Identification of Bot Commands By Run-time Execution Monitoring

Younghee Park, Douglas S. Reeves
Cyber Defense Laboratory, Computer Science Department
North Carolina State University, Raleigh, NC, USA
{*ypark3, reeves*}*@ncsu.edu*

***Abstract*—Botnets pose serious threats to the Internet. In spite of substantial efforts to address the issue, botnets are dramatically spreading. Bots in a botnet execute commands under the control of the botnet owner or controller. A first step in protecting against botnets is identification of their presence, and activities.**

In this paper, we propose a method of identifying the high-level commands executed by bots. The method uses run-time monitoring of bot execution to capture and analyze run-time call behavior. We find that bots have distinct behavior patterns when they perform pre-programmed bot commands. The patterns are characterized by sequences of common API calls at regular intervals. We demonstrate that commands aiming to achieve the same result have very similar API call behavior in bot variants, even when they are from different bot families. We implemented and evaluated a prototype of our method. Run-time monitoring is accomplished by user-level hooking. In the experiments, the proposed method successfully identified the bot commands being executed with a success rate of 97%. The ability of the method to identify bot commands despite the use of execution obfuscation is also addressed.

***Keywords*-Intrusion detection, Malware, Botnet**

I. Introduction

A botnet is a system that remotely controls malicious programs running on compromised hosts. Botnets are now a major source of network threats, including DDoS, spam, identity theft, click frauds, etc. [1]–[3]. Botnets are still rapidly proliferating, and communicate using a variety of protocols, such as IRC, HTTP, peer-to-peer, etc. The cumulative size of botnets is estimated to be in the millions of hosts [2], [4], [5]. Due to the huge number of botnets, and evolving botnet protocols, it appears difficult to block and/or remove all bots on the Internet. Therefore, a first line of defense is (at a minimum) to recognize bots and the actions they are taking.

As described in [3], bots are typically activated by bot *commands* through a communication and control channel (C&C channel) opened by attackers (i.e. botmasters) from remote sites. The issued bot commands may be simultaneously performed by a group of bots in the botnet, as they have been programmed. The study of bot behavior in response to issued commands is important for the development of effective countermeasures, for tracing botnet growth, and for protecting the vulnerable infrastructure that bots target. Identification of the victims targeted by botnets may also be facilitated by a thorough analysis of bot commands.

Much of the research on botnets has focused on detecting bots and C&C channels by analysis of network traffic [1], [2], [6]–[8]. These approaches have been very successful, but will have problems if traffic is encrypted, or if the botnet protocols are changed (e.g., using P2P protocols instead of IRC). In contrast, several papers have focused on analyzing bot behavior on the host [9], [10], using such common bot characteristics as remote initiation, C&C channel establishment, etc. However, these papers have only focused on bot detection, rather than on identifying specifically the actions that the bot is taking. Moreover, after establishment of a C&C channel, bots are often inactive until receipt and execution of bot commands [10]. Although the quick removal of bots and their communication channel are important, it is also important to identify the purpose of the bot, and the intended target, through monitoring of bot execution.

In this paper, we propose a method for monitoring and analyzing bot execution to identify the bot commands that are being executed. This method, called *BotTee*, is designed to recognize the characteristic behavior triggered by each command, independent of superficial differences in the syntax of various bot protocols. This task is difficult since bots of different families are independently programmed, and the creator may intentionally obfuscate command execution. However, we demonstrate that bot commands with the same purpose (designed to accomplish the same results) result in run-time behavior that is highly correlated, across all types of bots.

This similarity in run-time behavior may be due to several reasons. First, bot software of different types may have common origins; true (breakthrough) innovation in malware is rare [3], [9]–[11]. Second, bots inevitably use existing system libraries to successfully perform bot commands. The programming effort and execution overhead to implement these functions in the application otherwise may be too high, and/or privileged information may not be available directly to the application. Third, the major bot commands are well

This material is based upon work supported by the National Science Foundation under Grant Numbers 0627505 and 0831081. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1063-9527/09 $26.00 © 2009 IEEE
DOI 10.1109/ACSAC.2009.37

known, and their number is limited. Specifically, several important malicious activities that many bots are used for, such as DDoS, spam, etc., are repeatedly seen in bot variants, regardless of their origin.

BotTee works by intercepting Windows API system calls on the fly, for a set of popular calls. As a bot executes, the API call trace is compared to a set of call patterns, or *templates*. These templates are derived from previously-monitored execution of bot programs. Construction of the templates, and comparison of a template with an in-progress bot call trace, both make use of subsequence matching and statistical correlation techniques.

In experiments, BotTee was evaluated by executing real bot instances in a private network. The results show that syntactically different bot commands with the same purpose have behavior that is highly correlated. This holds true whether the bots are taken from the same, or different, bot families. The results demonstrate that bot commands can be accurately identified during execution.

The contribution of this work is the following. A technique for accurately identifying bot commands is presented. This method requires neither analysis of network traffic, nor reverse engineering of bot protocols or languages. It is demonstrated that bot commands with the same purpose (either in the same, or different families) exhibit very similar call behavior. The method requires intercepting a core set of popular system calls commonly used by bot programs, and has a modest performance impact on application execution.

The paper is organized as follows. Section II describes a new method for identifying bot commands, through runtime execution monitoring. Section III presents the results of experimentally evaluating this method. Section IV discusses the application of the proposed method, and section V compares the proposed method to previous approaches. Section VI summarizes the method.

II. The Proposed Method

BotTee is a system that monitors program execution to recognize bot commands. The execution of these commands is in response to control signals sent from a remote bot master. Monitoring involves "hooking" system API calls which are typically invoked during bot command execution.

The steps in the proposed method are as follows (refer to Figure 1). Firstly, a *template* must be created for each bot command to be identified. This template is a pattern based on a common subsequence of API calls, and information about the timing of those calls. The template generator has two components: a bot command identifier, and a correlation engine. The command identifier intercepts API calls when a command is executed. The intercepted call traces from multiple bots are then processed by the correlation engine to generate a semantic template for one bot command. This process is repeated for as many different bot commands as are desired.

General Commands	Host Control Commands
login/logout, reconnect, id alias, action, join, part privmsg, mode, cmdlist about/version, disconnect nick, rndnick, status, quit	remove/die, clone, open, delete sysinfo, shutdown, listprocess passwords, killthread, killprocess execute, sendkey/getcdkey keylogger, threads, opencmd
Network Control Commands	**Attack Commands**
server, netinfo, download, update, dns redirect, httpd/httpserver scan, visit	synflood, updflood httpflood, pingflood email

Table I
Classification for bot commands

Secondly, the derived set of templates is used to identify the commands that are executed by a bot. A pattern matcher recognizes a command by comparing the runtime execution trace to the set of templates. This process uses the same basic techniques for comparison as are used for template construction. The following sections explain the proposed method in more detail.

A. Bot behavior classification through bot commands

Table I shows a variety of bot commands which have been identified in actual botnets.[1] A bot command corresponds to a specific, programmed action to be taken by a bot program. Based on [3], we have classified bot commands into several groups. One group are *general* commands, invoked by the attacker to manage the botnet. Examples are obtaining a bot nickname (e.g. 'nick' in Table I), or making a bot terminate operation (e.g. 'disconnect', 'quit' in Table I). A second group are *host control* commands. These are used to obtain host information and/or cause some (malicious) actions on the host. Examples are application execution (e.g. 'execute'), and information extortion (e.g. 'sysinfo'). A third group are *network control* commands. These are used to obtain information about the host network (e.g. 'netinfo', 'scan'), and/or to control network behavior. Examples of the latter include changing the C & C server (e.g. 'server'), or redirecting traffic (e.g. 'redirect'). A last group of *attack* commands will launch attacks on intended victims. Examples include denial of service, or spam.

B. Hooking API calls

BotTee employs user-level *hooking* to intercept call traces on a host. Hooking is a powerful technique to understand how an application interacts with an operating system [12]. In addition, hooking can be used to change program execution. Hooking is useful to monitor bot behavior and gain control over a bot running on a compromised host, without requiring access to source code. Interception and

[1] A description of the function of each bot command is given in [3]. Table I includes only one command of each type; in practice, different bots have different command syntaxes.

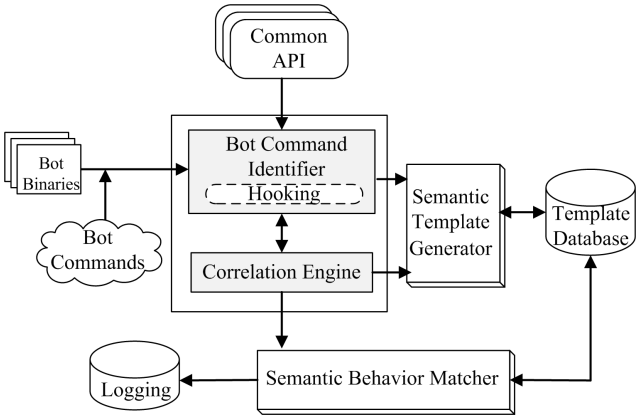


Figure 1. A system architecture for BotTee. It has two large parts: a semantic template generator and a semantic behavior matcher for a dynamic bot behavior identification.

analysis of the OS API calls is unaffected by instruction-level obfuscations of bot code, or by differences in bot command syntax.

The majority of bots today execute on PCs running the Windows OS. These bots invoke Windows functions through the API provided to applications, and available in the form of dynamically-linked libraries (DLLs). Both sequence and timing for API calls are used for analysis purposes. When each API call is intercepted, the time (in the execution history of the program) that the call is intercepted is also recorded.

Full interception of all API calls might have a performance impact that interfered with bot execution. A compromise is to intercept only a subset of system calls that are important for bot command analysis purposes. K. Jain and R. Sekar [13] developed a system call interposition infrastructure at the user level (for Linux), and suggested monitoring only a subset of system calls for intrusion detection, and showed the performance benefits. Similarly, we propose to hook only a limited set of Windows API calls. These calls are referred to as the *ComAPI* ("Common API" in Figure 1). The set of calls in ComAPI was derived by static analysis of actual bot binaries, using the method of [11]. This analysis identified approximately 300 commonly-used API functions from 50 real bot instances. Among them, 153 APIs were in file `kernel32.dll`; the rest were found in `user32.dll, advapi32.dll, ws2_32.dll (Wsock32.dll)`, etc. ComAPI consists of the complete set of user-level system calls executed by these 50 bot instances. Note that a user-level API call may result in the execution of zero, one, or more than one native API or core system service calls. As an example, the `DeleteFile` call in `kernel32.dll` invokes multiple system service calls, such as `NtOpenFile`, `NtSetInfoFile`, and `NtCloseFile`.

C. Bot Command Identifier

Once the selected set of the API calls are hooked, it is necessary to decide what sequence of system calls may correspond to a bot command. The proposed method for identifying the start and end of execution of a bot command in the intercepted system call trace, is as follows. Commands are sent to the bot by the bot master, and typically generate a response by the bot back to the master. These correspond to the system calls `recv` and `send` with a socket descriptor. An execution trace that is delimited by these system calls is a candidate for identification as a bot command.[2]

This initial step determines the start and end of each execution trace that may correspond to a bot command. Following this, each such trace must be further processed to improve the results of bot command identification. It was found in practice that one bot command may repeatedly execute the same API call many times (even hundreds of times). For purposes of bot command identification, it is not useful to know the number of repetitions of a specific API call; rather, the occurrence of a API call, and the sequence in which *different* API calls occur, is more likely to be useful for command identification. Therefore, repeated consecutive occurrences of the same API call in a trace are eliminated, under control of a parameter denoted γ. In processing the trace, if a single API call is intercepted more than γ times in a row, only the first γ occurrences are retained in the trace. For example, if the 4 system calls `A`, `B`, `C`, and `D` were intercepted in the length 14 sequence `AAABCCAAAADDDA`, after removal of consecutive repetitions with $\gamma = 2$ the result would be the length 10 sequence `AABCCAADDA`.

The result of this step is a set of execution traces, each corresponding to the execution of one bot command. Such a trace is termed a *semantic unit*. For instance, a semantic unit for the 'synflood' command in Table I, after removal of duplicate consecutive calls, is `socket`, `TLSGetValue`, `InterlockedDecrement`, `ioctlsocket`, `connect`, `WaitForSingleObject`, etc. These semantic units are processed by a correlation engine to derive templates, and to identify bot commands at runtime.

D. Correlation Engine

In this section, we discuss the correlation engine illustrated in Figure 1. This engine is used to create command templates, and to match captured system call traces to these templates. Semantic units, described in the previous section, are input to this process.

Generating templates for bot commands makes use of a longest common subsequence algorithm (LCS) [14], and

[2]The two calls, NtDeviceIoControlFile and URLDownloadToFile can be an alternative way to notice the arrival of the commands, along with the Winsock API calls. In our experiments, these two calls are always followed immediately by Winsock API calls. The proposed method focuses on identifying specific bot behaviors for bot commands by using a fixed set of user-level ComAPI only. Adding other (native) APIs can be considered as future work if needed.

statistical correlation. The motivation for this approach is based on the following two insights, obtained from investigation of real bot instances. First, in spite of superficial differences, if two bot commands have the same purpose, the API calls issued to perform them will be very similar, and their execution timing will be correlated. Second, the API function calls that are executed by the same bot command, regardless of the bot family, are typically executed with little variation. This may be due to reuse of code, imitation of functionality, popularity of widely used compilers and libraries, and/or the desire to code in an efficient way.

Let X and Y be different bots which execute two bot commands that have the same purpose. During execution of each command, many API calls found in ComAPI will be intercepted from the two bots. Suppose the command identifier produces two W semantic units from many execution traces (for a specified value of γ) for X and Y. In these W semantic units, let the ith execution trace (semantic unit) for X be denoted $S_i(X)$, and the jth execution trace for Y be denoted $S_j(Y)$. For each value of i and j, $S_i(X)$ and $S_j(Y)$ are execution traces for bot commands executed by different programs that potentially are the same command.

For each such pair, a standard algorithm [14] is used to find the longest common subsequence (LCS) of the two call traces. This LCS for traces $S_i(X)$ and $S_j(Y)$ is termed the *common API call trace* (CACT) for the pair, and is denoted $L_{i,j}$, with length $k_{i,j}$.

As mentioned, for each call in ComAPI that is intercepted, the time at which the function was executed is recorded. Let the timing vector for the CACT $L_{i,j}$ for X be denoted $T_{i,j}(X)$, with length $k_{i,j}-1$. The tth element in this vector represents the time interval between the execution of the tth and the $(t+1)$th system calls, occurring in that portion of $S_i(X)$ corresponding to $L_{i,j}$. Similarly, the timing vector $T_{i,j}(Y)$ corresponds to the intervals between successive API calls in that portion of $S_j(Y)$ corresponding to $L_{i,j}$.

Let the mean and standard deviation for the elements in $T_{i,j}(X)$ be denoted $\mu_{i,j}(X)$ and $\sigma_{i,j}(X)$, and similarly for $T_{i,j}(Y)$. The correlation coefficient $\rho_{i,j}$ of these two timing vectors may be computed in the standard way as

$$\rho_{i,j} = \frac{1}{k_{i,j}-1} \sum_{t=1}^{k_{i,j}} \frac{(T_{i,j}(X)[t] - \mu_{i,j}(X))(T_{i,j}(Y)[t] - \mu_{i,j}(Y))}{\sigma_{i,j}(X)\sigma_{i,j}(Y)} \tag{1}$$

The correlation coefficient has a maximum of 1, which occurs when the vectors are completely correlated.

Let H_1 be the hypothesis that the two commands from bots X and Y that are being compared are semantically the same command (i.e., have the same function or purpose). Define θ_1 as $P(\rho_{i,j} > \delta) \mid H_1)$, i.e., θ_1 is the probability that the two commands will have a sufficiently high correlation coefficient when H_1 is true. To decide that the two commands are the same, we require that θ_1 must be greater than 0.95, with a confidence level of .05, for an appropriately chosen value of δ.

When two bots that use similar techniques execute the same bot command, the correlation value of their timing vectors will be very high (close to 1). Even though attackers may randomly inject delays for timing obfuscation, it is unlikely they will be willing to tolerate too high a delay in the completion of their objective. The injection of extraneous system calls to obfuscate command execution is also possible, and is further investigated in section III-F.

The above scheme applies to pairs of bots, and can be extended to the analysis of any number of bots. Because the number of sequences in all of the execution traces for many bots is constant, the LCS problem is solvable in polynomial time by dynamic programming [14]. In such a case, if bot commands that have the same function are executed by many different bots, the CACT will be the longest common subsequence that occurs in all of the execution traces for that command, from all of the bots. The use of the common subsequence makes our scheme robust even if the execution traces are obfuscated. The timings for the CACT for all bot traces from which the template is created are also saved as part of the template.

Construction of templates is an off-line process that is conducted on previously-detected bots. These bots are controlled and monitored in a restricted environment, which means that commands are issued to them, during which their system call traces are captured. Therefore, the commands that are issued to the bots are known. It is further assumed that manual inspection has revealed which bot commands have the same purpose. The system call execution traces for such commands should therefore be considered for pairing in the construction of a template.

Lastly, the template database, as shown in Figure 1, stores the semantic templates derived by the correlation engine. For each bot command that is found to occur in more than one bot, there is one template. The templates are specifications including the CACT for that command and the timing vector for every execution trace for every bot that was analyzed that executes that command. The set of templates are used to identify individual bot behaviors in real time, as described in the next section.

E. A Real-time Semantic Behavior Matcher

The previous section has described how bot system call execution is hooked, and how the results are used to create a set of templates. Each template corresponds to a unique bot command. A bot that is currently executing may also be hooked. Its call execution traces can then be compared with this set of templates. The result will be the identification of bot commands being executed by the bot.

When a bot executes a bot command in real time, the behavior matcher attempts to match the command with an entry in the template database, as illustrated in Figure 1. The

method of searching for a matching template uses the same techniques employed for constructing templates.

Suppose the API calls in ComAPI are intercepted during the execution of a suspected bot program. The trace of these calls, denoted S, is then processed to identify the start and stop of execution of possible bot commands, as described above. The result is a set of semantic units for this program. For each semantic unit, duplicate consecutive system calls are removed, under the control of the user-specified repetition parameter γ.

The resulting reduced trace for one semantic unit U is then compared to all of the templates of bot commands. The comparison between U and a template starts by computing the CACT between the two, using a longest common subsequence algorithm.

To identify a bot command in the execution trace S of a bot, using a set of previously-derived templates, a candidate template must be identified. We propose for this purpose to select as a candidate template that one which has the longest common API call trace (CACT) with U, from among all templates. In the event of ties, the candidate template is then the template that has the smallest difference with the CACT. The difference between a template and the CACT with U is calculated as the difference between the length of this CACT with U, and the length of the template.

When a candidate template has been identified, the timing vectors of the CACT for U and this template are then created and correlated, as described by Equation 1. The result is a correlation coefficient ρ, calculated as follows. The correlation of U's timing vector with each timing vector in the template is computed according to Equation refcorrelation. ρ is chosen to be the maximum of these individual correlations. If the highest ρ exceeds a user-specified threshold δ with high probability (i.e. $\theta_1 \geq 95\%$), the semantic unit U is declared to be an execution of the bot command represented by the template.

An exception to the above matching process occurs if U consists solely of *routine* API calls. A bot installed on a host and prepared for interaction typically makes continuous API calls while waiting for commands from the master. These routine calls are easily identified during the first few seconds of bot execution. For instance, for SDbot, the routine API calls were found to be `InterlockedDecrement`, `InterlockedIncrement`, `TlsGetValue`, and `GetLocalTime`. If the system calls in a captured semantic unit are exclusively taken from this set of routine API calls, that semantic unit is regarded as being of lower importance. It is classified simply as belonging to the general command group (see Table I), and no further attempt is made to identify its function more precisely. Examples commands include 'alias', 'nick', etc.

Information logging: If desired, for selected bot commands that are identified (denial of service, spam, etc.), additional information can be recorded about the arguments of API calls that are hooked. This information may be useful for the detection of the victims targeted by an attacker, without requiring monitoring of network traffic. For example, if a bot is identified as executing a SYN flood attack, the arguments of the intercepted system calls will identify the victim, or the C&C server IP address and port number can be recovered when the bot connects to the master. Previous research [15], [16] suggested recording such information. However, BotTee logs and reports the valuable information only when malicious behavior (bot command) of particular interest is identified, rather than for all system calls executed.

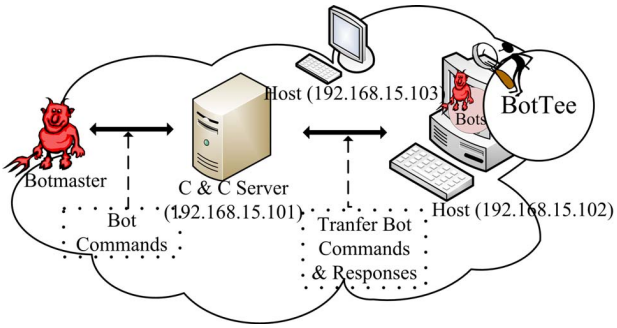


Figure 2. A botnet for experiments

III. Experimental Evaluation

The proposed method was evaluated experimentally. The conditions of the experiment, and the results (measurements of overhead, ability to identify bot commands, and robustness to call obfuscation) are presented in this section. The results demonstrate that BotTee can recognize the execution of specific bot commands in real time.

A. Implementation and Experiments

We implemented a prototype of BotTee; the prototype used the Deviare API [12] for intercepting Windows API calls on the fly. Deviare provides hook libraries to intercept any Windows API calls during runtime. BotTee uses this to obtain API call traces, and to record the arguments of those system calls. For accurate timing analysis, BotTee uses a timer function with a 1 microsecond accuracy, employing performance counter information [17].

A botnet in a private network was deployed, as shown in Figure 2. All machines ran the Windows OS, which is the main target of botnets. As shown in Figure 2, the first host was used as a C & C server, for instance by installing an IRC server like UnrealIRCd.[3] The second host was configured as a vulnerable host or honeypot, which could be exploited for bot recruitment purposes. The third host served as an alternative host, or as a target for an attack. A bot on the

[3]The botnet is based on the IRC protocol. However, our scheme is independent of any particular communication protocol. Our focus is to identify individual bot behavior through execution traces. The configuration for the experiment is just one typical botnet.

second machine immediately connects to the C & C server and joins a predefined channel. When the botmaster issues a command through the C & C server, the bot executes the bot command. BotTee monitors the system call behavior of the bot as it executes bot commands.

Bots from the bot families Agobot, Sdbot, Spybot, Jrbot, Akbot, Dbot, Rbot, and Hellbot were evaluated to validate BotTee. We used current bot source codes from http://securitydot.net. Each of the bot instances came from a different bot family when possible, and we selected one bot from each different bot family. Among 167 available bot source codes, there were 103 variants, including the bots we evaluated. Among them, Agobot, Spybot, Sdbot, and Jrbot are the most popular bot families, as explained in [3].

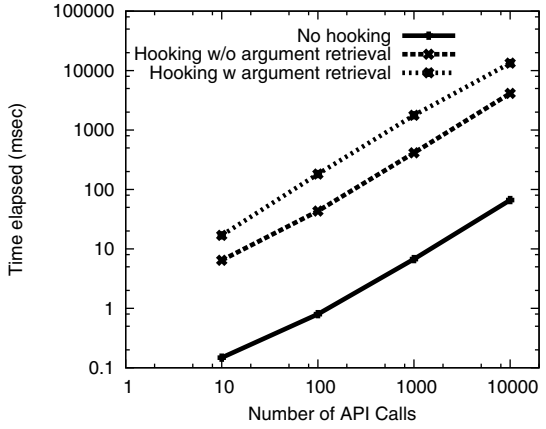


Figure 3. Hooking overhead with increase in the number of API calls. The logarithmic scale is used on both the X- and Y-axis.

To evaluate BotTee, it was required that one bot command with the same functionality be executed by at least four different bots. This allowed construction of each template to be based on the execution of at least three bots. The execution traces of the remaining bots were used to evaluate the effectiveness of the template for identifying bot commands.

In addition, we collected at least 100 execution traces for each bot command in the controlled environment. For these 100 or more traces, the ability of the method to correctly identify the bot command can be evaluated. For instance, to evaluate the method on Agobot, the execution traces from other bots were used to construct semantic templates, which were then used to identify the bot commands executed by Agobot. For all bots, we evaluated the detection rate for different bot commands.

B. Performance Overhead of Hooking

It is important to measure the overhead of hooking, as that determines the practicality of the proposed method. Figure 3 shows this overhead, as a function of the number of intercepted API calls. For normal environments without hooking, there is no overhead. The overhead is greater when hooking also captures the arguments of system calls. The overhead becomes significant only when more than 1000 API calls are intercepted. This motivates the decision to intercept only calls in ComAPI, which results in a smaller performance penalty. We show below this is still sufficient to identify bot commands accurately.

We assume that suspected bot is first detected by conventional means. After this, execution tracing can be turned on to identify which bot commands are being executed.

C. Correlation Results

The first experiment was intended to determine if different bot input commands with the same purpose produce execution traces that are highly correlated. The results are shown in Figure 4(a) and Figure 4(b).

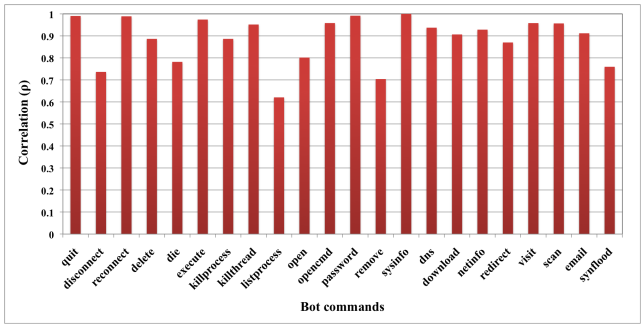
Figure 4(a) shows the average correlation with a probability of $\theta_1 \geq 95\%$ for each bot command, performed by all possible pairs of bots, for repetition factor $\gamma = 2$ (Note: $\gamma = 2$ for all experiments. The longer γ is, the more robust against API call injection. However, the efficiency for real-time matching might be reduced. According to the experiments, $2 \leq \gamma \leq 5$ would be recommended.). For this experiment, the average correlation score for bot commands with the same purpose, but performed by different bots, was 0.88. In the case of 'synflood', 'scan' and 'redirect' commands, which were performed by Jrbot, Spybot, Agobot and Rbot, the average correlation (ρ) was higher than 0.7. In the case of 'dns', 'download', 'visit' and 'email' commands, the average correlation (ρ) was higher than 0.9 All of these commands are closely related to Internet threats, and have a recognizable pattern of system calls to achieve their objective.

Compared with Figure 4(a), Figure 4(b) shows that different bot commands with different purposes performed by a pair of bots have low correlation values (ρ), with an average of 0.52. This further illustrates the accuracy of the proposed command identification method. As expected, very different bot commands will have very low correlation values. For example, the average correlation of 'delete' and 'open' commands was less than 0.3. However, the 'quit' and 'sysinfo' commands have a high correlation value, since most CACTs for the two commands consist of routine (non-distinctive) calls for each bot.

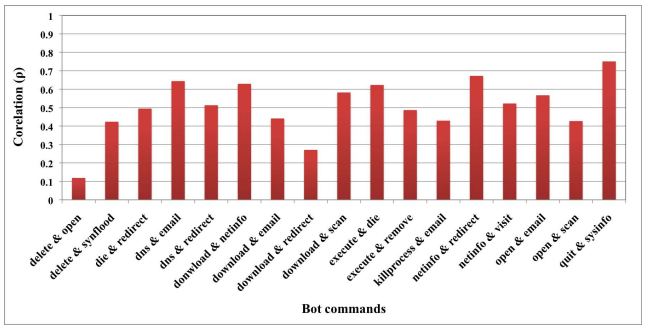
Bot Command	Bot Instance	ρ	θ_1 (%)	k
sysinfo	Agobot, Sdbot	0.87193	97	13
dns	Jrbot, Sdbot	0.99785	99	30
synflood	Jrbot, Rbot	0.99449	99	35
email	Rbot, Jrbot	0.91479	99	52

Table II
RESULTS FROM CORRELATION ENGINE FOR PAIRS OF BOTS (CORRELATION (ρ), PROBABILITY (θ_1 (%)), CACT LENGTH (k))

In addition, Table II shows one example of analyzing commands from pairs of bots. The table includes the



(a)



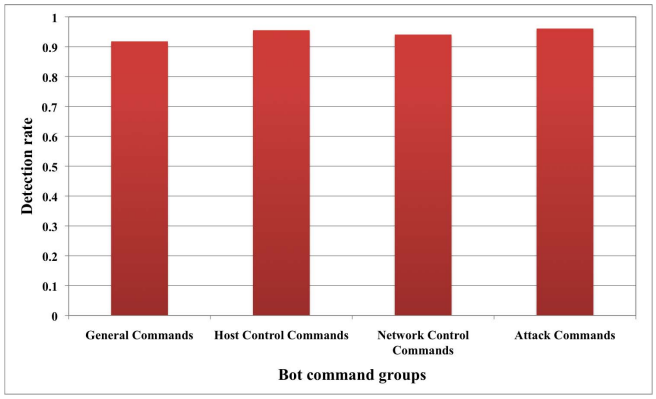
(b)

Figure 4. Average correlation value from the correlation engine. In Figure 4(a), the values are for bot commands with the same purpose, performed by a pair of different bots. In Figure 4(b), the values are for bot commands with *different* purposes performed by a pair of different bots. Each bar is the average values for all the possible execution traces of all the bots.

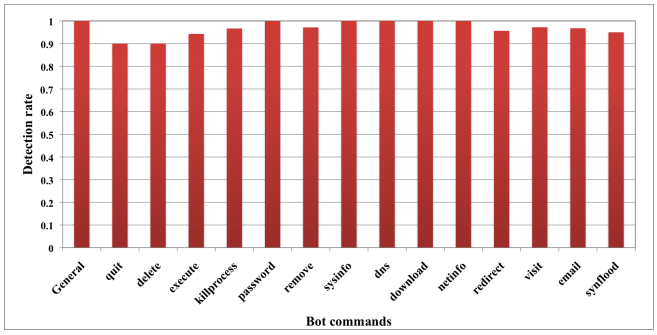
length of the longest common subsequence of API calls, the correlation, and the probability θ_1, for each such pair. As shown in Table II, the bot commands with the same function performed by two different bots have a very high correlation value and a probability (θ_1) that is greater than 95%. Additionally, the CACTs for each command include important APIs for identifying the execution of the bot command. These are termed the *featured APIs*. For example, in the case of the 'dns' bot command, the CACT is `recv, TlsGetValue, GetLocalTime, GetUserDefaultLCID, WideCharToMultiByte, GetTimeFormatA, GetConsoleMode, WriteConsoleA, WriteFile, inet_addr, ..., GetTickCount, InterlockedExchange, CloseHandle, gethostbynam, inet_ntoa, send`, with the length 30.

D. Identification of Specific Bot Commands

BotTee identifies each bot command based on the method described in section II-E. For each group of commands, Figure 5(a) shows that BotTee achieves more than a 95% identification rate (γ = 2); the exception is commands in the general group. In the groups of host control commands and network control commands, BotTee has approximately 95% detection rate. In the group of attack commands, BotTee shows more than 95% detection rate. Above all, commands in the attack command group are identified with a high detection rate. These commands exhibit distinctive system call patterns (type and frequency) that can be distinguished from those of other commands.



(a)



(b)

Figure 5. Detection rate for each command group or for each bot command as shown in Table I. The detection rate is evaluated for γ = 2, probability (θ_1) $\geq$ 95% correlation (ρ) $\geq$ 0.6 (δ).

By each command, Figure 5(b) shows in more detail how well BotTee identifies specific actions of bots on the host.[4] Most bot commands are identified correctly more than 95% of the time. For example, commands such as 'dns', 'download', 'visit', etc., are 100% correctly identified. The 'email' and 'synflood' commands, which also exhibit distinctive call patterns, are identified correctly more than 95% of the time.

Table III shows an example of the logging system in Figure 1 when a bot executes a bot command. In this example, the victim can be a host targeted by DoS attack or a vulnerable mail server to send spam. Other malicious behaviors of interest can be captured by the logging system and used for analysis of the botnet structure and use.

[4]The *general* group, as shown in Figure 5(b) includes all the commands which consist of routine system calls, such as 'id', 'alias', 'version', 'nick', etc., as indicated in Table I.

C&C server IP (Port)	Bot Command	Response	Victim	Behavior
192.168.15.101 (6667)	id	sdbot456	N/A	General
192.168.15.101 (6667)	sysinfo	cpu: 2000MHz. ram: 2086440KB total ...	N/A	Host
192.168.15.101 (6667)	synflood	SynFlooding: 192.168.15.103 port: 12345 ...	192.168.15.103 (1234)	**Attack**
192.168.15.101 (6667)	email	email sent to test2@localdomain.net	192.168.15.103 (25)	**Attack**

Table III
AN EXAMPLE OF LOGGING ARGUMENTS IN THE INTERCEPTED API FUNCTIONS. THE EXAMPLE IS DERIVED BY EXPERIMENTS IN SECTION III. THE LAST ENTRY (BEHAVIOR) IN THIS TABLE REPRESENTS EACH BOT COMMAND GROUP IN TABLE I.

E. False Identification

Using the templates derived from monitored bot execution, we evaluated the number of times that bot commands were incorrectly identified as being executed by non-bot software. Table IV shows the results. In the majority of cases, the length of the CACT shared by the non-bot application and the template was less than 10.

In two cases, the non-bot program trace was identified as executing the bot command 'download' after comparison with the template database. The CACT lengths in these two cases consisted of the initial 9 (Internet Explorer) or 13 (Winscp) system calls in the template for the bot 'download' command, which are evidently not distinctive enough to differentiate bots from non-bot programs. In point of fact, some commands executed by bots may well be quite similar to the actions taken by non-bot programs. Our assumption is other means are used discriminate bots from non-bots, so this isn't a serious drawback. In all other cases (including some not shown in Table IV), non-bot programs had low correlation with the bot templates, and were not identified as executing bot commands.

F. Detection Rate with API Call Injection Attack

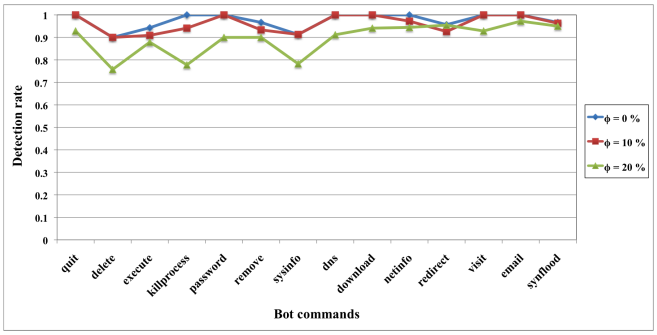


Figure 6. Detection rate for API call injection attacks.

If an attacker wishing to obfuscate the execution of bot commands introduced arbitrary system calls during command execution, there should be little impact on the proposed detection method. This is because many types of system calls are ignored by the correlation engine. A more serious attack would be the injection of ComAPI calls. Let the *attack rate* ϕ be defined as the fraction of ComAPI calls which are injected by the attacker for obfuscation purposes. In other words, some API calls in the complete set of ComAPI calls are intentionally injected into execution traces as much as ϕ. For instance, if the attack rate is 10%, one tenth of the hooked API calls are injected by the attacker for obfuscation purposes. Such injection may be intended to obfuscate timing analysis and correlation as well. In general, one bot command may generate an execution trace with several hundred hooked system calls, so a 10% attack rate may obfuscate execution traces significantly.

For purposes of evaluation, we launched injection attacks on bot execution, and compared with templates that gave high detection rates for each bot command, as shown in Figure 5(b). Figure 6 demonstrates the performance of BotTee under these conditions. As shown in the figure, BotTee shows a high detection rate for network-related attack commands, up to an attack rate of 20%. In the case of bot commands with specific patterns, such as 'email' and 'synflood', BotTee identifies such bot commands well, regardless of the attack rate. On the other hand, for commands having a short CACT, the detection rate is decreased when the attack rate reaches 20%. Such commands are classified into a general group, or are not involved in network threats, so accurate and exact identification is likely to be of lower priority. For bot commands which can be involved in network threats, such as 'dns', 'scan', 'visit', 'email', 'synflood', etc., BotTee is robust to call insertion attacks.

IV. DISCUSSION

The more bots that are observed, using techniques such as the one proposed in this paper, the more accurately that botnet-driven network threats can be identified. This can be facilitated by virtualization of hosts susceptible to bot infection on honeypots. For example, the Potemkin Honeyfarm system [18] provided highly scalable virtual honeypots based on special purpose gateways and a virtual machine monitor derived from Xen. This work emulated hundreds of thousands of IP addresses, using only a handful of physical servers.

BotTee can specify victims targeted by active botnets and infer the overall behaviors of the active botnets through closely monitoring bots activities without analyzing network traffic. To do so, we should allow the bots to communicate with botmasters for at least several hours [10]. During this time, the bots should not be allowed to infect a network of hosts. Even though honeypots are an essential decoy system,

Application	Description	Bot Command	Correlation (ρ)	Probability (θ_1 (%))	CACT Length (k)	Matching
Internet Explorer	Downloading a file	download	0.7713	97.49	9	Yes
	Clicking other links in the browser	visit	0.0225	14.28	52	No
Winscp	Downloading a file	download	0.8727	99.97	13	Yes
	Connecting the server	connect	0.8957	89.57	5	No
Outlook Express	Sending email	email	0.2529	51.92	25	No
Notepad	Executing notepad.exe on a local host	execute	0.8185	90.98	6	No
PuTTy	Connecting a FTP server	connect	0.5564	66.99	6	No
	Downloading a file from a FTP server	download	0.6573	92.35	9	No
	Running 'ping' command	synflood	0.3692	52.87	7	No

Table IV
RESULTS ON NON-MALICIOUS APPLICATIONS. THE BOT COMMANDS IN THIS TABLE COMPARABLY CORRESPOND TO THE ACTIVITIES OF THE BENIGN APPLICATIONS.

it is dangerous to let bots communicate through a honeywall [16], which only limits rates of communication. Since bots do not produce much traffic, and stealthily communicate with a botmaster, an intelligent system is required to control bot behavior while communicating. This can be achieved by BotTee through recognizing individual bot behaviors. By controlling specific bot actions, we can securely monitor bot behaviors as long as needed. The hooking technique allows potentially malicious bot commands to be replaced by more benign actions, or to be thwarted.

Due to the overhead of hooking, it is not practical to intercept all API calls at runtime. The above results demonstrate that the set of calls in ComAPI is sufficient to recognize individual bot behaviors, with little performance degradation. Through considering the arguments of API calls, BotTee can be advanced to control current active bots with high accuracy of the behavior identification. In addition, by using a compact finite state automaton approach [19], we can make BotTee more practical and robust against severe obfuscation. By using such an approach, without making semantic templates, BotTee can identify individual actions through dynamic specification.

V. RELATED WORK

Many network-based botnet detection schemes have been proposed in recent years. Felix [1], Evan [7] and Moheeb [2] investigated botnet dynamics. Felix [1] suggested a method to mitigate DDoS attacks from botnet by shutting down a centralized C&C server. Moheeb [2] thoroughly examined botnet behaviors by tracking IRC botnets through IRC protocol and DNS tracking techniques. All the above works employed honeypots to collect bot binaries that infiltrate active botnets on the Internet. In addition, David [5] studied global diurnal behaviors of botnets through a DNS sinkholing technique. Guofei proposed three schemes to observe network-level botnet behaviors for the detection of C&C channels: BotHunter [20], BotSniffer [6] and BotMiner [21]. Rishi [8] proposed a signature-based IRC botnet detection system by identifying IRC bot nickname patterns and IRC NICK messages which have a possibility to come from a bot. An anomaly-based botnet detection algorithm [22] was presented by combining IRC statistics and TCP work weight. In addition, a machine learning based botnet detection utilizes some general network-level traffic features of chat-like protocols like IRC. Except for BotMiner [21], most works have focused on IRC-based botnet. By contrast, BotTee deals with characteristics of bots and botnets at the host-level. It is independent of the network protocol used, or the employment of encryption, and is not specific to one bot family.

Several techniques have been presented for host-based bot *detection* by using Detours [12]. The research of application behaviors through system calls has been of interest for years [23]–[25]. Botswat [9] is a host-based behavioral bot detection system based on the Detours. It also analyzes the behavior of installed bots to distinguish malicious bots from benign processes by checking whether actions of bots are remotely initiated or not. Botswat investigated the number of distinct system calls invoked during a successful execution of bots, through source code inspection. BotTracer [10] is another host-based bot detection method that uses known bot characteristics: automatic startup, C&C channel establishment, and some specific attacks (information dispersion/harvesting). It uses system level activities to detect bots through only a few system calls related to disk access and process memory, which is not enough to recognize specific bot system-level activities. Additionally, BotTracer heuristically identifies the point of DDoS if the bot tries to connect another server to reply with a result. However, this situation can frequently happen in a botnet, such as the 'server' command, the 'scan' command, and the 'download' command, etc, as shown in Table I. By comparison, BotTee can point out the victims targeted by a current bot through the identification of bot commands. Lorenzo and et. al proposed a layered architecture for the detection of malicious behaviors [26]. They made a hierarchical behavior graph through tainting analysis in data control flow. They identified several malicious behaviors globally, such as proxying, keystroke logging, data leaking, and downloading. However, our approach recognizes more specific bot behaviors, including all common network threats, such as DoS, spam, click fraud, proxy, scan, etc. In addition, our approach is simpler

and more efficient than taint analysis.

VI. Conclusion

This paper proposes a method for identifying the high-level commands being executed by a bot, in real time. Such a capability is useful for analysis of bot activities, intended victims, and control structure. The method is based on hooking of selected system calls, and comparison of the resulting traces with a previously-captured set of bot command templates. The comparison itself involves computation of a least common call trace subsequence, and correlation of call trace timing.

This method was implemented in a system called BotTee. BotTee was evaluated in a private network with actual bot instances. Under these conditions, BotTee successfully identified important bot commands from system call traces. This held true even for commands executed by bots from other bot families, and not represented during construction of the command templates. BotTee is therefore useful for analysis of bot instances not yet seen. With proper threshold selection, BotTee does not mis-identify commands. In addition, BotTee is relatively robust to call injection attacks.

BotTee does not rely upon analysis of network traffic and is independent of the control protocol used. This approach may be combined with methods that examine network traffic, for better performance than each alone can achieve. Further, logging and inspection of the arguments of captured system calls can shed further light on the attacker's intentions.

References

[1] F. C. Freiling, T. Holz, and G. Wicherski, "Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks," in *10th European Symposium on Research in Computer Security*, 2005.

[2] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis, "A Multifaceted Approach to Understanding the Botnet Phenomenon," in *6th ACM SIGCOMM conference on Internet measurement*. ACM, 2006.

[3] P. Barford and V. Yegneswaran, "An Inside Look at Botnets," in *Special Workshop on Malware Detection, Advances in Information Security*, 2006.

[4] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis, "My Botnet is Bigger than Yours (Maybe, Better than Yours): why size estimates remain challenging," in *First Workshop on Hot Topics in Understanding Botnets*, 2007.

[5] D. Dagon, C. Zou, and W. Lee, "Modeling Botnet Propagation Using Time Zones," in *Network and Distributed System Security Symposium*. The Internet Society, 2006.

[6] G. Gu, J. Zhang, and W. Lee, "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic," in *15th Annual Network and Distributed System Security Symposium*, 2008.

[7] E. Cooke, F. Jahanian, and D. McPherson, "The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets," in *the Steps to Reducing Unwanted Traffic on the Internet Workshop*. USENIX Association, 2005.

[8] J. Goebel, "Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation," in *First Workshop on Hot Topics in Understanding Botnets*. USENIX Association, 2007.

[9] E. Stinson and J. C. Mitchell, "Characterizing Bots' Remote Control Behavior," in *International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment*, 2007.

[10] L. Liu, S. Chen, G. Yan, and Z. Zhang, "BotTracer: Execution-based Bot-like Malware Detection," in *11th Information Security Conference*, 2008.

[11] Q. Zhang, "Polymorphic and Metamorphic Malware Detection," Ph.D. dissertation, North Carolina State University, 2008.

[12] "Hooking," http://en.wikipedia.org/wiki/Hooking.

[13] K. Jain and R. Sekar, "User-level infrastructure for system call interposition: A platform for intrusion detection and confinement," in *Network and Distributed Systems Security Symposium*, 2000.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, second edition*. MIT Press and McGraw-Hill, 350-355. ISBN 0-262-53196-8, 2001.

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

[16] "The Honeynet Project," http://www.honeynet.org/.

[17] "Obtaining Accurate Timestamps under Windows XP," http://www.lochan.org/2005/keith-cl/useful/win32time.html.

[18] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage, "Scalability, fidelity, and containment in the potemkin virtual honeyfarm," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 148–162, 2005.

[19] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A Fast Automaton-based Method for Detecting Anomalous Program Behaviors," in *IEEE Symposium on Security and Privacy*, 2001.

[20] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, "BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation," in *16th USENIX Security Symposium*, 2007.

[21] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection," in *17th USENIX Security Symposium*, 2008.

[22] J. R. Binkley and S. Singh, "An algorithm for anomaly-based botnet detection," in *2nd conference on Steps to Reducing Unwanted Traffic on the Internet*. USENIX Association, 2006.

[23] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-Aware Malware Detection," in *IEEE Symposium on Security and Privacy*, 2005.

[24] D. Wagner and R. Dean, "Intrusion Detection via Static Analysis," in *IEEE Symposium on Security and Privacy*, 2001.

[25] Q. Zhang and D. S. Reeves, "MetaAware: Identifying Metamorphic Malware," in *23rd Annual Computer Security Applications Conference*, 2007.

[26] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, "A layered architecture for detecting malicious behaviors," in *11th international symposium on Recent Advances in Intrusion Detection*, 2008.