Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2015, Article ID 524627, 17 pages
http://dx.doi.org/10.1155/2015/524627

*Research Article*

# Overhead Analysis and Evaluation of Approaches to Host-Based Bot Detection

**Yuede Ji,[1,2] Qiang Li,[1,2] Yukun He,[1,2] and Dong Guo[1,2]**

[1] *College of Computer Science and Technology, Jilin University, Changchun, Jilin 130012, China*
[2] *Symbol Computation and Knowledge Engineer of Ministry of Education, Jilin University, Changchun, Jilin 130012, China*

Correspondence should be addressed to Qiang Li; li_qiang@jlu.edu.cn

Received 26 January 2015; Revised 4 May 2015; Accepted 4 May 2015

Academic Editor: José Molina

Host-based bot detection approaches discover malicious bot processes by signature comparison or behavior analysis. Existing approaches have low performance which has become a bottleneck blocking its wider deployment. Among the impact factors of performance, overhead is a crucial one. Many host-based bot detection approaches with high detection accuracy are not used practically because of their high overheads. For the development of host-based bot detection, unveiling the factors affecting the overhead is very significant. First, this paper classifies the typical approaches of host-based bot detection proposed in recent years by several metrics, information sources, interception mechanisms on host, intercepted system calls, trigger mechanisms, and correlation engine. Second, based on our analyses of aims and implementations of detection approaches, we identify three major factors affecting the overhead of approaches, namely, interception mechanism on host, type, and number of system calls intercepted and correlation engine. Third, we evaluate the influence of these factors via various experiments on real systems. Finally, based on the experiments, we propose several suggestions which are able to significantly decrease the overhead of host-based bot detection approaches.

## 1. Introduction

Malicious code (virus, Trojan, worm, spyware, botnet, etc.) has become a serious threat to the Internet. As an important class of malicious code, bot and botnet have a more hidden attack mode thus cause more serious threats [1] compared with other malicious codes. Bot is an instance of malicious code running on the victim host. It can hide itself, steal user privacy, and launch other malicious activities. A large scale of compromised hosts forms a botnet through the Command and Control channel (i.e., IRC, HTTP, P2P, etc.) under the control of a botmaster. Botmasters leverage botnets to conduct various cybercrimes such as spreading attack code and command, DDoS attacks, spamming, deploying spyware, and phishing. Botnet has become the major platform for most online criminal activities.

Up to now, a large number of researches have been carried on regarding the detection and defense of bot and botnet; however there still exist various challenges. Since bot and botnet can hide in the system, update attack mechanism to bypass the conventional detection techniques, and change communications from centralization to distribution, the stimulus of economic benefits and modularization of bot code promote the study of new attack methods against detection techniques and accelerate the generation of custom botnet software and its variants. Consequently, eliminating the huge threat caused by bot and botnet to end-host and network has become one of the most urgent tasks for researchers.

According to the execution location, existing bot and botnet detection approaches can be divided into two categories, network-based approaches and host-based. Network-based detection approaches only concern network traffic. The effectiveness of these approaches will be reduced when communication protocol of botnet changes and communication content is encrypted, or they are in a high-speed and large-scale network. In addition, these approaches cannot completely eliminate the threats of botnet, and the evidence of bot and botnet is not easy to keep. Host-based bot detection is more effective for protecting host security. It can monitor the behavior of suspicious process on the host even without prior knowledge. It can also capture the attack information before the botnet traffic being encrypted. Compared with network-based botnet detection, it can discover more insights of unknown bots especially when there is only a single bot in a local network. And, more importantly, we may completely eliminate the bot if we successfully detect it on end-host.

Host-based bot detection approaches are mainly classified into 5 categories: (1) capture the flow of infected host and determine whether it is a bot host according to predefined signatures of bots' various stages, such as BotHunter [2]. The disadvantage of this approach is that it cannot detect unknown bots. (2) Analyze the binary codes of suspicious programs and detect whether it is a bot by extracting features and sequences of code [3]. (3) Analyze abnormal behaviors of host, such as running status of operating system and suspicious processes, activities launched by bot instance, sequences of API calls. Based on these analyses, some approaches determine whether it is bot host through statistical anomaly, artificial immune systems, or correlation algorithms [4–7]. (4) Discover bot host through monitoring network connection and memory status, by manual processing and network tools [8]. (5) Use traditional host antivirus software. Antivirus software needs accurate malicious code library; however a bot can easily update itself through botnet to evade signature inspection. According to statistics [7], even with the latest antivirus software, there are still 22.97% hosts infected by malicious code.

However, some common challenges also exist in these host-based detection approaches: (1) the effectiveness of existing abnormal detection and behavior detection approaches is not enough to detect unknown bots. In most cases, a lot of prior knowledge needs to be trained and defined in this class of approaches; thus both false positive rate and false negative rate will increase when new bots occur. (2) Bot detection is a multifaceted and multiphased process. Due to the lack of information of local network anomaly and other victims' similar behaviors, misjudgment often occurs when detecting bots in a single host. (3) Detection approaches based on the abnormal behaviors usually intercept a large number of system calls and launch complex real-time or sliding-window calculation and thus consume a large portion of CPU and memory resource. In particular when many benign programs are running, there is a significant increment of host overhead which thus affects users' normal use. Accordingly, the overhead of host-based bot detection approach has become a bottleneck blocking its wider deployment. If we can reduce its overhead to an acceptable level, it will greatly facilitate the development of host-based bot detection approaches.

This paper studies the factors affecting the overhead of host-based bot detection approaches and proposes appropriate suggestions. The main contributions are stated as follows:

(1) We classify the typical approaches of host-based bot detection proposed in recent years by several metrics: key assumptions, host environments, information sources, interception mechanism on host, classification of behavior and alert, ways of constructing databases, trigger mechanism, and correlation engine. We compare and analyze the characteristics of host-based bot detection approaches through these metrics.

(2) We make deep analyses of the aim and implementation of the detection approaches. We identify three major factors affecting hosts' overhead, namely, interception mechanism on host, type and number of system calls intercepted, and correlation engine.

(3) We select four typical approaches to evaluate the influence of these factors via various experiments on real systems. We summarize and discuss the affecting factors and suggestions based on our experimental results. We discover that using Windows Hook or Detours to intercept has a smaller impact on host overhead, specially selected collection of system calls such as Common API has a tolerant overhead, and correlation engine should be carefully deployed and optimized according to the characteristics of the corresponding correlation algorithm.

The remainder of this paper is outlined as follows: Section 2 classifies the typical approaches of host-based bot detection proposed in recent years by several metrics. Section 3 makes deep analyses of detection approaches' aim and implementation. Section 4 selects some typical detection approaches to evaluate these factors via detailed real system experiments. Section 5 discusses the limitations and future works. Finally, Section 6 concludes the paper.

## 2. The Classification of Existing Approaches

Currently, primary host-based bot detection approaches include (1) BotSwat [9], proposed by Stinson and Mitchell, which can distinguish between bot behaviors and benign programs through judging whether the input data of commands executed in the host is received from the network. (2) BotTracer [10], proposed by Liu et al., is to judge bot infection from the three indispensable stages in the process of bot attacking. (3) Martignoni uses hierarchical behavior graphs [11, 12] to detect malicious behaviors. (4) BotTee [4], proposed by Park and Reeves, extracts the suspicious system call sequences to match with the bot command patterns. (5) Kolbitsch et al. use intercepted system call sequences to match malicious behavior graphs [13], and judge whether the behavior is malicious based on the matching results. (6) Al-Hammadi et al. use Dendritic Cell Algorithm (DCA) [14, 15] for host-based bot detection to detect botnets on host, based on the knowledge of Artificial Immune Systems [16]. (7) Zeng et al. first propose the idea of combining both network-level and host-level information to detect botnets [5]. After combining host-level and network-level information, they can find the infected hosts in Local Area Network (LAN). (8) JACKSTRAWS [17], proposed by Jacob et al., uses machine learning to identify C&C connection accurately. (9) Shin et al. propose the EFFORT approach

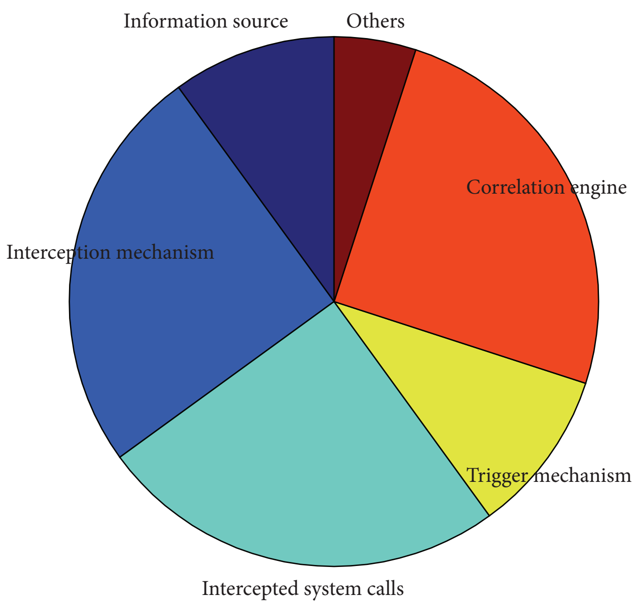


Figure 1: Evaluation metrics affecting host overhead.

[18], correlating multiple modules of a process to generate the final result to determine whether it is infected. (10) PeerPress [19], proposed by Xu et al., combines host-level and network-level information to proactively detect P2P malware. Host-level dynamic binary analysis automatically extracted Malware Control Birthmarks (MCB) of P2P malwares. Then network-level scanner probes the hosts in the network to detect whether they are infected by P2P malwares.

To better view these approaches, especially the factors related to host overhead, we propose 5 important evaluation metrics to evaluate them. These metrics have different impact on the overhead of these approaches as shown in Figure 1. Through a comparative analysis of these evaluation metrics, we can have a more in-depth understanding of host-based bot detection approaches and the factors affecting the overhead of these approaches.

*2.1. Classification Metric 1: Information Sources.* There are 3 information sources: host, network, and host and network. Studying the information sources can help us understand the impact of them on bots detection better. In this paper, we only discuss the approaches of host-based bot detection, so network-based botnet detection is out of scope. In this case there are 2 primary information sources: the host and the host and network.

The approach of Zeng [5] combines host and network information. The information on host is a feature vector through monitoring registry, file system, and network stack, while the information on network is a feature vector of the network behaviors by analyzing the NetFlow of the edge router in LAN. Then they combine the host and network information through clustering and correlation to determine which host is infected.

Other approaches use the internal information on hosts. In the approach of Martignoni [11], the host information refers to the information of users' input and all running processes in emulators, including malicious information and benign information. The information of BotTee [4] is from host and divided into two classes. One class is that, while generating templates, the information needed is the previously detected bots' behaviors on hosts. The other class is that, while detecting, the information comes from the running processes on hosts, including both malicious and benign ones. In the approach of Kolbitsch [13], the host information refers to the system calls on hosts. They only concern about the system calls that may be used to launch malicious activities; that is, they only need malicious information. For Al-Hammadi's approach [14], the information source refers to the function calls and system calls on hosts, including both benign and malicious ones. In BotSwat [9], host information mainly refers to abnormal system calls and users' normal input on hosts. It also includes network data monitored on host, which is used to judge the infected memory area. In BotTracer [10], host information mainly refers to automatically started processes' information in virtual machine, and it also includes traffic information of inbound and outbound on hosts. For JACKSTRAWS [17], the host information is malicious system calls and traffic information. In EFFORT [18], the information on hosts includes human operations and system calls which focus on the suspicious processes, and it also includes network connection information on hosts.

Only using host information, including processes information, information of system call sequences, and network connection information on host can effectively detect bots on host. It can also terminate or kill the suspicious process. However, this class of detection only focuses on one host, neglecting other hosts' information in LAN. Due to the fact that bots are very easy to spread, it is highly possible that there are many infected hosts at the same time in LAN. So utilizing network information can make detection more accurate. The approach of Zeng [5], which combines host and network information, also has some disadvantages. For instance, the hosts focus on statistics, so the bot process cannot be found. What is more, even if the final detection result shows that the host has been infected, we can do nothing towards the bots. However, this approach is the first one to combine host information and network information. The information selected and correlation methods are all in an exploration stage. As a result, whether it is more effective to detect bots and whether we can deal with the bot's process on hosts will be our future concern.

*2.2. Classification Metric 2: Interception Mechanisms on Host.* Host-based bot detection approaches use interception mechanism to obtain information on host. They get information about all aspects of hosts through intercepting preselected system calls. The difference between interception mechanisms reflects in the realization of programs. Some use Windows Hook to intercept; some use third-party libraries, such as Detours [20] and Deviare API [21], to intercept; others use tools which have been packaged like Process Monitor [22] to intercept. The major difference is reflected in the effectiveness of interception and the impact on host overhead. Therefore,

Table 1: Interception mechanisms on host.

| Approach | BotSwat | BotTracer | Martignoni | BotTee | Kolbitsch | Al-Hammadi | Yuanyuan Zeng | JACK-STRAWS | EFFORT |
|---|---|---|---|---|---|---|---|---|---|
| Interception mechanism | Detours | Detours | Windows Hook | Deviare API | Anubis | APITrace | Process Monitor | Anubis | Windows Hook |
| System calls intercepted | Up to 2,200 API functions | A limited number of Win32 functions | sysenter, sysexit | Common API | A subset of interesting system calls | Selected API calls | All system calls | Winsock API | Windows system calls related to keyboard/mouse events |

it is very important to choose a reasonable interception mechanism.

As shown in Table 1, BotSwat [9] uses Detours library provided by Microsoft to intercept library calls and system calls. BotTracer [10] also uses Detours library to intercept. The approach of Martignoni [11] uses Windows Hook to intercept. BotTee [4] uses Deviare API [21] to intercept system calls on host. Kolbitsch's [13] approach uses dynamic malware analysis environment Anubis [23] to monitor system calls. In the approach of Al-Hammadi [14], it uses APITrace [24] to intercept. The approach of Zeng [5] uses an approach similar to Process Monitor [22] to intercept system calls at registry, file system, and network stack on hosts. The approach of JACKSTRAWS [17] uses Anubis to monitor network behaviors. EFFORT [18] uses Windows Hook to intercept system calls.

*2.3. Classification Metric 3: Intercepted System Calls.* Although all the approaches intercept system calls, the type and number of intercepted system calls are different from each other. As shown in Table 1, BotSwat [9] intercepts library calls and system calls. They intercept up to 2,200 APIs and it has a great impact on host overhead. BotTracer [10] intercepts system calls related to memory and disk access because information harvesting/dispersion have to access the disk or memory. The approach of Martignoni [11] hooks the sysenter instruction, which is the begining of kernel calls, and the sysexit instruction, which is the end of kernel calls, so that the emulator can pass the monitored event stream to behavior matcher in real time. This approach only intercepts two instructions; thus the overhead is small. BotTee [4] intercepts a subset of all system calls called Common API. System calls in this subset are indispensable for the execution of bots. Kolbitsch's [13] approach intercepts a subset of system calls which can be used to execute malicious activities. These system calls are relevant to security and they are in the bottom of behavior graphs. Al-Hammadi [14] intercepts selected system calls to generate log files. The approach of Zeng [5] intercepts system calls at registry, file system, and network stack on hosts. The approach of JACKSTRAWS [17] hooks Winsock API to obtain the data from network. EFFORT [18] hooks Windows system calls related to keyboard/mouse events.

Interception mechanism plays an important role in obtaining host information; however the type and number of intercepted system calls are different. Currently, intercepting all system calls has a high impact on system overhead; thus it is not suitable for present approaches. Consequently, under the condition of not decreasing detection accuracy, most of them just intercept several, a set or one class of system calls to reduce the number of intercepted system calls.

*2.4. Classification Metric 4: Trigger Mechanism.* In order to reduce the impact on host overhead, detection approaches do not run all the modules in real time. Generally, there are one or several monitor modules running in the system. If there are accordant trigger events or system calls, the monitor module will trigger the correlation engine to detect. Based on above approaches, there are two information sources in trigger processes: the information on host and the information on network.

Some approaches [4, 9–11, 13, 14, 17, 18] trigger the detection approach by the information on host. In the approach of BotSwat [9], the user input component and tainting component always run in the background to continuously monitor the user input events and the tainting from network. When special system calls and parameters are monitored, the behavior-check procedure will be triggered. Martignoni's [11] approach intercepts the sysenter and sysexit and generates an event flow when the monitored process terminated. The monitor will pass this event flow to behavior matcher to trigger other modules. When BotTee [4] finds a certain process calling the system calls within the range of interception, it will generate a system call sequence and trigger the other detection modules. Kolbitsch's [13] approach uses scanner to monitor the system calls and parameters to generate the analyzed behavior graphs and trigger the correlation engine for match. In BotTracer [10], after starting the virtual machine, the processes automatically started on host will be found. After filtering out the processes on the white list, BotTracer [10] will continue to monitor the remaining processes. When finding outgoing traffic from any remaining process, this process will be flaged as suspicious. Then a traffic model will be established to trigger the match of C&C model. When the approach of Al-Hammadi [14] is detecting IRC bots and if there is one process on host trying to connect to the IRC server via IRC standard port, the process will be intercepted and the relevant information of this process will be recorded into SigLog or AntiLog, which are the signal log and the antigens log. Then the two log files will be passed to other modules to trigger detection approaches. Through monitoring Winsock API, the approach of JACKSTRAWS [17] will generate the monitored behavior graphs and then match them with the templates in template database to trigger other modules. In EFFORT [18], through the analysis of human operations, network connections, and the processes by human-process-network correlation analysis module, they can find out network connection processes driven by bots. Then the other three modules will be triggered to detect.

The approach of Zeng [5] triggers the detection approach by the network information. In this approach, the host monitor runs in the background to monitor host behaviors at registry, file system, and network stack. The host suspicion level generator calculates the overall suspicion level based on the behaviors in a certain period of time. At the same time, the network analyzer analyzes the Netflow in a certain period of time and extracts network features for cluster analysis. When a cluster is found, the information will be sent to the correlation engine to trigger it. According to the cluster information provided by network analyzer, the correlation engine sends request for the suspicion level and features of each host in this cluster and then generates a detection result.

*2.5. Classification Metric 5: Correlation Engine.* After obtaining the host and network information, we need to process the information comprehensively and generate the final detection results. This comprehensive processing mechanism is called correlation engine. There are three ways to process information in correlation engine: matching the collected information with the database, using the approaches of statistics and using self-defined correlation rules.

Through obtaining information and triggering detection approaches, the correlation engine will deal with the collected data and then match them with the previous database to determine whether the host is infected. There are many approaches using this way to correlate, such as JACKSTRAWS [17], the approaches of Martignoni [11], and Kolbitsch [13]. The approach of Martignoni [11] uses behavior matcher as the correlation engine to match incoming event streams with behavior graphs from the bottom. According to the match rules, if successful, behavior matcher will generate an event, which can be used in higher level of behavior graphs. And when the incoming event stream matches with a high-level behavior graph successfully, it means a bot process is detected. For Kolbitsch [13], the approach uses scanner to analyze the system calls of suspicious processes and then generates system call sequences of this progress. Then the scanner will match the system call sequences with the behavior graphs in malicious behavior graph sets. Based on their match rules, if a match is found, this process is malicious and it will be terminated. JACKSTRAWS [17] makes the programs run in a sandbox and extracts the behavior graph to match with all C&C templates. If the matching value reaches a certain threshold, it means that there is a match and the relevant connections will be detected as C&C connections. They use maximum common subgraph (mcs) to calculate the distance between two graphs.

Some approaches process the collected information through the statistical approach, and the result is used to judge the suspicious hosts [4, 5, 14]. For example, the approach of Al-Hammadi [14], in the final analysis stage, will calculate the MCAV value of each antigen type according to the formula. Based on the MCAV, they design an enhanced coefficient, called MAC, on which they judge whether an antigen is malicious or not. In the approach of Zeng [5], the host analyzer will pass the degree of suspicious and the statistic data of network characteristics to the correlation engine when a cluster of suspicious hosts have been recognized by network analyzers. Then the correlation engine will generate a detection score and a detection result for each host by using the correlation methods. According to this result, whether the host has been infected could be determined. BotTee [4] uses the method of match and statistics. At first BotTee [4] passes system calls to the correlation engine. The correlation engine will calculate the results and pass them to the semantic behavior matcher according to LCS and statistical methods. Then the semantic behavior matcher will match the results with the templates in database. When the optimal match reaches a certain threshold, it will be considered as bot behavior, otherwise benign behavior.

Some approaches use user-defined correlation rules, which deal with the collected data and then generate a detection result to determine whether the host has been infected [9, 10, 18]. The correlation engine of BotSwat [9] is an approach of behavior detection. When it is triggered, tainting component and user input component will be queried. Tainting component can provide the information about whether a particular memory region is considered tainted. User input component can provide the information about whether the data value or memory region is considered clean or whether a syscall invocation is likely the result of user input. Then according to this information and predefined correlation rules, they can determine whether the process is malicious. There are three steps in BotTracer [10]. Every step is trying to remove benign process from the suspicious process. The third step monitors the processes selected from the first two steps and traces the relevant system calls and parameters. Then based on the flow mode they can determine whether the suspicious process performs information harvesting or dispersion. And thus they can determine whether the suspicious process is bot. The correlation engine of EFFORT [18] will collect the results of the three previous modules, weighting these three results by a weighted voting system to generate a final result. Therefore, they can determine whether the process is malicious or not.

## 3. Overhead Analysis

In order to identify the factors affecting approaches' overhead, we select three typical detection approaches to deeply analyze their flow and time complexity. They are Stinson's BotSwat [9], Younghee Park's BotTee [4], and the approach of Zeng [5].

*3.1. Flow Analysis.* There are many factors, such as the steps, the operations, and the calculation of detection approaches, affecting the delay of the detection of suspicious behaviors. And these factors determine whether the detection approaches can quickly and accurately detect suspicious behaviors. Thus analyzing the flow path of detection

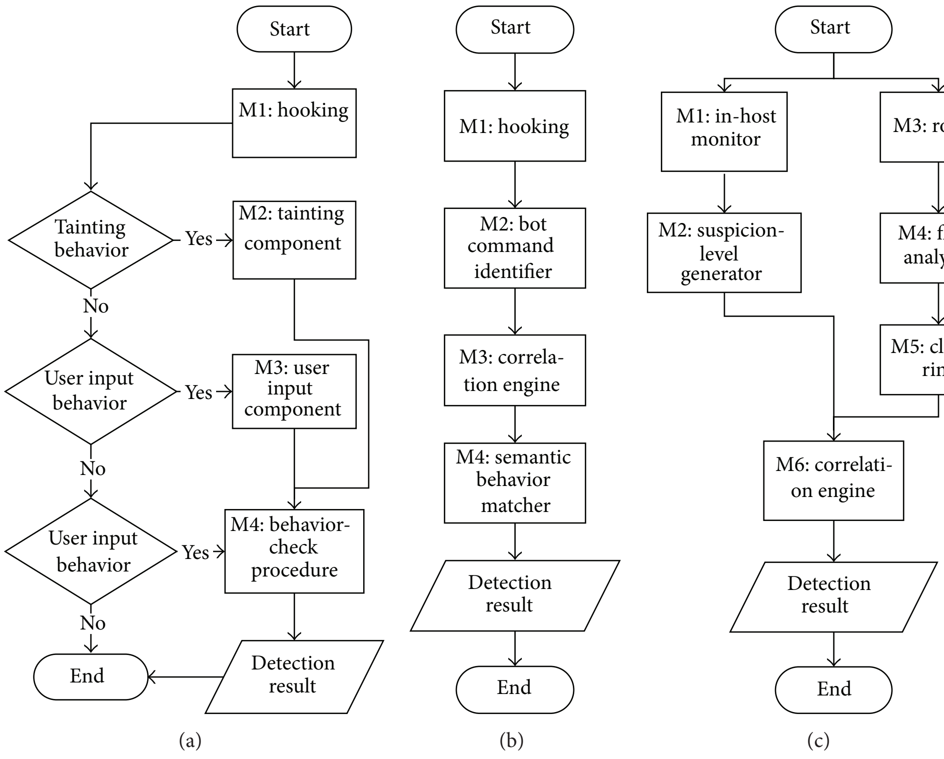


Figure 2: Flow charts of the three approaches.

approaches are significant for the decrease of overheads. We select three from the above approaches to compare and analyze: Stinson's BotSwat [9], Younghee Park's BotTee [4], and the approach of Zeng [5].

Figure 2 is the flow charts of the three detection approaches: (a) represents BotSwat, (b) represents BotTee, and (c) represents the approach of Yuanyuan Zeng.

BotSwat [9] intercepts up to 2,200 API, according to the class the API belongs to; they can separately trigger the Tainting Component M2, User Input Component M3, and Behavior-Check Procedure M4. When the information from network, which are system calls related to the behavior of network receiving, are intercepted, M2 will be triggered and will trace the behaviors of the network, and then the memory to which network behaviors has written will be marked taint. M2 exports an interface that enables querying whether a particular memory region is considered tainted. When the intercepted system calls belong to user behaviors or application behaviors controlled by users, M3 will mark the related data and memory as clean, and it will export an interface to query whether a data value or memory region is considered clean or whether a system call is likely the result of user input. When intercepted system calls belong to the selected system calls, it will trigger M4, and then M4 will determine whether to mark them as external control through querying M2 and M3.

BotTee [4] uses Deviare API [21] to intercept Common API. When intercepting recv and send, Deviare API will trigger Bot Command Identifier M2, which will analyze the system call sequences between recv and send to generate a set of execution traces and each execution trace is called a semantic unit. Then these execution traces will be passed to the correlation engine M3, which will analyze these sematic units to generate the command templates. Finally, the generated command templates will be passed to semantic behavior matcher M4, and M4 will match the command templates with the templates in the database to determine whether it is bot command.

Zeng et al. [5] combine host information and network information to detect bot. They use Process Monitor to monitor the information on host and process the information of each time window. Registry Monitor, File System Monitor, and Network Stack Monitor compose the host-based monitor M1, which passes the extracted feature vector to suspicion level generator M2. Router M3 will pass the collected Netflow of each time window to Flow Analyzer M4. Then M4 will analyze the Netflow, extract the feature vector, and pass it to the Cluster Analyzer M5. M5 will cluster the hosts in LAN based on the network feature vector of each time window and the preprocessed information of host distance and then pass the results to correlation engine M6. Through sending requests to all hosts in each cluster, M6 will combine host information with network information to calculate the final detection result to determine whether the host has been infected.

*3.2. Time Complexity Analysis.* Table 2 shows the time complexity analysis of the three approaches. BotSwat calls only two query interfaces from Behavior-Check Procedure M4 to get the detection result. The time complexity of query and calculation is $O(1)$; therefore the overall time complexity is $O(1)$. The trigger mechanism of BotTee has three modules: M2, M3,

Table 2: Time complexity of each module (X denotes no computing, — denotes no such module).

| | M1 | M2 | M3 | M4 | M5 | M6 | Total |
|---|---|---|---|---|---|---|---|
| BotSwat | X | $O(1)$ | $O(1)$ | $O(1)$ | — | — | $O(1)$ |
| BotTee | X | X | $C(m,2)n\log(n)$ | $tmn\log(n)$ | — | — | $O(tmn\log(n))$ |
| Zeng | X | $O(1)$ | X | $O(1)$ | $O(n^2)$ | $17MN$ | $O(n^2)$ |

and M4. M2 is to obtain system call sequences without any calculation. For the correlation engine M3, it needs to analyze all the semantic units to generate sematic templates through the method of Longest Common Subsequence [25, 26] (LCS). Suppose there are $m$ semantic units and the length of each semantic unit is $n$. Because every two semantic units need to calculate their LCS once, the total number of calculations is $C(m,2)$. The time complexity of LCS's common algorithm is $O(n^2)$ [25], and it can reach $O(n\log(n))$ after optimizing. Thus, the time complexity of M3 is $C(m,2)n\log(n)$. Semantic behavior matcher matches the suspicious semantic unit with every template in the database by the match algorithm of LCS. Suppose there are $m$ semantic units and $t$ templates in database; the time complexity of M4 is $tmn\log(n)$; thus the overall time complexity is $(C(m,2)+tm)n\log(n)$. Due to the fact that $m$ is much smaller than $t$ and $C(m,2)$ is much smaller than $tm$, the overall time complexity is $O(tmn\log(n))$. The trigger mechanism of the approach of Zeng [5] has four modules: M2, M4, M5, and M6. For the obtained feature vectors of host, the host-based suspicion level generator M2 only needs to predict once by using trained LIBSVM [27] and then needs one calculation. So the time complexity is $O(1)$. When it comes to the process of network information, the time complexity of the flow analyzer M4 is $O(1)$. The algorithm of cluster analysis M5 is Hierarchical Clustering Algorithms [28], and the time complexity of this algorithm is at least $O(n^2)$, in which $n$ is the number of involved objects, and in this experiment it refers to the number of hosts in LAN. After clustering, information will be passed to the correlation engine M6. Suppose the correlation engine only calculates the results of $M$ hosts and there are $N$ hosts in each cluster, the complexity is $17MN$ and the overall time complexity is $n^2+17MN$. Because $M$ is smaller than $n$ and $N$ is also smaller than $n$, the overall time complexity is $O(n^2)$.

The time complexity of BotSwat [9] is $O(1)$, BotTee [4] is $O(tmn\log(n))$, and Zeng et al.'s [5] is $O(n^2)$. It can be seen that the approach of BotSwat [9] can detect suspicious processes more quickly. The time complexity of BotTee [4] is so high that the delay is very serious, while the approach of Zeng [5] has a certain degree of delay. Due to the fact that high time complexity can affect the overhead of hosts and detection accuracy, we should reduce the time complexity of the detection approach as far as possible under the condition of not decreasing the accuracy.

We conclude Table 2 based on the analysis. As shown in Table 2, the module with high time complexity is mainly the correlation engine and the module providing data for correlation engine. In addition, due to the interception of system calls and processing, interception mechanisms and intercepted system calls (M1 in Figure 2) will have major impact on approaches' overhead. Thus, the three major factors affecting approaches' overhead are the interception mechanisms, intercepted system calls, and correlation engine. Also there are some more factors which generate high detection overhead. For example, in BotSwat, the overhead of tracking tainted data in tainted engine and the overhead of tracking clean data in user-input component are also very time and resource consuming. For BotTee, the behavior graph construction has high overhead. For Zeng's work, the information synchronization between hosts and centralized server is also important for overhead. The capture of network information in the router will consume the resources of the router. These factors are important for the specific approach; however they are not that common for host-based bot detection approaches and a little difficult to analyze. Thus we only compare and analyze the three factors and we may analyze other factors in our future works. We implement three experiments to evaluate our analyses.

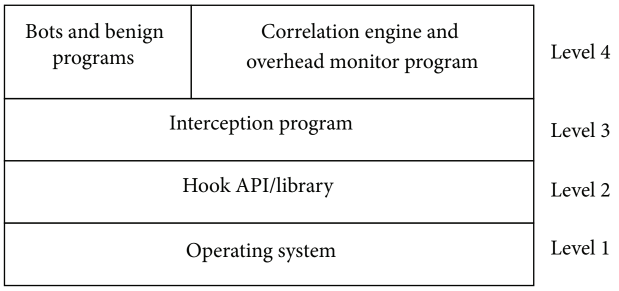


Figure 3: Implementation architecture.

## 4. Experimental Evaluation

*4.1. Implementation.* In our experiments, we make the following assumptions: (1) suppose the detection accuracy or the effectiveness is uninfluenced by these factors. That means these factors only influence the overheads. (2) Suppose CPU usage and memory usage reflect the overhead. We know these two values may not expose the slight overhead difference; however they can reflect the real impact on normal use. (3) Suppose we can get the information provided for correlation engine.

Those detection approaches are evaluated in different implementation architectures. In order to compare and analyze their overhead, we need to evaluate them in the same implementation architecture. As shown in Figure 3, this is the implementation architecture of our experiments. There are four levels. Level 1 is operating system, like Windows XP. Level 2 is hook API or library, like Windows Hook API, Detours, and Deviare API. Level 3 is our interception program based on hook API or library in level 2. Level 4 has two different parts. One is bots and benign software; these are the information sources to our interception programs in level 3. The other one is correlation engine and overhead monitor program; correlation engine deals with the data captured by interception program in level 3; overhead monitor program monitors the usage of CPU and memory.

In our experiments, we make three different experiments and each of them focuses on one factor. In the first two experiments we use Windows Hook, Detours, and Process Monitor to intercept system calls. The detection approaches we simulate are BotSwat [9], BotTee [4], and Zeng's [5] approach. BotSwat uses Detours, BotTee uses Deviare API, and Zeng's approach uses Process Monitor.

Experiment 1 is about interception mechanisms. There are three types of interception mechanisms: using Windows Hook, using third-party libraries, such as Detours and Deviare API [21], and using packaged tools, such as Process Monitor [22]. We choose one approach from each class to compare and analyze their differences. The program using Windows Hook uses the same technique as [29] and uses system-wide Windows Hook. The system calls are rewritten in a DLL file and the DLL file will be loaded when hook engine starts. The program using Detours injects DLLs by remote threads (function Create Remote Thread). It also loads a DLL file including the system calls needed to be hooked. The program using Process Monitor uses its commands to program. Process Monitor intercepts all system calls and stores the information in log files, we export the log files to csv format for later analysis. The main program has two threads: one controls Process Monitor and the other one analyzes the log file. The first two programs intercept the same number of system calls, and the last program intercepts all system calls.

Experiment 2 evaluates the type and number of system calls intercepted. Park proposes the concept of Common API in BotTee [4]. Common API refers to a set of APIs extracted through the analysis of a large number of existing bot commands' execution. Every API in this set is called by at least one bot command. They believe it is not necessary to intercept system calls out of this set, and in this way they can improve the efficiency and accuracy of the detection approach. BotSwat intercepts almost all system calls. Several other approaches intercept a small number of key APIs. We select three different numbers of system call sets. One is the Common API; we get the Common API using the method in [4]. One is a small number of key system calls; it only has several important system calls. The last one is all system calls. In order to reduce the impact of different interception mechanisms, we use Detours to intercept a small number of key system calls and Common API, and we use Process Monitor to intercept all system calls.

Experiment 3 evaluates correlation engine. We compare and analyze Zeng's approach [5] and EFFORT [18]. The reason we choose these two approaches is that they represent two typical approaches. Zeng's approach [5] correlates information obtained from hosts and that from network, which is an approach combining host and network. EFFORT [18] obtains multiple detection results about suspicious processes on host and correlates multiple detection results to get the final detection result, which is a host-based correlation method. The correlation method in EFFORT [18] mainly uses support vector machine (SVM). As the third assumption, in the experiment we assume that we can get the detection results of all parts from the host. Assuming there are some rules of judgment, according to these rules different numbers of detection results of the host can be produced randomly as the positive samples (judged as bots in detection results) and negative samples (judged as benign hosts in detection results) of training data. In Zeng's approach, we assume that the detection has been completed on the host, and the detection results can be obtained from the host. We capture the network flow at the CERNET network during a day as the background data. After the data has been filtered, we use pvclust [30] to process the data with hierarchical clustering and then combine the detection results provided by host with the network-related information to get the final detection result through correlation algorithm.

*4.2. Evaluation Methodology and Experiment Setup.* Experimental environment has a small impact on the overhead, accuracy, and other features of the detection approach. So the closer to real environment is, the more convincing the results will be. The experimental environments are mainly divided into two classes: the normal operating system and the environment using virtual machines or emulators.

Some approaches use normal operating systems, such as BotSwat [9], the approach of Al-Hammadi [14], BotTee [4], and Kolbitsch's [13] approach. BotTee [4] establishes an independent network, in which all hosts run the Windows operating system, and the first host works as the C&C server while the second host as an infected host and the third as an attack target. Meanwhile, they all use Deviare API [21] to intercept the system calls of Windows API. The approach of Kolbitsch uses a single-core, 1.8 GHz Pentium 4 running Windows XP with 1 GB of RAM.

Some approaches use virtual machines or emulators [5, 10, 11, 18]. BotTracer [10] uses Windows XP Professional, version 5.5.3 of VMware workstation as a virtual machine, VMware Converter [31] to clone hosts. In the experiment, they establish a controlled network. BotTracer [10] runs on a host with 2.79 GHz CPU and 2 GB RAM, and it intercepts Win32 function calls by using Microsoft Detours 2.1 Express [32] at the same time. The approach of Martignoni [11] connects one target Qemu virtual host to another virtual host VMgway. In VMvict, the system-level emulators monitor benign or malicious processes. The VMgway serving as gateway has three functions: to isolate emulators from the external network to prevent further infection; to provide a real network environment for malwares that can identify networks; and to control the behaviors of bots as C&C server. Zeng's [5] botnet runs in a controlled environment where VMware virtual machines are used to run Windows XP operating systems, to collect the execution traces through monitoring a virtual network. EFFORT [18] installs their systems on 11 different hosts in real life and collects a few days' data. When detecting, a virtual environment is established and runs three virtual machines, which are infected host,

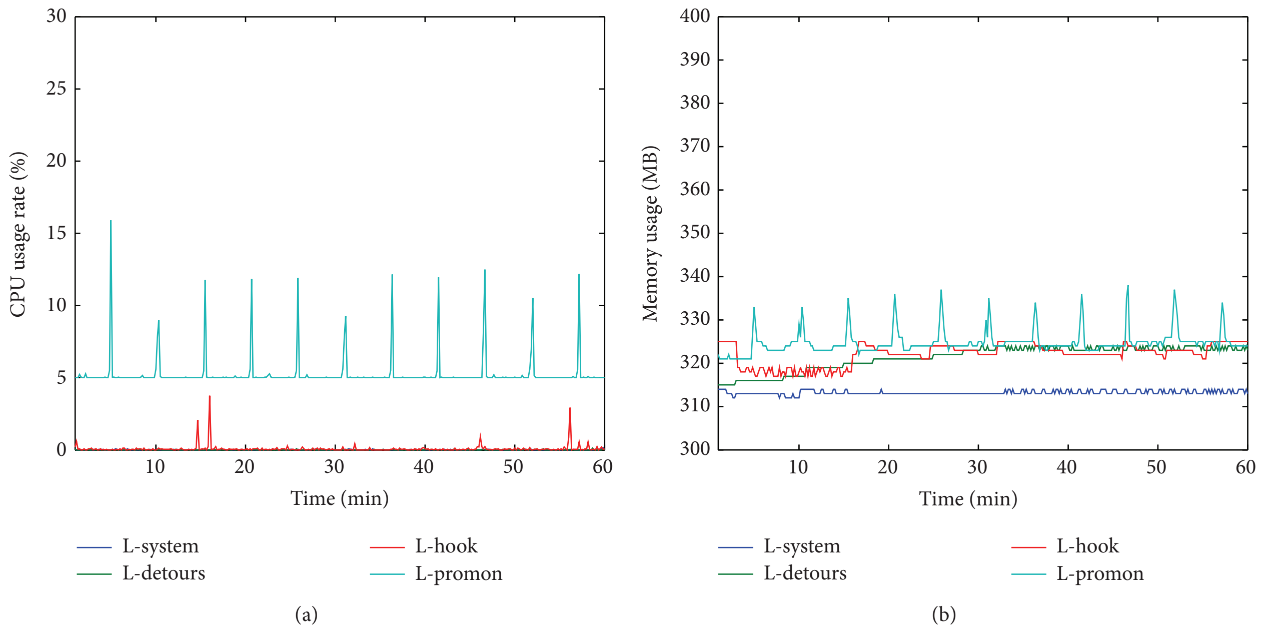


Figure 4: Overhead of running operating system in testing the type of interception mechanisms.

controller, and monitor. Windows XP SP3 operating system and basic software are installed.

We select the normal operating system and establish a controlled LAN. There are four independent hosts; one acts as the botmaster, while the other three act as infected hosts, Host1, Host2, and Host3. They all connect to the same router. The four hosts have the same configurations: the Intel Q6600 quad-core processor, 2.40 GHz, 2G RAM, and Windows XP SP3 operating system. For the choice of bots, due to the uncontrollable of P2P bots, we only use HTTP bots and IRC bots, and they are HTTP-based bot Zeus [33], IRC-based bot SdBot [34], and Agobot [35]. For the choice of benign programs, considering the benign programs used in the above detection approaches, we choose uTorrent, IE, Firefox, Eudora, eMule, mIRC, and so forth.

We perform three experiments with each one focusing on one affecting factor. In each experiment, we test the overhead of the operating system, one typical benign program running, all the benign programs running, and all the benign and bot programs running. When testing the overhead of benign programs, we start them all at the beginning of the test time window. During the time window, we normally use the benign programs, such as surfing on the Internet using IE and chatting with friends using mIRC. For the bots, we also start them all at the beginning. During their running, we control the botmaster to send some commands to them. For each case, we test them for one hour for several times and finally take the average result of all these times as the final result.

### *4.3. Experiment Results*

*4.3.1. Experiment 1: The Type of Interception Mechanisms.* There are four sets of experimental result figures. In each set, (a) shows the change of CPU usage rate and (b) shows the change of memory usage. L-system represents running only operating system, L-detours represents using Detours to intercept, L-hook represents using Windows Hook to intercept, L-promon represents using Process Monitor to intercept, L-one represents running one typical benign program, L-benign represents running all benign programs, and L-benign-mal represents running all benign and malicious programs. Figure 4 shows the change only with operating system running. Figure 5 shows the overhead of running a typical benign program, mIRC. Figure 6 shows the above-mentioned benign programs are added to the host and the host can have access to the Internet. Figure 7 shows that two classes of bots are added to the host. In order to avoid bots' propagation out of our control, we cut the link to the Internet.

In Figure 4(a), L-system and L-detours are almost always at 0%. Besides a few fluctuations, L-hook remains at 0%. The program using Process Monitor to intercept occupies at least 5% CPU usage at run time. A fluctuation appears every five minutes because the time window is set at 5 mins. After 5 mins Process Monitor will convert data format for later analysis and the conversion process occupies a lot of CPU portion. In Figure 4(b), the memory usage of pure operating system remains at 312 MB, and L-detours and L-hook remain at 320 MB. While due to every 5 mins' data format conversion, that is, a fluctuation appearing per 5 mins, L-promon remains at 320 MB in a normal state of monitoring.

Using Detours and Windows Hook to intercept system calls has a small impact on CPU and memory usage. Using Process Monitor has a greater impact on CPU and memory usage, while not that great compared with the entire host.

To evaluate the difference when all or only one program is running, we evaluate the overhead of a typical benign program, m-IRC. Figure 5 presents the detailed overhead of running m-IRC. In Figure 5(a), L-system, L-one, and L-detours are almost always at 0%. L-hook has a few fluctuations between 0% and 2%. L-promon costs at least 5% CPU usage

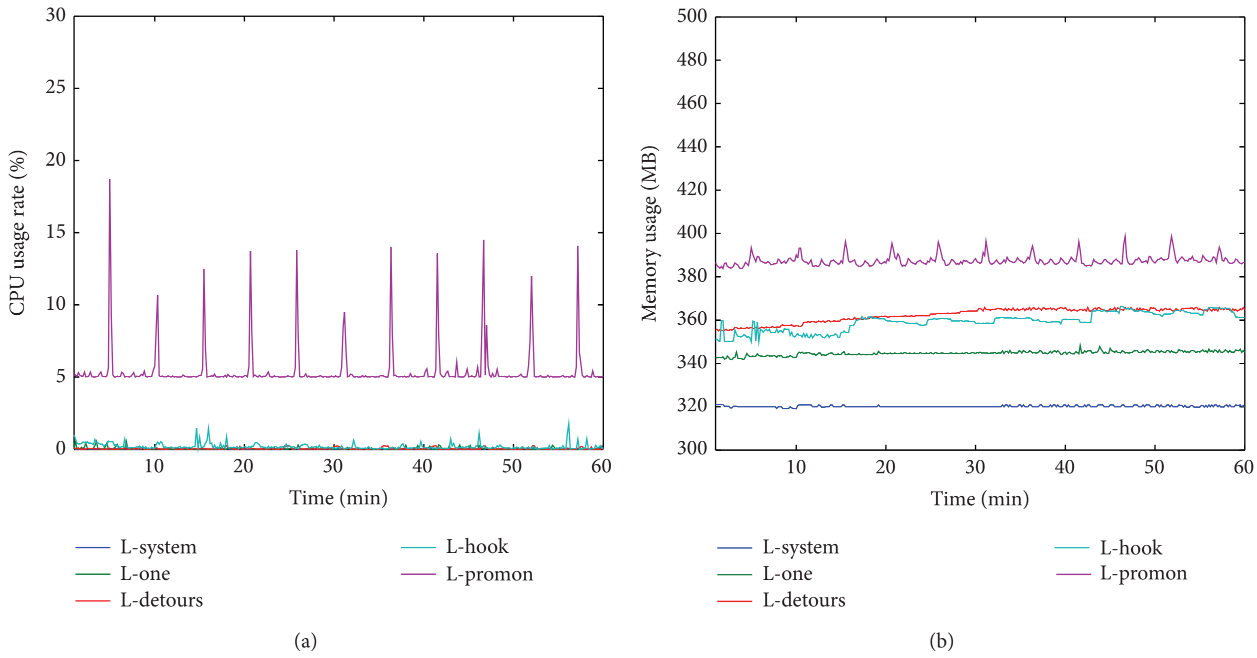


Figure 5: Overhead of running one benign program in testing the type of interception mechanisms.

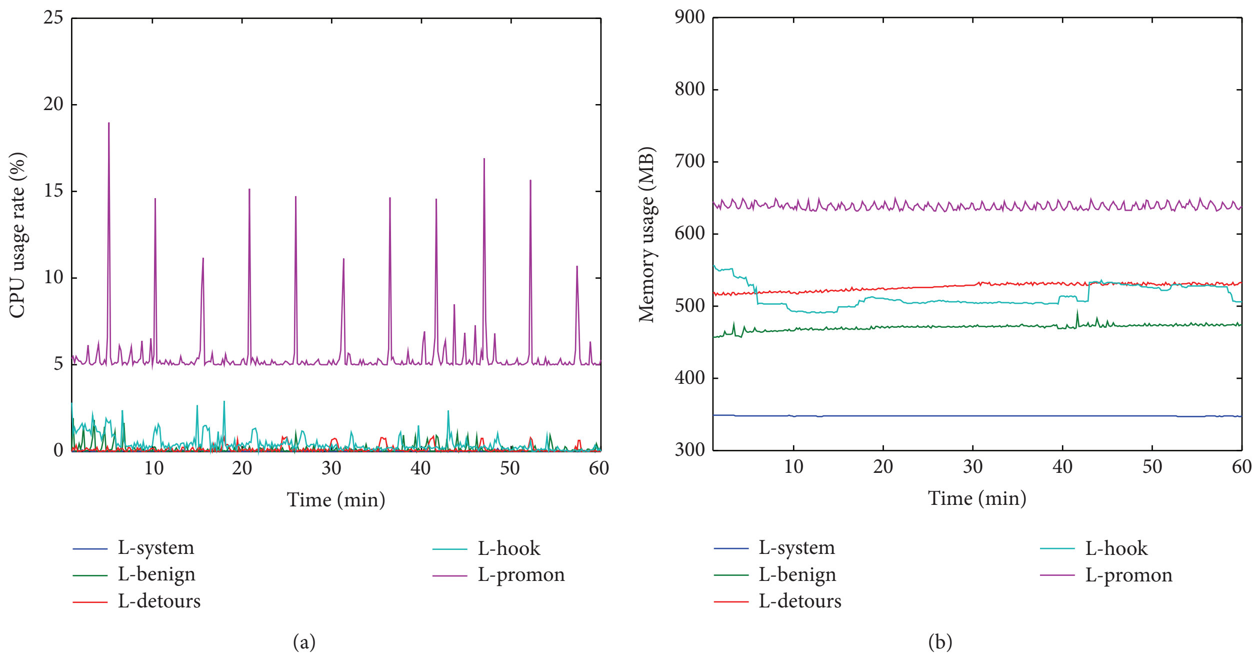
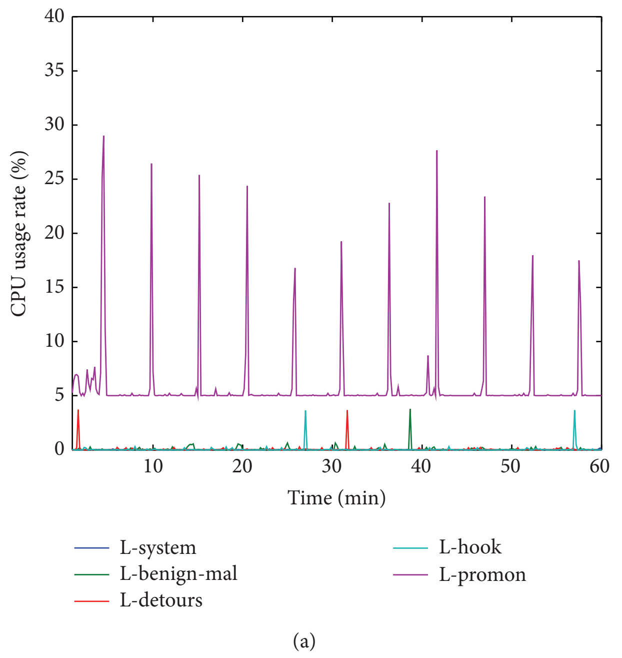


Figure 6: Overhead of running all the benign programs in testing the type of interception mechanisms.

and has a sudden rise during the format conversion about every 5 mins. In Figure 5(b), L-system remains at 320 MB, L-one almost remains at 340 MB, L-detours remains almost at 360 MB, and L-hook rises and falls between 350 MB and 360 MB. L-promon fluctuates between 380 MB and 400 MB. Compared with Figures 4 and 6, we can get two conclusions. First, a benign program occupies a low CPU utilization in the initial stage. However, it is much less than what all benign programs occupy. Second, a benign program only has a little impact on CPU utilization. However, for memory usage, it has an obvious impact on all the three interception mechanisms.

In Figure 6(a), L-system remains at 0%; L-benign and L-detours almost stay at 0%–2%. L-hook is higher than L-benign in the first 20 mins; however they are almost the same in the following 40 minutes. L-promon occupies at



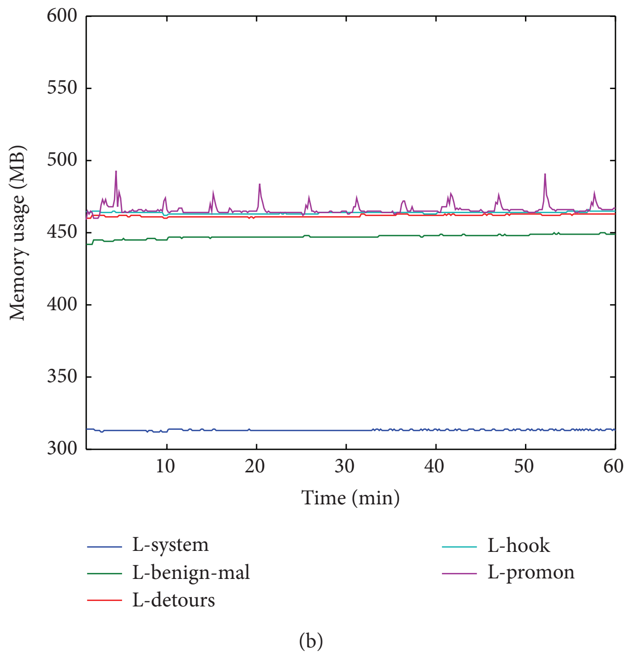


Figure 7: Overhead of running both benign and malicious programs in testing the type of interception mechanisms.

least 5% CPU. In addition to the CPU time that format conversion occupies, CPU usage experiences some fluctuations in the first 10 mins due to benign programs' execution. In Figure 6(b) L-system remains at 350 MB, and L-benign almost remains at 470 MB; while L-detours remains almost at 520 MB, L-hook rises and falls between 500 MB and 550 MB. L-promon remains almost at 640 MB. The result of Figure 6 is almost consistent with the results of Figure 4.

Figure 7 shows that both benign programs and two classes of bots are running in the host. In order to avoid bots' propagation out of our control, we cut the link to the Internet. Therefore, in this experiment benign programs cannot run as well as in Figure 6 and the usage rate of CPU and memory will reduce. We install bot server in Bot Master to send bot commands; Host1, Host2, and Host3 are infected by bots. In experiments we send different commands to bot hosts at set intervals.

In Figure 7(a), L-system remains at 0%, L-benign-mal, L-detours, and L-hook remain almost at 0% with several fluctuations. The reason is that the commands we send launch some sensitive operations and they are intercepted by interception tools. During the conversion stage, the CPU usage rate of L-promon is higher than that in Figure 6. This shows that malicious programs create a lot of system calls, especially the key system calls. In Figure 7(b), L-system remains at almost 320 MB, L-benign remains at about 450 MB, and L-detours and L-hook remain at almost 460 MB. L-promon remains almost at 460 MB except for being in the data format conversion stage.

Using Windows Hook and Detours has a small impact on the overhead of detection approaches while having a high efficiency. Process Monitor has a great impact on the overhead of detection approaches, especially in the data format conversion stage. The disadvantage of using Windows Hook is that we need to add the APIs manually, and its overhead highly depends on the choices of APIs. Therefore, it is easy to get false negatives or false positives because of human factors. We need to program every detail of Windows Hook, thus its programming complexity is rather high. However, Detours is a library which is already programmed well and all we need to do is add necessary codes according to the requirements. It is easy to program and implement and not easy to make errors.

From this experiment, it is more suitable to use third party libraries (e.g., Detours) to intercept system calls.

*4.3.2. Experiment 2: The Type and Number of System Calls Intercepted.* There are also four sets of experimental result figures below. Figure 8 shows the result of running only operating system, Figure 9 shows the overhead of running a typical benign program, Figure 10 shows the result of running all benign programs, and Figure 11 shows the result of running both benign and malicious programs. In each figure set, (a) shows the impact on CPU usage rate and (b) shows the impact on memory usage. The benign programs are those mentioned above and have access to the Internet. The malicious programs are the three ones mentioned above and have no access to the Internet.

In Figure 8(a), L-system, L-detours-less, and L-detours remain almost at 0%. Programs' running occupies 5% in L-promon, so L-promon fluctuates in data format conversion. In Figure 8(b), L-system occupies about 312 MB. L-detours-less and L-detours occupy about 323 MB in a stable state. L-promon occupies about 323 MB during the monitoring stage and fluctuates in data format conversion stage.

In Figure 8 using Detours to intercept a small number of APIs has no obvious difference with using Common API. There is only a little difference in memory usage in the first 30 mins in Figure 8(b).

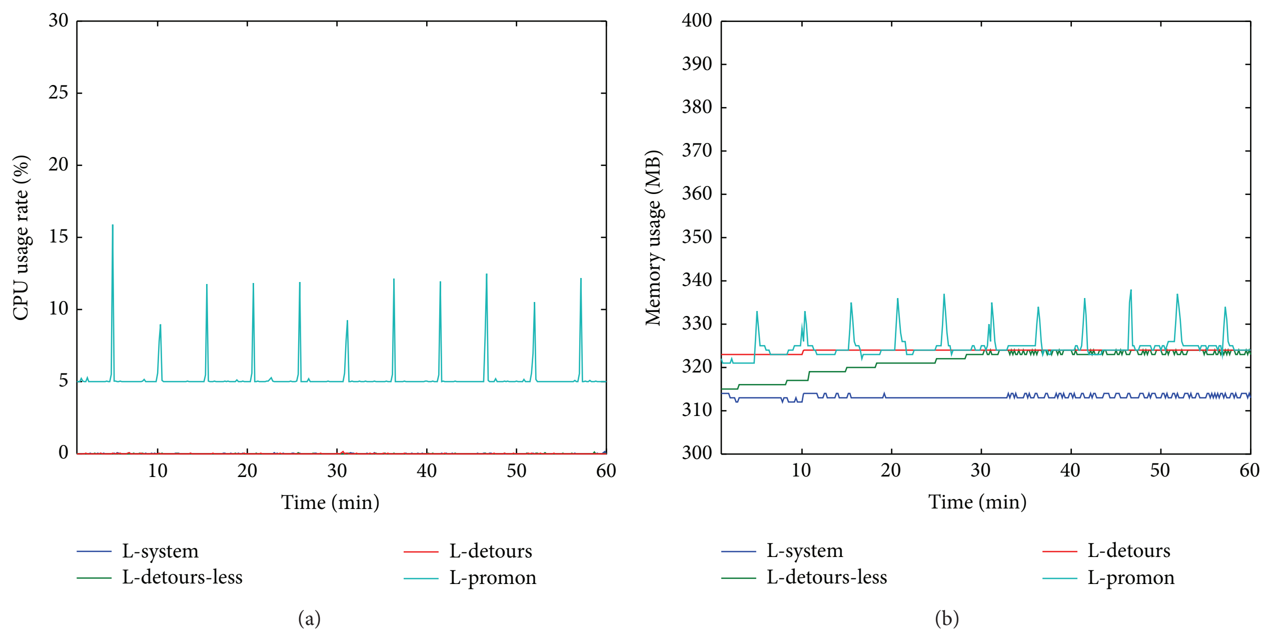


Figure 8: Overhead of running only operating system in testing the type and number of system calls intercepted.

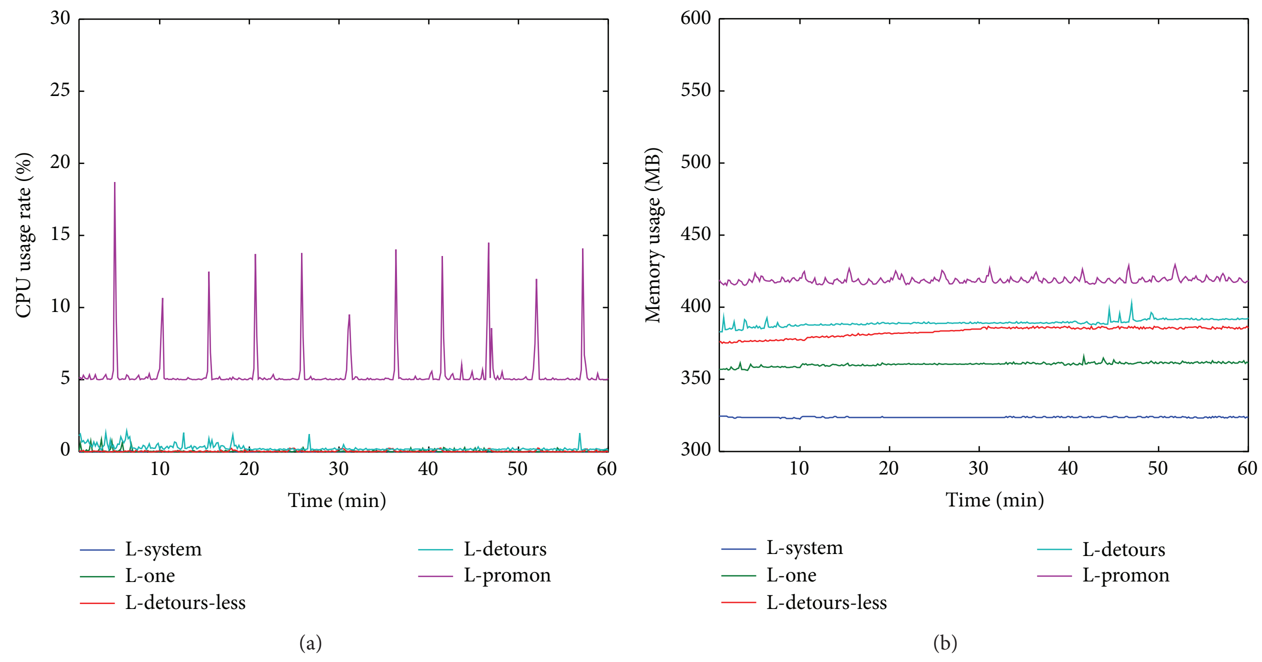


Figure 9: Overhead of running one benign program in testing the type and number of system calls intercepted.

We run m-IRC to test the impact of one program on the type and number of system calls intercepted. In Figure 9(a), L-system, L-one, and L-detours-less remain almost at 0%, while L-one occupies a less CPU usage during the initial stage. L-detours has a little fluctuations between 0% and 2%, and L-promon still has a sudden rise during data format conversion. In Figure 9(b), L-system remains at about 320 MB, L-one almost remains at 360 MB, L-detours-less remains almost at 380 MB, and L-detours remains almost at 390 MB, while L-promon fluctuates between 310 MB and 320 MB. Compared with Figures 8 and 10, we can see that (1) in the initial stage a benign program occupies a little CPU usage, which is much less than all benign programs. (2) For CPU usage, a benign program has a little impact on Detours and Process Monitor. However, for memory usage, it has an obvious impact on all.

In Figure 10(a), L-system, L-benign, and L-detours-less remain almost at less than 1%, while L-detours rises and falls in the first 20 mins between 1% and 5%, and, the rest of the time, it remains almost at 1%. In L-promon there are a few fluctuations during monitoring stage, while it remains almost

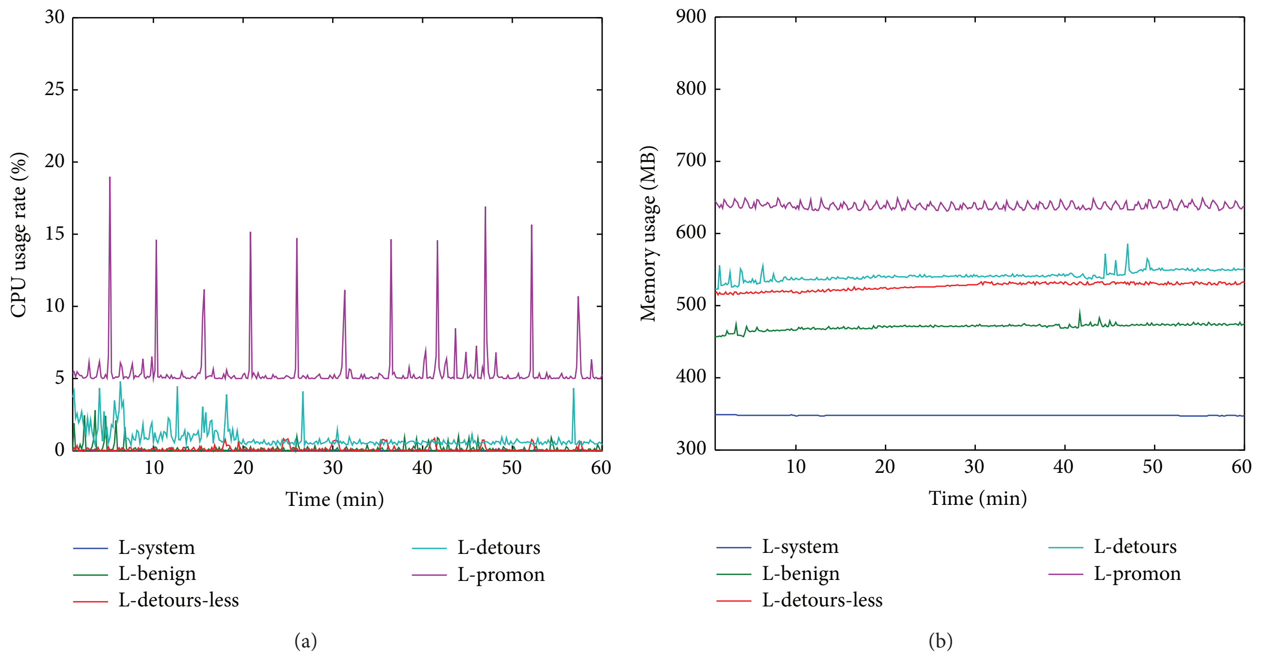


FIGURE 10: Overhead of running all the benign programs in testing the type and number of system calls intercepted.

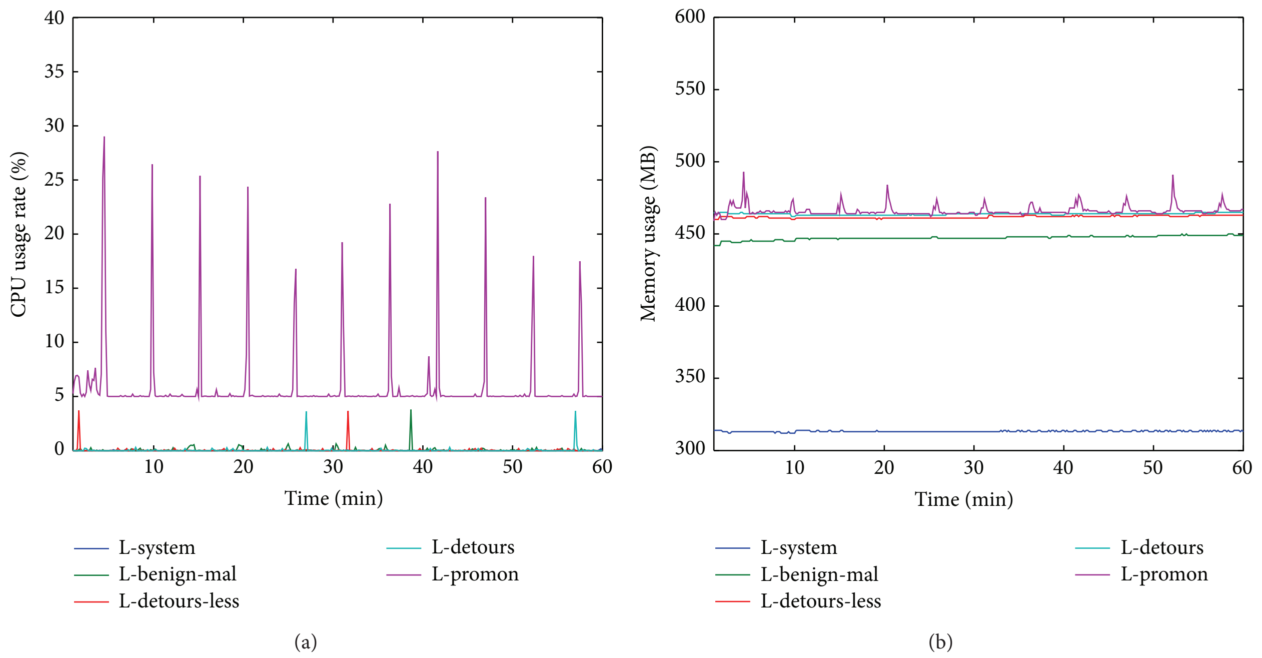


FIGURE 11: Overhead of running both benign and malicious programs in testing the type and number of system calls intercepted.

at 5%. The data format conversion stage occupies a lot of CPU. In Figure 10(b), L-system remains almost at 350 MB, L-benign remains almost at 470 MB, L-detours-less remains at about 525 MB, L-detours remains at about 550 MB, and L-promon remains at about 640 MB.

We can see that using Detours to intercept Common API occupies more CPU and memory than to intercept a small number of APIs, while the difference is very small.

In Figure 11, due to the disconnection with network, benign programs' running is restricted. Therefore, in Figure 11(a), L-system, L-benign-mal, L-detours-less, and L-detours remain at 0% to 1% with a few fluctuations. While L-promon occupies more CPU in data format conversion process than that in Figure 10(a). We believe this shows that the execution of malicious programs brings a large number of system calls. In Figure 11(b), L-system remains

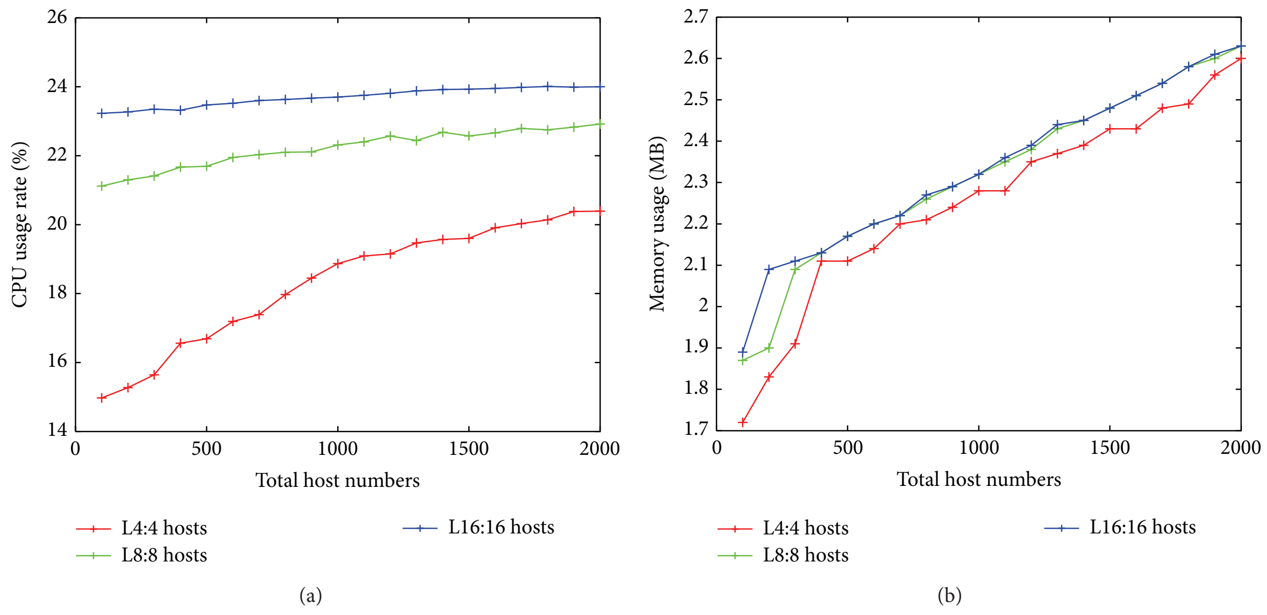


Figure 12: Overhead of Zeng's correlation method under different parameters.

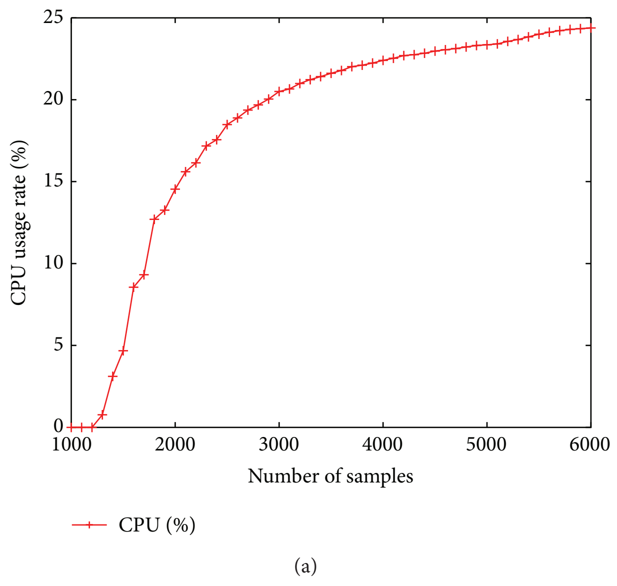
at about 320 MB, L-benign-mal remains at about 450 MB, L-detours-less and L-detours remain at about 460 MB, and L-promon remains at about 460 MB in monitoring stage though there are fluctuations in data format conversion stage.

Using Detours to intercept Common API occupies more CPU and memory than to intercept a small number of APIs, while the difference is very small. Therefore, the overhead of intercepting Common API is acceptable. Intercepting all APIs needs to process a lot after interception, such as removing unrelated APIs; thus the efficiency will fall and the overhead will increase. Intercepting a small number of APIs has a higher efficiency; however it will significantly increase false negative rate. Therefore, intercepting Common API increases the efficiency of detection approach significantly without reducing the detection accuracy. It is more reasonable to use Common API to intercept system calls.

*4.3.3. Experiment 3: Correlation Engine.* In Zeng's approach [5], the input information of correlation engine has two parts: the host information and the network information. The complexity of the algorithm mainly lies in the calculation of the average distance of a host in its cluster. In the experiment, we control the number of hosts before clustering and the number of hosts doing correlation calculation at the same time. Then we test the overhead of correlation algorithm under different circumstances. The number of hosts before clustering is divided into 20 cases ranging from 100 to 2000 and the number of concurrent hosts doing correlation computing is divided into three levels of 4, 8, and 16. We conduct detecting experiments multiple times for each level of each test case and record the CPU and memory usage. Figure 12(a) is the change of CPU; as we can see from the figure, when a small number of hosts do correlation computing, the total number of hosts in their network has a great impact on CPU usage, and when there are a large number of concurrent hosts, the total number of hosts has a small impact on CPU usage. Figure 12(b) is the change of memory usage. During the calculation of the average distance of a host in its cluster, the correlation engine needs to load hosts' information that have been clustered. When the concurrent number increases to a certain extent, it almost needs to load all hosts' information, while memory usage changes a little, so the major factors affecting memory usage are still the number of hosts in the network.

The correlation engine of EFFORT [18] detects the suspicious process on the host through three modules which are process reputation analysis, system resources analysis, and network information analysis to finally generate the detection report. SVM algorithm is used in calculating each modules weight when correlating the results of each module. The complexity of the correlation algorithm mainly lies in the number of samples and the number of support vectors. We use LIBSVM [27] to test the overhead of correlation engine. In the experiment, sample data is divided into 51 test cases from 1000 to 6000, and each test case, respectively, run for multiple times, recording the average usage of CPU and memory when the correlation algorithm is running. Figure 13(a) shows the CPU changes in the training stage of the correlation algorithm; when the sample data increases, the CPU usage increases obviously, while the growth rate decreases when it reaches a certain extent. However, with the decrease of growth rate, the training time significantly increases. Figure 13(b) shows the result of the experiment about memory changes. Since, in the training process, sample data needs to do matrix operation, which applies for a lot of space, the memory change increases almost in a linear way.

Zeng's correlation engine combines host detection results and the related information of other hosts in the network, while EFFORT's correlation engine only uses the detection



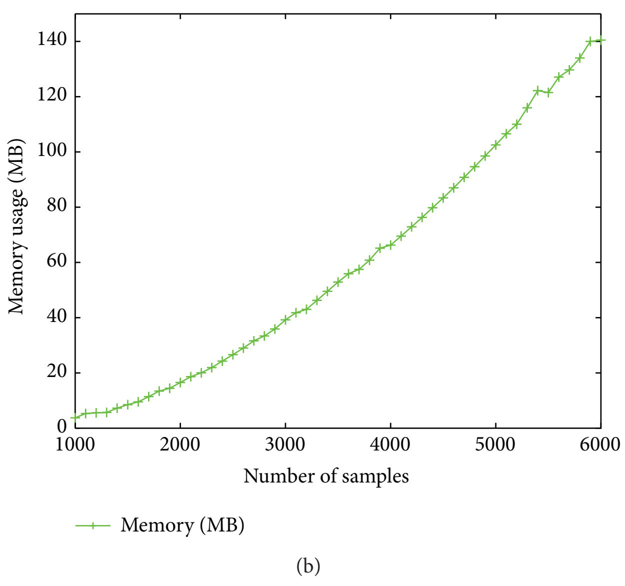


FIGURE 13: Overhead of EFFORT correlation method under different parameters.

TABLE 3: Overhead of the type of interception mechanisms.

| Interception mechanism | System | | Benign | | Malicious | |
|---|---|---|---|---|---|---|
| | CPU | Memory | CPU | Memory | CPU | Memory |
| Windows Hook | 0.0823% | 322.1556 | 0.4596% | 514.2444 | 0.0425% | 463.8000 |
| Detours | 0.0071% | 321.3278 | 0.1131% | 526.4917 | 0.0444% | 461.7306 |
| Process Monitor | 5.2647% | 324.8750 | 5.5251% | 638.3167 | 5.8490% | 465.9056 |

results of multiple modules on hosts. Zeng's correlation engine is not deployed in bot host and has a small impact on the overhead of hosts. The host deploying the correlation engine handles all detection results and it has a great impact on the host. The correlation method of EFFORT has a great impact on host overhead at the training stage. And, with the increment of sample data, the memory increases very quickly. However, in the prediction stage it only needs to handle a single host's detection result and has a small impact on the overhead of the detected host. Therefore, it can be seen that the correlation engines of the two approaches both have small impact on the overhead of the detected hosts. However, they have a great impact on the host deployed as the correlation engine.

*4.4. Summary.* According to the above analysis and experiments, the major factors affecting host overheads are interception mechanisms on host, the type and number of system calls intercepted, and correlation engine. We summarize the detailed overhead values in Tables 3 and 4. We can draw the following knowledge:

(1) There are three primary interception mechanisms: using Windows Hook to intercept, using Detours, Deviare API, and other third-party libraries to intercept, and using packaged tools, such as Process Monitor [22], to intercept. After the experiment in Section 4.3.1, we calculate the average values of CPU usage and memory usage in three different situations. As shown in Table 3, using packaged tools, such as Process Monitor, has a great impact on host overhead, and using Windows Hook and Detours has a smaller impact on host overhead.

(2) The type and number of intercepted system calls can be divided into three classes: intercepting all system calls, intercepting a specific subset of API calls, such as Common API, and intercepting a small number of key system calls. After the experiment in Section 4.3.2, we calculate the average values of CPU usage and memory usage in three different situations. As shown in Table 4, intercepting all system calls has a great impact on host overhead, and intercepting Common API and a small number of key APIs has a smaller impact on host overhead.

(3) Correlation engines can be divided into two classes based on detection approaches: correlation of internal information on host, such as EFFORT, and correlation of information on host and information on network, such as the approach of Zeng. The experiment result in Section 4.3.3 indicates that correlation of internal information on host has a great impact on host overhead because the operations of calculation match, especially the more increased suspicious information, the more overheads increase. Correlation engine that uses both host and network information is not

Table 4: Overhead of the type and number of system calls intercepted.

| Intercepted system calls | System | | Benign | | Malicious | |
|---|---|---|---|---|---|---|
| | CPU | Memory | CPU | Memory | CPU | Memory |
| Few APIs | 0.0071% | 321.3278 | 0.1131% | 526.4917 | 0.0444% | 461.7306 |
| Common API | 0.0058% | 323.8444 | 0.9025% | 541.5444 | 0.0431% | 462.6302 |
| All system calls | 5.2647% | 324.8750 | 5.5251% | 638.3167 | 5.8490% | 465.9056 |

deployed in the suspicious host; thus this approach has no impact on host. It has great impact on the overhead of host which acts as correlation engine. Due to the delay of information exchange, large-scale computation, and other reasons, the real-time detection is affected.

According to the experimental results and summary, we propose the following optimizations:

(1) For experiment 1, the Process Monitor adopts similar mechanisms as Windows Hook but with more overhead of creating an independent GUI process to achieve real-time monitoring. Detours and Windows Hook use different hooking mechanisms; however their difference on CPU and memory usage is little. We may take a more fine-grained comparison between Detours and Windows Hook in our future works. In actual programming, programmers should pay attention to every interception details of Windows Hook. The programming complexity is much higher than Detours. Therefore, we believe using third-party library, such as Detours, is more reasonable.

(2) In the aspect of the type and number of intercepted system calls, intercepting all system calls has a great impact on host overhead, especially when many benign programs are running. Intercepting a small number of key system calls has a very low overhead, yet which system call we should select is difficult to decide. If we select inappropriately, the false negative rate will be significantly increased. When intercepting a specific subset of system calls, such as Common API, the subset is created through the analysis of existing bots or other malicious codes. The coverage thus is very wide, and it will not increase the false negative rate. Therefore, intercepting a specific subset of system calls, such as Common API, can significantly reduce the impact on host overhead without increasing the false negative. We believe that using a specific subset of system calls, such as Common API, is a more reasonable choice.

(3) Since the design of correlation engine is flexible and there is no sole criterion, we propose the following measures to improve the correlation engine: minimize the time and space complexity of the correlation methods and optimize the calculation modules; enumeration approach should be abandoned when using database to match; unrelated match should be reduced by increasing the index, buffer pool, and so forth. In calculating module, we should try to avoid large-scale computing in real time and reduce online detection overhead by increasing offline computing, such as preprocessing.

## 5. Discussion

There are several limitations which are also our future works. (1) We suppose the detection accuracy or the effectiveness are uninfluenced by these factors. Only measuring the overheads without considering detection accuracy is not completed. One might be willing to pay more in overhead penalty with high detection accuracy. We will take a more accurate analysis with both overhead and detection accuracy in our future works. (2) We suppose CPU usage and memory usage reflects the overhead. These two values may not expose the slight overhead difference. Also we measure the CPU and memory usage in a time interval of 10 seconds. We will take a more fine-grained analysis with more accurate criteria and metrics. (3) Beside these three factors, there are some more factors which generate high detection overhead, such as, tainted data tracking, clean data tracking, behavior graph construction, and information synchronization between hosts and centralized server. We may analyze these factors in our future works.

## 6. Conclusions

Since traffic-based approaches cannot completely eliminate the threats of botnet, detection approach based on a single host is considered more effective. However, the overhead of host-based bot detection approaches remains a bottleneck blocking its wider deployment. This paper compares and analyzes the features of host-based bot detection approaches and the complexity of each module. We identify three major factors affecting the performance of approaches, including interception mechanisms on host, the type and number of system calls intercepted, and correlation engine. Then we evaluate these factors through experiments. Finally, we summarize the experiment results and discuss some optimizations which can significantly improve the performance of host-based bot detection approaches.

## Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback to this paper. This work was supported in part by the National Natural Science Foundation of China under Grants nos. 61170265 and 61472162.

## References

[1] Z. Bu, P. Bueno, R. Kashyap, and A. Wosotowsky, "The new era of botnets," White Paper from McAfee, 2010.

[2] G. Gu, *Correlation-Based Botnet Detection in Enterprise Networks*, ProQuest, Ann Arbor, Mich, USA, 2008.

[3] Y. Park, Q. Zhang, D. Reeves, and V. Mulukutla, "AntiBot: clustering common semantic patterns for bot detection," in *Proceedings of the 34th Annual IEEE International Computer Software and Applications Conference (COMPSAC '10)*, pp. 262–272, IEEE, Seoul, Republic of Korea, July 2020.

[4] Y. Park and D. S. Reeves, "Identification of bot commands by run-time execution monitoring," in *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC '09)*, pp. 321–330, IEEE, December 2009.

[5] Y. Zeng, X. Hu, and K. G. Shin, "Detection of botnets using combined host- and network-level information," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*, pp. 291–300, IEEE, July 2010.

[6] T. Kwon and Z. Su, "Modeling high-level behavior patterns for precise similarity analysis of software," in *Proceedings of the 11th IEEE International Conference on Data Mining (ICDM '11)*, pp. 1134–1139, IEEE, December 2011.

[7] X. Wang and X. Jiang, "Artificial malware immunization based on dynamically assigned sense of self," in *Information Security*, vol. 6531 of *Lecture Notes in Computer Science*, pp. 166–180, Springer, Berlin, Germany, 2011.

[8] F. Y. Law, K. Chow, P. K. Lai, and K. Hayson, "A host-based approach to botnet investigation?" in *Digital Forensics and Cyber Crime*, vol. 31 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 161–170, Springer, Berlin, Germany, 2010.

[9] E. Stinson and J. C. Mitchell, "Characterizing bots' remote control behavior," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 4579 of *Lecture Notes in Computer Science*, pp. 89–108, Springer, Berlin, Germany, 2007.

[10] L. Liu, S. Chen, G. Yan, and Z. Zhang, "Bottracer: execution-based bot-like malware detection," in *Information Security*, pp. 97–113, Springer, 2008.

[11] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, "A layered architecture for detecting malicious behaviors," in *Recent Advances in Intrusion Detection*, Lecture Notes in Computer Science, pp. 78–97, Springer, Berlin, Germany, 2008.

[12] N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

[13] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proceedings of the 18th Conference on USENIX Security Symposium*, pp. 351–366, USENIX Association, 2009.

[14] Y. A. A. Al-Hammadi, *Behavioural correlation for malicious bot detection [Ph.D. thesis]*, University of Nottingham, 2010.

[15] J. Greensmith, J. Feyereisl, and U. Aickelin, "The DCA: SOMe comparison," *Evolutionary Intelligence*, vol. 1, no. 2, pp. 85–112, 2008.

[16] L. N. De Castro and J. Timmis, *Artificial Immune Systems: A New Computational Intelligence Approach*, Springer, 2002.

[17] G. Jacob, R. Hund, C. Kruegel, and T. Holz, "Jackstraws: picking command and control connections from bot traffic," in *Proceedings of the USENIX Security Symposium*, 2011.

[18] S. Shin, Z. Xu, and G. Gu, "EFFORT: efficient and effective bot malware detection," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM '12)*, pp. 2846–2850, Orlando, Fla, USA, March 2012.

[19] Z. Xu, L. Chen, G. Gu, and C. Kruegel, "PeerPress: utilizing enemies' P2P strength against them," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '12)*, pp. 581–592, October 2012.

[20] G. Hunt and D. Brubacher, "Detours: binary interception of Win32 functions," in *Proceedings of the 3rd Conference on USENIX Windows NT Symposium*, 1999.

[21] Deviare API hook overview, Feburary 2014, http://www.nektra.com/products/deviare-api-hook-windows/.

[22] Process monitor, http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx.

[23] Anubis: Analyzing unknown binaries, 2014, http://anubis.iseclab.org/.

[24] Apitrace, 2014, http://apitrace.github.com/.

[25] Longest common subsequence problem—wikipedia, the free encyclopedia, 2014, http://en.wikipedia.org/wiki/Longest_common_subsequence_problem#cite_note-BHR00-2.

[26] T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Mass, USA, 2nd edition, 2001.

[27] "Libsvm—a library for support vector machines," 2014, http://www.csie.ntu.edu.tw/~cjlin/libsvm/.

[28] A tutorial on clustering algorithms, http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/hierarchical.html.

[29] API hooking revealed—codeproject, Feburary 2014, http://www.codeproject.com/Articles/2082/API-hooking-revealed.

[30] "Pvclust: an r package for hierarchical clustering with $p$-values," 2014, http://www.is.titech.ac.jp/~shimo/prog/pvclust/.

[31] Vmware vcenter converter, convert physical machines to virtual machines, 2014, http://www.vmware.com/products/converter.

[32] Detours—microsoft research, 2014, http://research.microsoft.com/en-us/projects/detours/.

[33] Zeus (trojan horse)—wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Zeus_(Trojan_horse).

[34] Thread description: Backdoor:w32/sdbot.mb, 2014, http://www.f-secure.com/v-descs/sdbot_mb.shtml.

[35] Agobot—wikipedia, the free encyclopedia, 2014, http://en.wikipedia.org/wiki/Agobot.