C server. A popular approach is to rely on an existing IRC server to act as the C&C server. However, it is also possible to use a customized server. A malicious attacker, sometimes known as the bot herder, can remotely control the bots by issuing commands through the C&C server. The C&C communication channel is often encrypted to prevent anyone but the authorized bot herders from controlling a botnet. A botnet can have more than one C&C server to make the botnet more robust against crackdown.

Any bot in a botnet can be used to carry out attack actions. This means that it is typically difficult to track down a single attack origin for crackdown in a botnet attack. Botnet is thus a very popular choice for conducting attacks such as e-mail spamming. When the bots in a botnet are instructed to carry out attacks on a targeted victim around the same time, the botnet can become a very effective DDoS attack weapon. For instance, the botnet MyDoom [8] was used to carry out a DDoS attack on the web site of SCO Group.

2.2 Overview of Binary Analysis and Classification

For the analysis of bot binaries, there are two different approaches: static analysis and dynamic analysis. Static analysis analyzes a bot binary without actually running it. In its simplest form, static analysis can be a straightforward string pattern matching within a binary. More advanced static analysis may involve disassembly of binary, constructing function call graph, and semantic analysis of the disassembled code. For instance, Liang [11] merges function calls into modules that characterize specific types of high-level tasks such as file and registry operation. Zhang and Reeves [21] look for common patterns of assembly code sequences in malware binaries. Han [7] uses the full-name here (API) list in the full-name here (IAT) table as a signature to cassify samples. None of the above works can deal with obfuscated binaries. In the work by Natarij [13], they design a binary-to-gray-level image converter to calculate the similarity of binary codes. While they can identify different malware from the same packer, they are unable to distinguish different malware from the same packer unless the packer has weak encryption schemas.

Static analysis typically runs very fast. It does not require actually running the bot binaries (and possibly causing damages). However, it can be easily defeated by binary obfuscation [6]. One common technique used in binary obfuscation is encrypting the binary, so a straightforward string matching or disassembly will not be able to give any meaningful analysis result. More advanced static analysis tools may attempt to decrypt an obfuscated binary, but still the obfuscation can introduce extra layers of protection. For instance, the layout of a binary can be restructured and redundant data fields or garbage codes can be added to the binary to cause noise to the static analysis process. Some obfuscation tool such as Themida [15] can even translate an x86 binary into a binary for some unknown architecture and use a virtual machine (VM) of the corresponding architecture to execute the obfuscated binary.

The weakness of static analysis on obfuscated binary has led to interests in the development of dynamic binary analysis techniques. One approach is API hooking, in which key system APIs are hooked by monitoring routines to track their usage. Since API hooking incurs overhead only when the hooked APIs are invoked, the dynamic analysis process can be made quite efficient. However, a limitation with API hooking is that those in-between instruction sequences that do not involve system APIs will not be analyzed. It is also possible that a bot binary can attempt to unhook the monitoring routine or make direct API call into the kernel to bypass the dynamic analysis [20].

Another approach for dynamic analysis is through full system emulation [2], where a bot binary is executed in an operating system that runs on a hardware platform emulator (e.g. QEMU). The emulator can be modified to extract detailed runtime information such as instructions executed, memory content at arbitrary address, and so on. This kind of dynamic analysis can be very thorough. Typically, the emulated environment is isolated from the outside world, so the dynamic analysis process cannot be bypassed or disabled. However, it is possible that a bot can detect the emulated environment (e.g. through fingerprinting BIOS, and so on.) and refrain from showing its full behavior [14]. The approach also comes with significant runtime overhead due to emulation. For instance, systems running on QEMU can experience a 4~10 times slowdown compared with systems running directly on the underlying hardware [4].

Bayer, Kruegel and Kirda [3] proposed a system named "TTAnalyze" that executes a binary sample inside a virtual machine to observe the binary's runtime behaviors including file modification, registry modification and network access. A popular tool for online binary dynamic analysis is CWSandbox [20], where one can upload suspicious binaries for dynamic analysis in their sandboxed environment. A limitation with dynamic analysis is that only those executed control paths are analyzed by default. This limitation can be addressed by symbolic execution [12]. Li, Xu, Zheng and Xu [10] also use system call sequence similarity to classify samples. Their method focuses on the patterns of continuous system call. In comparison, our framework also considers more detailed features such as gap shift (Sec. 0) in a system call sequence. LeDoux [9] combines signatures from Anubis and CWSandbox to achieve higher accuracy, but more signatures also means more time to analyze samples.

3 System for Analysis and Classification of Obfuscated Bot Binaries

Figure 2 shows the architecture of the system. First, the controller fetches a bot binary sample from disk storage (step 1). It then starts the recorder (step 2) to begin dynamic analysis on the bot sample. During the dynamic analysis, the system calls invoked by the bot sample will be collected (step 3 and 4). The recorder relies on the dynamic instrumentation tool PIN [16] to record the system calls invoked by the binary during its execution. The data

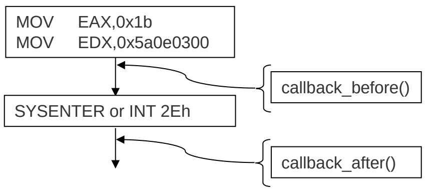


Figure 3: Intercept system calls through instrumentation (Windows platform)

```
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
);
```

Figure 4: Example of Windows system call (native API)

collected are stored in the database (step 5). Once the sample stops running or when a predefined timeout limit is reached, the controller will terminate the recorder and initiate the classifier. The classifier will classify the sample based on its system call trace (step 6 and 7).

3.1 Analysis of Bot Binaries

As mentioned in Section 2.2, API hooking is susceptible to tampering. On the other hand, full system emulation incurs a high overhead and is not suitable for the analysis of a huge volume of bot binaries. Instead, we use process-level binary instrumentation [16] as the mechanism for the dynamic analysis of bot binaries. Process-level binary instrumentation can instrument monitoring routine code into a bot binary's process memory at runtime. The instrumentation tool can breakpoint the execution of a process at locations of interests and insert monitoring code at those locations (e.g. locations where a system call is about to be invoked). An instrumented process is executed natively on the hardware, so the analysis process can be made almost as fast as that of API hooking. On the other hand, instrumentation is more versatile than API hooking in the sense that the monitoring code can be instrumented almost anywhere in the text segment of a process, not just at the system call sites. However, instrumentation-based analysis is typically limited to user-mode process and is not suitable for analyzing kernel-mode malware such as rootkit. For analyzing kernel-mode malware, it is more appropriate to rely on full system emulation.

In Figure 3, on 32-bit Windows platform, the invocation of system call relies on either software interrupt INT 2Eh or the SYSENTER instruction to transfer control into the kernel-mode system call handler. The system call number is passed by the EAX register. The call arguments are passed by the stack. A pointer to the arguments on the stack will be passed through the EDX register. We use PIN API *PIN_AddSyscallEntryFunction()* to instrument the monitoring routine *callback_before()* right before each SYSENTER/INT 2Eh instruction. This allows the recorder to intercept the invocation of each system call and collect the corresponding system call number, call arguments, and thread ID. The monitoring routine can acquire these information through PIN API *PIN_GetSyscallNumber(), PIN_GetSyscallArgument(), and PIN_GetTid()* respectively. On the other hand, the analyzer also instruments the monitoring routine *callback_after()* right after each SYSENTER/INT 2Eh instruction. This is used to collect the return value of each system call.

Some of the system call arguments may be pointers. For instance, the Windows system call *ZwQueryValueKey* has six call arguments (Figure 4). The second argument *ValueName* and the fifth argument *ResultLength* are pointers. When collecting system call information in *callback_before()*, the recorder will deference pointer arguments and record the values stored at the memory addresses pointed by the pointers.

3.2 Features for Classification: System Call Sequence

The analyzer will group the collected system calls from a bot binary based on thread IDs. In the current implementation, the analyzer only keeps the system calls of the main thread (the thread that contains the most number of system calls). The system calls in the main thread is then sorted into a system call sequence based on the invocation time of each system call.

An example of a system call sequence from an obfuscated bot sample is shown in Figure 5. The system calls in the sequence can be roughly divided into four segments. Segment A includes system calls related to the initialization of a new process. (e.g. loading of the executable image and the related library files). Segment B represents the stub loader embedded by an obfuscation tool used for initializing the runtime environment. In the case of UPX [19], segment B is mainly about the decompression of program text. For Themida, segment B corresponds to the loading and initialization of the built-in virtual machine. Of most interest to us is segment C, which contains the system calls made by the original bot binary itself. System calls in segment C characterizes the behavior of a bot binary. Segment D contains system calls used for the deallocation of resources (files, memory, etc.) at the time of process termination.

3.3 LCS Similarity of System Call Sequences

The number of bot binaries is huge. The proposed framework comes with a classification process to help the study of bot binaries by automatically identifying and grouping bot binaries into classes. The similarity between two bot binaries is judged by the similarity between their system call sequences.

Bot binaries can bear similarity in their system call sequences for at least two reasons. First, a bot binary is often obfuscated into different forms to avoid signature-based detection. The obfuscated binaries will still contain the system call behavior of the original binary, or they will not be able to fulfill the same intended functionality as the original binary. The other reason for similarity in bot binary system calls is because malware writers may reuse some code pieces from previous malware. By looking for similarity in the system call sequences, the classification process can help identify the bot variants more quickly.

The similarity between two bots is defined based on their system call sequences. Specifically, the similarity is defined by the longest common subsequence of the system call sequences of the two bots. Let us assume that the two system call sequences are 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 the respective system calls made by the two bots in ascending invocation time order. The longest common subsequence $LCS(X,Y)$ is a common subsequence of X and Y with maximal length $|LCS(X,Y)|$.

To evaluate the system call sequence similarity *S(X,Y)* between two call sequences *X* and *Y*, we define *S(X,Y)* as

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

which is the ratio of the maximal length of the common system call sequence to the length of the shorter sequence of 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 subset of Y, or Y is a subset of X. The similarity value S(X,Y) is then compared against a threshold value TS to decide if X and Y should be placed in the same class. The decision rule is

$$\begin{cases} S(X,Y) \le T_s \cdots\cdots \text{Different class,} \\ S(X,Y) > T_s \cdots\cdots \text{Same class.} \end{cases} \tag{2}$$

while the value of T_S is decided in Sec. 0 for maximizing true positive rate and true negative rate.

3.4 Improve Classification Accuracy with Gap Shift Ratio

System calls in the longest common subsequence *LCS(X, Y)* may not always come from the same locations in sequence

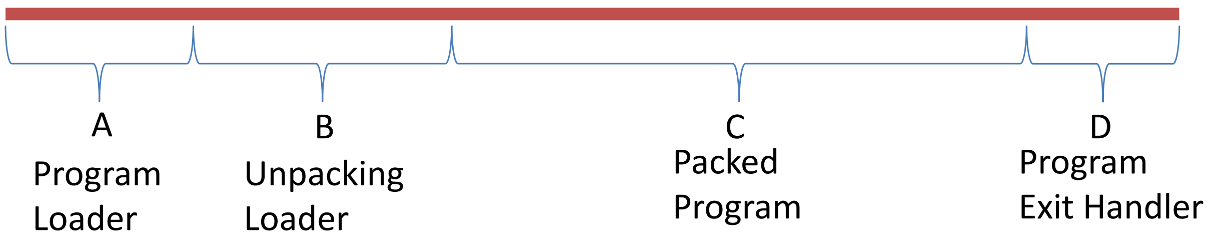


Figure 5: Segments of system calls in an obfuscated binary

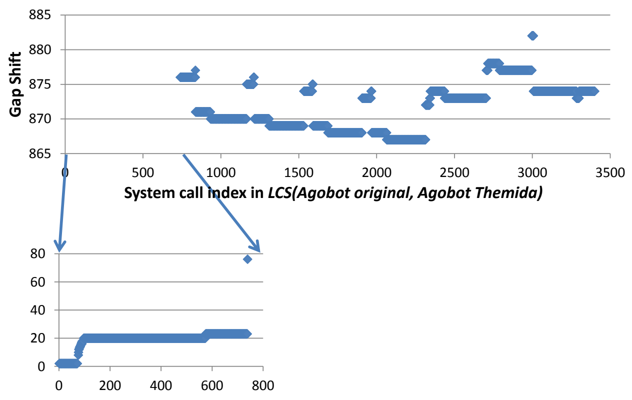


Figure 6: The gap shift sequence of Agobot original vs. Agobot Themida

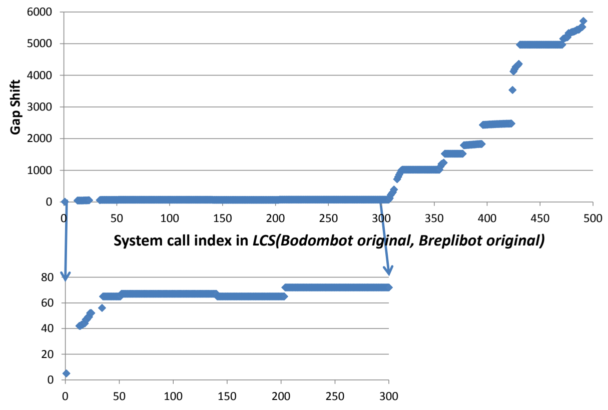


Figure 7: Gap shift value chart of Bodombot original & Breplibot origional

X and *Y*. Although these system calls appear in both *X* and *Y*, they may carry quite different semantic meanings. For instance, two consecutive CreateProcess() calls could very likely come from a function used in some initialization work. On the other hand, two CreateProcess() calls that spread far apart may more likely come from two separate functions that are not related to each other. Due to this reason, the LCS similarity between two unrelated bot binaries can sometimes become erroneously high. This will cause the classification process to put the two binaries into the same class by mistake according to Equation 2.

To address the deficiency in classification with LCS similarity alone (Equation 2), we propose a heuristic that factors in the effect of the gap shifts in system call sequences. Specifically, after we obtain the LCS sequence $(S_1, S_2, \cdots, S_k, S_{k+1}, \ldots, S_l)$ of *X* and *Y*, we will determine the respective indices for each system call S_k in *X* and *Y*. This would create two sequences of indices: I_X: $(p_1, p_2, p_3, \ldots, p_l)$ for *X* and I_Y: $(q_1, q_2, q_3, \ldots, q_l)$ for *Y*. For example, p_1 is the index of system call S_1 in *X* and q_1 is the index of S_1 in *Y*. If S_1 is the first system call in *X*, then p_1 is 1. And, if S_1 is the 100th system call in *Y*, then q_1 are 100.

The gap shift sequence *G* is constructed by taking the difference of each pair of elements from I_X and I_Y, so we have $G: (p_1 - q_1, p_2 - q_2, p_3 - q_3, \ldots, p_l - q_l)$. We then define *N(G)* as the number of the distinct values in the sequence *G*. According to our observation, for two bot binaries that should belong to the same class, their *N(G)* value will be small. Because they are similar in their behaviors, their system calls in common should bear similar semantic meanings, and the relative gap shifts should be similar as well. On the other hand, for two unrelated binaries, the corresponding *N(G)* value will be usually high.

Figure 6 shows the gap shift sequence between Agobot original (unpacked) and Agobot Themida (obfuscated by Themida). The gap shift values for the first 762 system calls are below 80 because they correspond to the initialization of a new process (Segment A of Figure 5). This part of the system call sequence is hardly affected by the Themida packer. From the 763th system call and onward, we can see a huge shift (about 865) in the system call indices. This shift is due to the unpacking loader code (Segment B in figure 5) inserted by the Themida packer between the 762th system call and the 763th system call. The two bot binaries are related, and as we can see from the plot, the gap shift values only take on a few levels (the corresponding *N(G)* value is 27).

Figure 7 shows the gap shift sequence between two different bots: Bodombot and Breplibot. The LCS similarity between these two bot binaries is 0.97, which will cause incorrect classification according to Equation 2. Looking at the gap shift sequence plot in Figure 7, we can see that the gap shift values take on many different levels (the *N(G)* value is 100). This indicates that the common system calls as identified by LCS are located at quite different locations in Bodombot and Breplibot, meaning that the corresponding behaviors shall be quite different.

The *N(G)* value also increases with the length of a gap shift sequence. We can normalize it by the length of the gap shift sequence *L*=|*G*| and define the gap shift ratio R as

$$R = \frac{N(G)}{L}. \quad (3)$$

Combined with Equation 2, the criteria for determining if two bot binaries belong to the same class is now defined as

$$\begin{cases} S \le T_s \cdots\cdots\cdots\cdots\cdots\cdots \text{Different class}, \\ S > T_s \text{ and } R > T_r \cdots \text{Different class}, \\ S > T_s \text{ and } R \le T_r \cdots \text{Same class}. \end{cases} \quad (4)$$

3.5 Improve Classification Accuracy of Call Sequences with Segment Identification

In Figure 5, we see that only segment C of a system call sequence is of relevance for identifying bots with similar behaviors. The system calls in segments A and D are common to most executable files, and segment B is

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

Figure 8: System call sequence in segment B from a Themida-obfuscated binary

introduced by an obfuscation tool. We can improve the classification accuracy by ignoring segments A, B, and D in the calculation of LCS similarity and gap shift ratio. Segment A and D are easy to identify and ignore as they are very much the same across all executables.

Segment B, on the other hand, is much more difficult to deal with, because it depends on the type of obfuscation tool in use. As a result, we have to build profiles for each different obfuscation tool in order to identify and remove segment B effectively. As an example, a Themida-obfuscated binary always has the system calls shown in Figure 8 in segment B, which can be reliably removed to improve classification accuracy.

To build the profile, we use LCS to identify the common subsequence over a bunch of binaries obfuscated by a given packer (e.g. Themida). The resulting common subsequence that is left should include only segment A, B, and D. Since segment A and D are standard to any executable, we can trim them away in the recorder and extract segment B as the profile for the corresponding obfuscation tool.

4 Experiments

We conduct four experiments to evaluate the proposed framework. The first two experiments (Section 4.1 and Section 4.2) look at the effect of obfuscation on LCS similarity and gap shift ratio. Ideally, neither of them should be significantly affected by obfuscation, or the proposed framework would fail to accurately classify obfuscated bot binaries according to Equation 4. In the third experiment (Section 4.3), we look at how the selection of different threshold values T_S and T_R affects the classification accuracy. In the fourth experiment (Section 4.4), we evaluate the overall effectiveness of our framework with a large sample of 564 real-world bot binaries.

4.1 LCS Similarities and Gap Shift Ratios between Variants of a Bot Sample

In this experiment, we calculate the LCS similarities and gap shift ratios between bot variants, which are created by obfuscating 10 (unpacked) bot samples with different packers. We use the 10 unpacked bot samples (Table 1) as the baseline (denoted as group A) in this experiment. We then obfuscate each of those 10 bot samples with ASProtect [1] to create ASProtect-obfuscated test targets (denoted as group B). We also create 10 Themida-obfuscated test targets (denoted as group C) and 10 UPX-obfuscated test targets (denoted as group D). For each bot sample, there are six different combinations for evaluating the LCS similarities and gap shift ratios: (A,B), (A,C), (A,D), (B,C), (B,D), and (C,D). For instance, in the case of (A,B), we will take one bot from group A and calculate the LCS similarity and gap shift ratio of it with the corresponding

Table 1: List of bots used in the experiment

Id	*MD5*	*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

ASProtect-obfuscated version of the bot from group B. This yields 10 data points, and overall, there will be 60 data points, which are summarized in Figure 9.

Figure 9 shows the distrbution of the 60 data points. Here, each data point corresponds to the LCS similarity (S) and the gap shift ratio (R) between two variants of a bot sample. Each of the circle in Figure 9 represents a group of data points with the same LCS similarity and gap shift ratio.

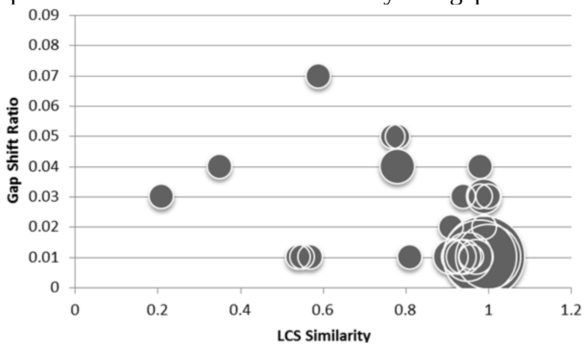


Figure 9: Distribution of LCS similarity and gap shift ratio

The diameter of the circle is proportaionl to the number of data points in that circle. As we can see, most of the data points present a high LCS similarity values (close to 1) indicating that the two corresponding variants are from the same origin. On the other hand, the gap shift ratios are low (near 0.01), which also indicates the variants are from the same origin. This shows that LCS similarity and gap shift ratio are not sensitive to obfuscation with respect to identifying bot variants of the same origin.

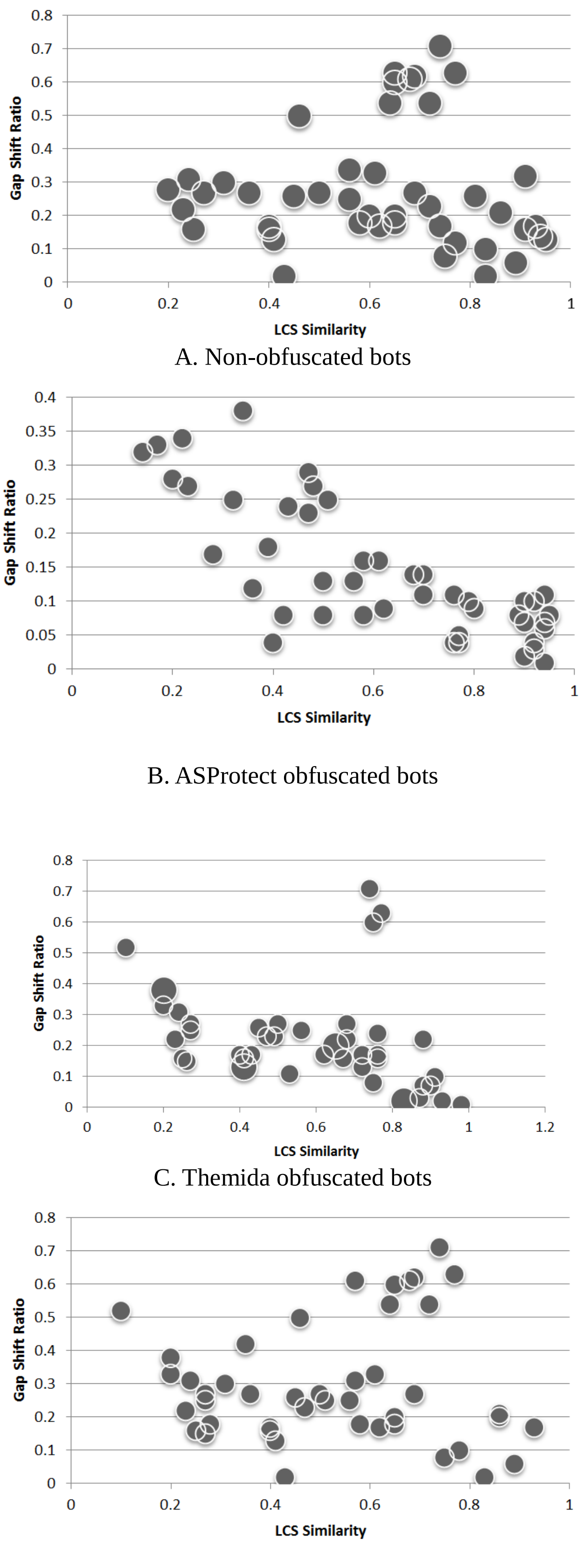
4.2 LCS Similarities and Gap Shift Ratios between Distinctive Bot Samples

In this experiment, we evaluate the LCS similarities and gap shift ratios between bot samples of different origins. First, we calculate the pair-wise LCS similarities and gap shift ratios for the 10 unpacked bot samples (group A in Sec. 0). The result is presented in Figure 10-A. We then calculate the pair-wise LCS similarities and gap shift ratios for the ASProtect-obfuscated bot samples (group B) with the result shown in Figure 10-B. The results for Themida-obfuscated bot samples (group C) and UPX-obfuscated bot samples (group D) are presented in Figure 10-C and Figure 10-D respectively.

This result shows that the LCS similarities (S) between bot samples of different origins are widely dispersed. The LCS similarities no longer concentrate near 1 as in Sec. 0. Some of the data points have high LCS similarities, but comparing to Figure 9, their gap shift ratios (*R*) are mostly above 0.05. Thereby, if we consider both the LCS similarity and gap shift ratio together as in Equation 4, we can also reliably distinguish bot samples of different origins.

4.3 Choosing T_S (LCS Similarity Threshold) and T_R (Gap Shift Ratio Threshold)

From the previous two experiments, we know that for bot variants from the same origin, their LCS similarity values are close to 1 and their gap shift ratios are close to 0. On the other hand, for bot samples from different origins, their LCS similarities are widely dispersed and the gap shift ratios tend to be larger. Based on the observation, we designed the classification criteria of Equation 4. To determine the proper threshold values T_S and T_R in



A. Non-obfuscated bots

B. ASProtect obfuscated bots

C. Themida obfuscated bots

D. UPX obfuscated bots

Figure 10: Result distribution on the same obfuscation

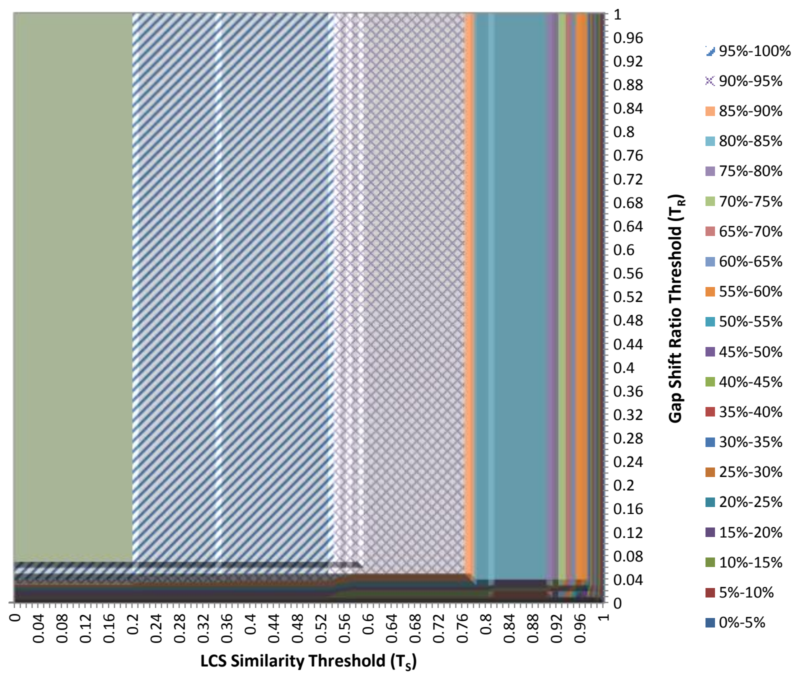


Figure 11: True positive rate

Table 2: Classification accuracy

True Positive ate	*True Negative Rate*
94%	93%

Equation 4, we experiment with different T_S and T_R values and look at the corresponding classification accuracy in terms of true positive rate (TPR) and true negative rate (TNR). True positive rate represents the percentage of bot samples classified in the same group, which are indeed from the same origin. On the other hand, true negative rate represents the percentage of bot samples classified into different groups, which indeed belong to different origins.

The effect on TPR and TNR when varying the LCS similarity threshold (T_S) and gap shift ratio threshold (T_R) is shown in Figure 11 and Figure 12. We thereby consider 0.53 as an appropriate threshold value T_S and 0.05 as the threshold for T_R because this can achieve an overall 95% TPR and 92% TNR.

4.4 Classification Accuracy on a Large Sample of Bots

In this experiment, we conduct a large scale experiment with 560 distinct bot samples from the honeypot at campus, along with 4 legitimate programs: notepad, Firefox, MS Word, and 7-Zip. For each of the 564 binaries, we create 3 obfuscated variants with ASProtect, Themida, and UPX respectively. This results in a total of 2256 binaries, including original programs and obfuscated ones. We then use the proposed framework to analyze and classify all the binaries. The threshold T_S is set to 0.53 and the threshold T_R is set to 0.05 according to Section 4.3.

The classification result is summarized in Table 2. Overall, we can see that the framework achieves a decent 94% true positive rate and 93% true negative rate on the classification of the 2256 binaries.

5 Conclusions

We propose a framework for the automatic analysis and classification of obfuscated bot binaries. The framework use dynamic analysis to extract the system call sequence of a bot binary. Since system calls define the interactions between a program (the bot binary) and the operating system, obfuscation can hardly alter the call sequence without breaking the interactions. We rely on this property and use the system call sequence to characterize the behavior of a bot binary.

For the classification, we define a similarity metric between two bot binaries based on the longest common subsequence (LCS) of their system call sequences. The LCS similarity does not consider the relative locations of system calls in the two binaries, and that can cause mis-classification in some cases. This is addressed by a heuristic called gap shift ratio, which detects excessive variation in the relative locations of system calls.

Although obfuscation can hardly change the original system call sequence in a bot binary, it can often introduce additional system calls into an obfuscated binary. Most of them are due to the obfuscation tool's stub code. The additional system calls are noises to the classification process, and we have come up with a segment

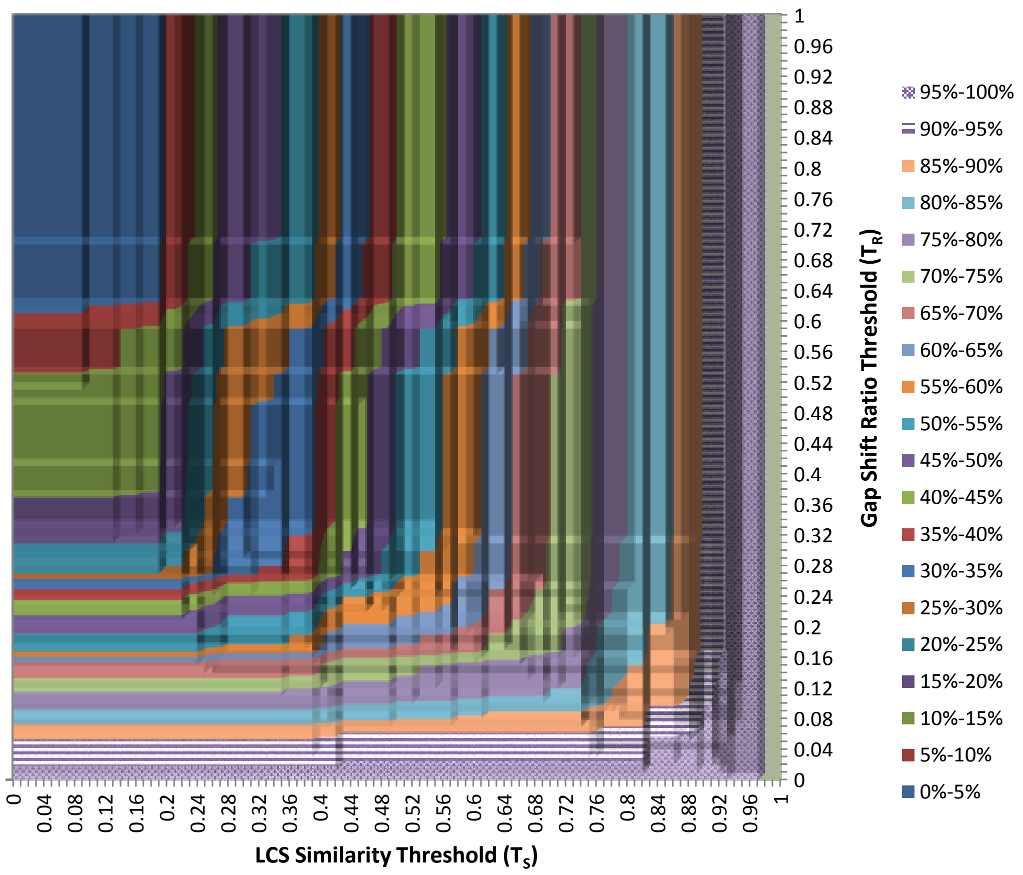


Figure 12: True negative rate

identification process to filter out these noises. Overall, the framework can achieve 94% true positive rate and 93% true negative rate.

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 plan to implement an on-line analysis process, where the system can work as an anti-virus tool that can detect running bots on a computer.

References

[1] ASPack, Software Protection Tools for Software Developers. (http://www.aspack.com/asprotect.html)

[2] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," *Network and Distributed System Security Symposium*, 2009.

[3] U. Bayer, C. Kruegel, and E. Kirda, "Ttanalyze: A tool for analyzing malware", in *15th Annual Conference of the European Institute for Computer Antivirus Research*, 2006.

[4] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference*, 2005.

[5] K. K. R. Choo, "Zombies and botnets," *TRENDS & ISSUES in crime and criminal justice*, 2007.

[6] C. Collberg, C. Thomborson, and D. Low, *A taxonomy of Obfuscating Transformations*, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[7] K. Han, I. Kim, and E. Im, "Malware classification methods using API sequence characteristics," in *The International Conference on IT Convergence and Security*, Sewon, Korea, 2011.

[8] M. Landesman, *The Secrets to Mydoom's Success*. (http://antivirus.about.com/cs/allabout/a/mydoomddos_3.htm)

[9] C. LeDoux, A. Walenstein, and A. Lakhotia, "Improved malware classification throughsensor fusion using disjoint union," *Information Systems, Technology and Management Communications in Computer and Information Science*, vol. 285, pp. 360-371, 2012.

[10] J. Li, M. Xu, N. Zheng, and J. Xu, "Malware obfuscation detection via maximal patterns," in *Third International Symposium on Intelligent Information Technology Application*, 2009.

[11] Z. Liang, T. Wei, Y. Chen, X. Han, J. Zhuge, and W. Zou, "Component similarity based methods for

automatic analysis of malicious executables," in *Virus Bulletin Conference*, 2007.

[12] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," I*EEE Symposium on Security and Privacy*, 2007.

[13] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware Images: Visualization and Automatic Classification," in *The 8th International Symposium on Visualization for Cyber Security*, Pittsburgh, U.S.A, 2011.

[14] A. A. e. Omella, *Methods for Virtual Machine Detection*. (http://www.s21sec.com/descargas/vmware-eng.pdf)

[15] Oreans Technology : Software Security Defined. (http://www.oreans.com/themida.php)

[16] Pin, Pin - a dynamic binary instrumentation tool. (http://www.pintool.org/)

[17] J. Sheu, "An efficient two-phase spam filtering method based on e-mails categorization," *International Journal of Network Security*, vol. 9, no. 1, pp. 34-43, July 2009.

[18] J. Udhayan and T. Hamsapriya, "Statistical segregation method to minimize the false detections during DDoS attacks" *International Journal of Network Security*, vol. 13, no. 3, pp. 152-160, Nov. 2011.

[19] UPX, UPX: the Ultimate Packer for eXecutables – Homepage. (http://upx.sourceforge.net/)

[20] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, 2007.

[21] Q. Zhang and D. S. Reeves, "Metaaware: Identifying metamorphic malware," in *Annual Computer Security Applications Conference*, 2007.

Ying-Dar Lin is Professor of Computer Science at National Chiao Tung University (NCTU) in Taiwan. He received his Ph.D. in Computer Science from UCLA in 1993. He served as the CEO of Telecom Technology Center during 2010-2011 and a visiting scholar at Cisco Systems in San Jose during 2007¡V2008. Since 2002, he has been the founder and director of Network Benchmarking Lab (NBL, www.nbl.org.tw), which reviews network products with real traffic. He also cofounded L7 Networks Inc. in 2002, which was later acquired by D-Link Corp. He recently, in May 2011, founded Embedded Benchmarking Lab (www.ebl.org.tw) to extend into the review of handheld devices. His research interests include design, analysis, implementation, and benchmarking of network protocols and algorithms, quality of services, network security, deep packet inspection, P2P networking, and embedded hardware/software co-design. His work on ¡§multi-hop cellular¡¨ was the first along this line, and has been cited over 500 times and standardized into IEEE 802.11s, WiMAX IEEE 802.16j, and 3GPP LTE-Advanced. He was elevated to IEEE Fellow in 2013 for his contributions to multi-hop cellular communications and deep packet inspection. He is currently on the editorial boards of IEEE Transactions on Computers, IEEE Computer, IEEE Network, IEEE Communications Magazine - Network Testing Series, IEEE Wireless Communications, IEEE Communications Surveys and Tutorials, IEEE Communications Letters, Computer Communications, Computer Networks, and IEICE Transactions on Information and Systems. He recently published a textbook "Computer Networks: An Open Source Approach" (www.mhhe.com/lin), with Ren-Hung Hwang and Fred Baker (McGraw-Hill, 2011). It is the first text that interleaves open source implementation examples with protocol design descriptions to bridge the gap between design and implementation.

Yi-Ta Chiang performed this research while at National Chiao Tung University. He is now an engineer at Network Benchmarking Lab. His research interests include Network Security and performance evaluation. Chiang has an MS in computer science from National Chiao Tung University.

Yu-Sung Wu received the B.S. degree in Electrical Engineering from National Tsing Hua University, Taiwan in 2002, and the Ph.D. degree in Electrical and Computer Engineering from Purdue University, West Lafayette, Indiana in 2009. He is an assistant professor in the Department of Computer Science, National Chiao Tung University, Taiwan, where he leads the Laboratory of Security and Systems. His research interests include security, dependability, and systems.

Yuan-Cheng Lai received the Ph.D. degree in computer science from National Chiao Tung University, Hsinchu, Taiwan, in 1997. In August 2001, he joined the faculty of the Department of Information Management at National Taiwan University of Science and Technology, Taipei, Taiwan, where he has been a professor since February 2008. His research interests include wireless networks, network performance evaluation, network security, and content networking.