Computer Networks 57 (2013) 2628–2642

Contents lists available at SciVerse ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

EFFORT: A new host–network cooperated framework for efficient and effective bot malware detection

Seungwon Shin *, Zhaoyan Xu, Guofei Gu

Texas A&M University, College Station, Texas, United States

ARTICLE INFO

Article history:
Received 6 April 2012
Received in revised form 8 December 2012
Accepted 31 May 2013
Available online 13 June 2013

Keywords:
Botnet
Botnet detection
Network security

ABSTRACT

Bots are still a serious threat to Internet security. Although a lot of approaches have been proposed to detect bots at host or network level, they still have shortcomings. Host-level approaches can detect bots with high accuracy. However they usually pose too much overhead on the host. While network-level approaches can detect bots with less overhead, they have problems in detecting bots with encrypted, evasive communication C&C channels. In this paper, we propose *EFFORT*, a new host–network cooperated detection framework attempting to overcome shortcomings of both approaches while still keeping **both** advantages, i.e., effectiveness and efficiency. Based on intrinsic characteristics of bots, we propose a multi-module approach to correlate information from different host- and network-level aspects and design a multi-layered architecture to efficiently coordinate modules to perform heavy monitoring only when necessary. We have implemented our proposed system and evaluated on real-world benign and malicious programs running on several diverse real-life office and home machines for several days. The final results show that our system can detect all 17 real-world bots (e.g., Waledac, Storm) with low false positives (0.68%) and with minimal overhead. We believe EFFORT raises a higher bar and this host–network cooperated design represents a *timely effort* and a *right direction* in the malware battle.

1. Introduction

Botnets (networks of bot malware controlled machines) are considered as one of the most serious threats to current Internet security [10,32]. To eradicate threats posed by bots/botnets, a lot of research has been proposed so far, and they fall into two main categories: (i) network-level detection and (ii) host-level detection. Network-level detection approaches focus on network behavior of bots/botnets and they typically concentrate on finding signatures or common communication patterns between bots and their masters [11,10,12,32]. Host-level detection approaches investigate bot runtime behavior in the host and they mainly employ system call monitoring and/or data taint analysis techniques [18,20,28].

Both detection approaches have their own advantages and disadvantages. Network-level approaches can detect different types of bots without imposing overhead to the hosts, because they mainly monitor network traffic. However, their limitations appear when they need to detect a bot communicating through encrypted messages or randomized traffic [29]. Host-level approaches, on the contrary, analyze suspicious runtime program behavior, so that they can detect a bot even if it uses an encrypted or evasive communication channel. However, they typically suffer from performance overhead because they need to monitor all invoked system calls [18] at real-time and/or taint memory locations touched by the program [28].

Observing their clear advantages and disadvantages motivates us to consider a new system with merits from

* Corresponding author.
E-mail addresses: seungwon.shin@neo.tamu.edu (S. Shin), z0x0427@cs.tamu.edu (Z. Xu), guofei@cs.tamu.edu (G. Gu).

1389-1286/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.comnet.2013.05.010

both approaches: (i) effectiveness and (ii) efficiency. For the effectiveness, the system should detect malware with few misses. In addition, the system should not put too much burden on both host and network to achieve the efficiency.

As a promising step toward such a system, we propose *EFFORT*, a new detection framework balanced with high accuracy and low overhead. *EFFORT* considers both host- and network-level features that are helpful to enhance strong points of each other and complement weak points of each other, and it coordinates these features to achieve the main goal (i.e., detecting bots *effectively and efficiently*).

To build EFFORT, we start with investigating several notable intrinsic characteristics of recent popular botnets. First, bots are usually automatic programs without requiring human-driven activities. Second, bots highly rely on DNS (instead of hard-coded IP address in binaries) for flexible and agile C&C (command and control). In addition, they use more advanced DNS tricks for robustness and evasion, such as fast-flux service networks or even domain fluxing [14]. Third, bots access system resources anomalously (e.g., registry creation and file read/write) to steal system information or launch themselves automatically. Finally, bots are likely to distribute their information for their malicious activities to the network (e.g., sending massive spam) instead of gaining information from the network which is common in normal networked user applications. Thus, they tend to minimize incoming C&C command communication to reduce the exposure/detection probabilities.

Based on their characteristics, we find several useful features at host and network level. For efficiency, we perform lightweight human–process–network correlation analysis. We correlate interactions between human and process, and record correlated clues between processes and outgoing DNS connections. Thus, we can filter majority benign programs, and focus on very few suspicious automatic programs contacting DNS servers and start further investigation.

To detect bots effectively, we further employ three interesting new modules. First, we monitor *system resource exposure patterns* of the suspicious process. Second, we build a *reputation engine* to characterize the reputation of a process through examining the process and its social contacting surfaces. Our intuition is that the reputation of a process could be approximately inferred by the reputation of its social contact surface, i.e., reputations of communicated remote hosts. This is intuitively sound because bots are likely to communicate with "bad" targets while good software tends to communicate with "good" ones. Although a pre-established host reputation database (or blacklist) is helpful, we do not require it as prior knowledge. Instead, we can use anomaly-based features from DNS registration information. Furthermore, we want to leverage community-based knowledge and intelligence by using public search engines to locate information about certain communicated targets and then infer their reputation. Third, we analyze network information trading rate for any network process to infer how likely the program is information gaining oriented or information leaking/outgoing oriented.

It is well accepted that probably no any single module/feature can detect all bots. We do not claim that bots cannot evade any of our proposed individual module. Instead, we rely on a correlative approach to combine information from multiple complementary modules for a final decision, because the chance that a bot evades all our modules is very slim unless they sleep or behave like normal programs.

In short, our paper makes the following contributions.

- We propose a new *host–network cooperated* framework for bot malware detection with correlative and coordinated analysis. This design demonstrates an important step from current state of the art toward both *effective* and *efficient* botnet detection.
- We implement *EFFORT*, a first-of-its-kind real-world prototype system containing several novel modules to cover bot invariants at both host and network levels.
- We extensively evaluate our system on real-world data collected on many real-life machines for several days. Our results show that EFFORT can detect all 17 real-world bots and it has only 8 false positives (out of 1165 benign processes) in about a week of testing (without any whitelisting). We demonstrate that EFFORT has almost negligible overhead for host modules and its network modules such as process reputation analysis are scalable.

2. System design

The overall architecture of EFFORT is shown in Fig. 1, which contains five modules:

- *Human–process–network correlation analysis module*. It analyzes the interaction and correlation between human activity, process, and network connection. It tells whether a network connection is human-driven or bot-driven.
- *Process reputation analysis module*. It characterizes the reputation of a process from the process itself (who you are) and its social contact surface (the communicated targets, i.e., whom you have talked to).
- *System resource exposure analysis module*. It examines the system resource exposure patterns to a suspicious process in detail.
- *Network information trading analysis module*. It monitors incoming/outgoing network traffic in order to infer the information gain/loss in a very light-weight way.
- *Correlation engine*. It collects all analysis results and correlates them to make a final decision whether the process is likely a malicious bot or not.

The overall operation of EFFORT is summarized as follows. First, *Human–process–network correlation analysis module* finds a process producing bot-driven network connections and notifies other three modules – *process reputation, system resource exposure, and network information trading analysis modules* – about the found process. Second, these three modules investigate the suspicious process in detail and each module issues a detection result (but not a final decision) by its own analysis engine. Finally,

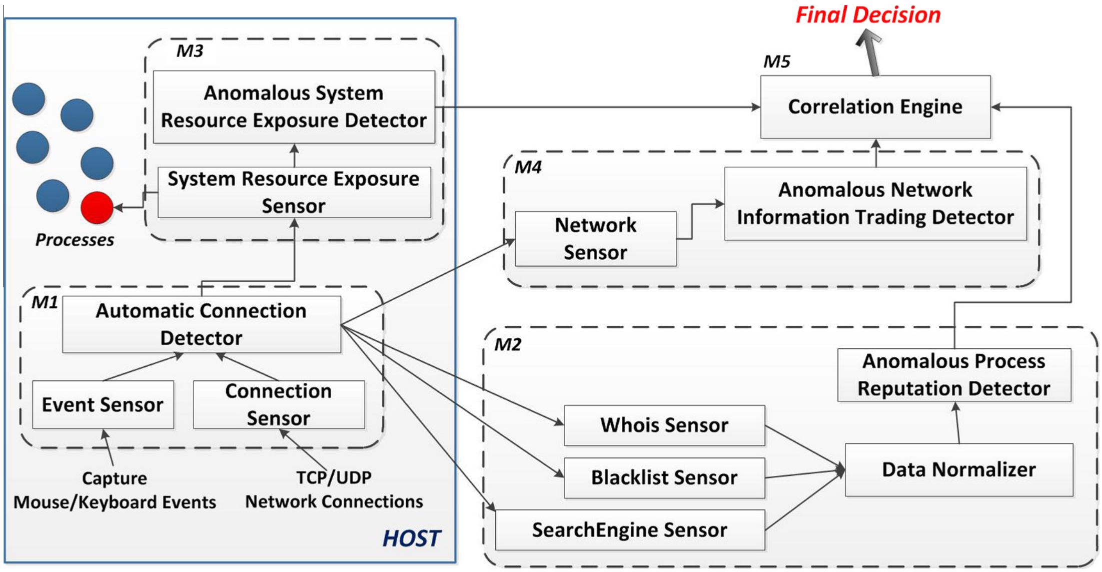
Final Decision
M5
Correlation Engine
M3
Anomalous System Resource Exposure Detector
System Resource Exposure Sensor
Processes
M4
Network Sensor
Anomalous Network Information Trading Detector
M1
Automatic Connection Detector
Event Sensor
Connection Sensor
Capture Mouse/Keyboard Events
TCP/UDP Network Connections
HOST
M2
Anomalous Process Reputation Detector
Whois Sensor
Blacklist Sensor
SearchEngine Sensor
Data Normalizer

Fig. 1. EFFORT design architecture. M1 is *human–process–network correlation analysis module*, M2 is *process reputation analysis module*, M3 is *system resource exposure analysis module*, M4 is *network information trading analysis module*, and M5 is *correlation engine*.

Correlation engine collects the detection results from the above modules and finally decides whether the process is malicious or not (final decision).

2.1. Human–process–network correlation analysis

Since most running processes are benign, it is relatively inefficient to monitor **all** of them in *fine-grained* detail (e.g., monitor system call level activities) **all** the time. Thus, our *human–process–network correlation analysis* module is designed to sift benign programs out.

Human–process interactions monitoring: Keyboard and mouse are the basic components that link human and the computer. We monitor keyboard and mouse events of the host to understand which program has human activity/interaction. To do this, our *event sensor* hooks Windows system calls which are related to keyboard and mouse events, and then determines which program generates those events.

Some previous studies (e.g., [6]) also apply function hooking to capture automated processes. We differentiate from them in several aspects. Previous approaches will suffer if a bot simply imitating human behaviors and creating fake mouse/keyboard events from virtual devices to confuse the sensor. To address this problem, we employ two more robust approaches. First, our sensor digs into the sources of the events. If the events are resulted from physical devices connected via PS2 or USB interface, it trusts them; otherwise it regards them as suspicious. Second, the module investigates whether a process generating events is running in foreground or not. We assume that if someone produces mouse or keyboard events, a process related to the events is shown on the current screen with an activated windows (i.e., running in foreground). Thus, if a process producing events is running in foreground, we trust the process; otherwise we regard the process as suspicious.

Note that in current implementation, we trust operating system and we believe it provides true information, a **common** assumption widely used in this line of research [18,6]. Of course, some malware (e.g., rootkit) may infect operating system, deliver fake information, and even kill our system. This issue could be solved by employing hardware/TPM [13] or Hypervisor-based introspection and protection [9,16] (and thus is out of the scope of this paper).

Process–network interactions monitoring: Further differentiating from some previous studies which mainly focus on identifying automated processes, we choose to trace the process–network interaction and identify automated processes that generate automated network connection (particularly DNS queries). We use a *connection sensor* to record outgoing network connections from processes in the host. In particular, it cares about one special network connection, DNS query. As briefly discussed before, botnets heavily rely on using DNS for flexible, efficient, and evasive C&C rallying. They can even use fast-flux service networks [14] to frequently change the IP addresses associated with one domain name, or even use domain fluxing [27] to frequently change domain names. By monitoring these DNS queries, we can obtain valuable information later in detection analysis, e.g., we can determine if they are human driven or not. Furthermore we can even detect whether there is fast-flux in use.

Interaction model generation and automatic connection detection: Combining information from the *event sensor* and the *connection sensor*, we create a model to describe which process correlates to which network connection. The model employs three metrics: (i) time difference between the time when a process issues a DNS query and the prior time when the process produces a mouse/keyboard event, (ii) the source of the events, and (iii) whether a process is running foreground or not at the time. We regard an event of a process is generated from human, if the time difference is very small, the event is from actual physical devices, and the process is running foreground.

Practical considerations: However, in practice, this intuitive model may not work for all DNS queries (but work well for IP address case). The reason is because some operating systems provide helper functions of relaying a DNS query for other processes, e.g., Windows uses the *svchost.exe* process for this purpose for some sockets. Thus, DNS queries are sent from helper processes instead of the original program. To address this issue, we maintain a list of returned IP address(es) from a DNS query (sent by helper processes), and observe successive outgoing network connections to wait for finding the actual program (process) to connect to the returned IP address(es).[1] If we find the process, we can create a model by using metrics (mentioned above) for the process.

2.2. Detecting malicious processes

With the help of the previous module, we can focus on some suspicious processes. However, we need more deep analysis to investigate whether they are really malicious or not. To do this, we perform a set of independent and parallel checks. We check the reputation of the process and its social contact surface (the reputation of targets it has communicated with). We investigate the system resource exposure patterns to the process. Furthermore, we investigate network information trading of the process. We detail our design of these three modules as follows.

2.2.1. Process reputation analysis module

First we use a novel approach to determine the reputation of the suspicious process. A quick intuitive observation is that we could determine the reputation of a process by not just looking at "who you are", but also referring to "whom you have talked to". Bots are likely to contact some "bad/suspicious" servers/peers *automatically* in order to be controlled. On the contrary, benign programs are relatively unlikely to connect to "bad" targets *automatically*. Thus the problem of determining the reputation of a process could be roughly inferred by the contacting social surfaces of the process and it can be approximately reduced to the accumulation of "bad" communication targets (domains). In the context of domain names, then we need to determine the reputation of a domain name.

Domain information collection: We collect reputation information of the domain by employing three types of sensors. First, we employ a *whois sensor* to detect some anomaly features in its *registration information*, such as domain creation date. Second, we use a *blacklist sensor* to investigate its *previous records* in well-known blacklists (e.g., SpamHaus [26]), which give us relatively clear clues about whether the domain has a history of malicious activity or not. Finally, since blacklists might not be complete, we apply a *search engine sensor* to get another heuristic that can leverage *community-based* knowledge and intelligence, i.e., asking a search engine to infer the reputation of given domain names (IP address could work too). It is motivated by the googling idea in existing work [30].

Table 1
Numerical value of the selected features of domain reputation analysis module.

Feature	*Numerical values*
Domain creation date	Current date–creation date
Domain expiration date	Expiration date–creation date
Number of registration	Number of DNS service providers registering domain name
Blacklist	if NOT in any blacklist it is 1, otherwise 0
Web search engine results (title, URL, summary, malicious keywords)	The title, URL, and summary are 1 if we find the domain/process name in each item, otherwise 0. The malicious keyword is 1 if we cannot find any malicious keywords from the returned pages, otherwise 0

Feature extraction and normalization: Before applying collected data to a model creation, we express the features numerically and normalize them. In terms of domain registration information, we use the following features (the intuitions of the expressions are described in brackets): (i) difference between current date and domain expiration date (most malicious domains registered very recently), (ii) difference between domain expiration date and creation date (most malicious domains have short life time), and (iv) number of domain registration (malicious domains are typically registered to few name servers). All features are summarized in Table 1.

An interesting and novel component in the process reputation module is our search engine sensor. The general intuition is that some of the knowledge about the domain or process is probably already discovered/reported/summarized by other people in the world. Thus, we can leverage the wisdom of the whole Internet community by using search engines like Google. More specifically, we consider the following simple yet effective features: (i) whether the domain name is well-indexed (thus returning many results), (ii) in the top returned web page results, whether the domain name and the process name are frequently used in a malicious context, e.g., they are surrounded by malicious keywords such as bot, botnet, malware, DDoS, attack, spam, identity theft, privacy leak, command and control (C&C).

Again we use numeric values to represent these heuristics as shown in Table 1. Typically, contents of returned search results include three different types of information: (i) the title, (ii) the URL, and (iii) the relevant snippet of the returned web page. We treat each type as different features, and we assign "1" if returned contents (title, URL, and summary) include the queried domain name. We inspect the returned results to see whether there are any malicious keywords or not. If there are any, we give "0" for its value.

In the case of the blacklist, it is very obvious that if a domain can be found in blacklists, it is suspicious. We give

[1] At this time, we do not need to monitor all network connections, we only monitor first packet of each connection.

“0” if it is in blacklists, otherwise “1”. The features related to returned results by a search engine and blacklist are already normalized, i.e., their values are between “0” and “1”. However, features of domain registration can be varied dynamically. To make their values range between “0” and “1”, we employ a *Gaussian normalization* approach. It regards a distribution of data as Gaussian function and maps every data point into the probability of Gaussian function.

Process reputation model creation: We employ a Support Vector Machine (SVM) classifier [5] for the *process reputation model*. The SVM classifier maps training examples into feature spaces and finds (a) hyperplane(s) which can best separate training examples into each class (the detailed information of the SVM classifier will be explained in Appendix A.1).

In this model, we consider that the normalized features, which are mentioned above, are training examples. In addition, we define that there are two classes – benign and malicious – in this model, thus the normalized features will represent one of the two classes. Finally, we will find (a) hyperplane(s) which can best separate training examples into each class. Then, we can obtain a SVM classifier for the *process reputation model*.

Anomalous process reputation detection: It is very frequent that a process contacts several different domains during a certain period. Thus, we examine all contacted domains using our trained SVM model, and determine whether “bad” domains (i.e. classified as malicious domains) exist or not. If there exists at least one, we consider the process reputation as bad (malicious), otherwise it is good (benign).

2.2.2. System resource exposure analysis

If a bot infects a host, it usually tries to do something useful for its master (to make profit), e.g., stealing information, sending spam, and launching DDoS attacks [15]. These operations consume system resources – memory, cpu, and network – of the host, read/modify files or registries [20]. If we monitor how system resources are exposed to a process (and to what degree), we could infer its anomalous access patterns.

System resource exposure patterns monitoring: A *system resource exposure sensor* monitors resource access activities of a suspicious process. It monitors how critical resources such as files, registries, and network sockets are exposed to the target process. Although this sensor shares some similar motivation with related work [20], we use different and more light-weight features and detection models as described below.

System resource exposure model creation: To build this model, we use the following heuristics: (i) typically normal processes rarely access files in other user’s folders and system directories, (ii) typically normal processes do not modify critical registries (with a few exceptions), and (iii) typically normal processes do not create a large number of sockets in a short time period. These heuristics are not perfect, i.e., some normal processes might have some of these patterns. Our goal of this module is not to have zero false positive, instead, we want to detect most of these system-resource-consuming malware. Thus, we believe these heuristics are reasonable.

More specifically, these heuristics can be represented as the following events Y_i of a process:

- Y_1: access files in other user’s folders.
- Y_2: access files in system folders.
- Y_3: modify critical registries.
- Y_4: create a new process.
- Y_5: create too many network sockets within a short time window.

To build a system resource exposure model, we employ a One-Class SVM (OCSVM) classifier [24]. We use one-class instead of two-class SVM is because we will only use benign programs in training. To get the ground truth information of the system resource usages of malware is tricky, e.g., some malware may refuse running or behave normally. Thus, even if we obtain the information of malware, it may not represent its behavior clearly. To address this issue, we only use the system resource access patterns of known **benign** processes (i.e., one side of data). More detailed information on our use of OCSVM is explained in Appendix A.2.

2.2.3. Network information trading analysis

Typically, most user programs will act as clients rather than servers, and clients will try to gather information rather than distributing information. That is, if we treat a program as a communication information processing unit, normal client programs are more likely to be an information gaining process. However, a bot will behave differently. Usually, the data that a bot receives is a command from a botmaster, therefore the amount of the data may be small (to minimize the chance of being detected). However the data sent out by the bot could be relatively large as it performs malicious operations in the network. Information theft, DDoS attack, and massive spam sending are good examples.

Lightweight network traffic monitoring: To observe network information trades, a *network sensor* captures network flows between a process and a target address and stores them. An important thing here is that this sensor monitors network traffic generated by the specific process not by the host. It could give us more fine-grained observations of network information trading. Our sensor is very simple and lightweight, because it does not need to analyze payload contents and it is robust against encryption used by bots.

In addition, we monitor the host level network connections to obtain an aggregated view of network information trading. At this time, the sensor only measures the number of outgoing connection trials (i.e. TCP SYN packets and first UDP packets). We believe that this aggregated view gives us a good clue to find DDoS, network scan, or massive spam mail sending.

Network information model creation: We use a simple method to model the network information trade rate, i.e., the ratio of incoming and outgoing packets/bytes exchanged between a process and a remote site in a certain period. We define the number of incoming and outgoing packets as θ_1 and θ_2, and the number of incoming and outgoing bytes as δ_1 and δ_2. Thus, each ratio can be represented as $\frac{\theta_1}{\theta_2}$ and $\frac{\delta_2}{\delta_2}$.

To observe an aggregated view, we employ a time window w_i for each host i. We measure how many network connection trials happen in the time window.

Anomalous information trading detection: In the case of the fine-grained view, if one or both of the ratios of a process is (are) smaller than some threshold γ_1 (for packet) and γ_2 (for bytes), we consider the process anomalous, otherwise normal. Also, we consider the network behavior of the host is anomalous, if a host creates network connection trials larger than a threshold τ in w_i.

2.2.4. Correlation engine

After each module makes its own decision, the *correlation engine* will combine these results and make a final decision using a weighted voting system. It is worth noting that as any intrusion detection system, most of our individual modules might be evaded by very carefully designed bots. Indeed in the evaluation we will show that most modules will have some false negatives and/or false positives. However, when combining all modules together, we can achieve much better results, as demonstrated in Section 4. We also note that even individual evasion is possible, it will unavoidably compromise the utility and efficiency of bots. We extensively discuss possible evasions and solutions in Section 6.

At the correlation stage, we should determine the weights of the decision of each module. We can also employ SVM technique to determine which element (i.e. decision result of the module) is more important (i.e. should have more weight) [24]. To apply the SVM technique, we need training examples of both sides – malicious and benign. However, here we have the same issue as the *system resource exposure model* creation mentioned in Section 2.2.2. It would be relatively difficult to collect all the information of the malicious side. Thus, we decide to employ OCSVM to determine the weight [24]. The way how to determine the weights is same as the method explained in Appendix A.2.

2.3. Evasion ways and associated costs for each module

It is possible for botmasters to evade each module. If they want to evade the *human–process–network correlation analysis module*, their bots should not produce automatic network connections. Thus, bots have to compromise a program which is very frequently used by people and contact their masters when people are using the program. Moreover, the contacts have to be done when people create real mouse or keyboard events. It might be very hard to implement bots which can meet all conditions described before.

Likewise, we consider possible evasion ways of our modules and the costs from the evasions. We summarize them in Table 2. As explained in Table 2, to evade our module, botmasters consider a lot of different factors and it makes them hard to implement new bots. Although they build a new bot, which are able to evade all our modules, the bot might be useless because it is very hard for the bot to perform malicious activities in the host. We leave a more detailed discussion of evasion in Section 6.

3. System implementation

3.1. Host-level modules implementation

Our *human–process–network correlation analysis module* captures the mouse and keyboard events using Windows system functions. Basically, Windows provides functions to capture the events from external devices [23]. Using these APIs, we implement the *event sensor* which identifies which process generates the events. In addition, it investigates whether the events are generated from real physical devices and the process is running foreground with the help of Windows system functions. We also add the function to store captured information (process, event time) to the Shared Memory area.

To capture the outgoing DNS queries, TCP SYN, and UDP packets, we use the WinPcap library [31]. It provides functions to collect raw level network packets on the Windows OS, with little overhead. Moreover, *connection sensor* does not monitor all network packets, but monitor only DNS, TCP SYN and UDP packets. It also reduces the overhead, since those packets comprises a small portion of all network packets.

Whenever there are network events we should capture, our module also identifies which process produces them and verifies whether the process is related to the human actions or not. However, if a process uses a helper process for a DNS query, we could not directly use it. To address this problem, we check the process that produces the DNS query automatically and if it is a helper process (e.g., *svchost.exe*), the module waits a DNS reply which contains the IP address of the domain. Then, if there is an automatic connection from the process to that IP address after the DNS query, the module knows that the process issues

Table 2
Evasion ways and associated costs for each module.

Module	*Evasion ways*	*Evasion costs*
Human–process–network correlation	Compromise process which people frequently use, and initiates connections when user actually run	Very hard to run bots in the host
Process reputation	Use benign domains for C&C channels	hard to control bots, thus less efficient
System resource exposure	Do not access system and other users' folders, do not create processes, do not create a lot of network sockets	Very hard to perform malicious activities
Network information trading	do not distribute information to remote site or send more information to victim host	Hard to perform DDoS and send spam email massively, more chances of botmaster being exposed to public

the DNS query. We use `GetExtendedTcpTable` and `GetExtendedUdpTable` functions to recognize which process creates the connections. If we observe the TCP or UDP connection, we will call these functions to identify which process acquires the source port number of the connection.

We implement the *system resource exposure analysis module* based on EasyHook [8]. EasyHook is a successor and enhanced version of Detours [7], and it provides an interface letting us perform Windows API hooking. The hooking ability allows us to observe how a process works and which system calls are invoked by the process. We select 28 system calls to understand the exposure patterns of the process. The selected system calls are related to the access of the system resources, such as files, registries, network sockets, and creation of a process. In addition, we employ TinySVM library [19] to create the *system resource exposure model*.

3.2. Network-level modules implementation

To gather network features, the *process reputation analysis module* should utilize multiple network services such as whois services, blacklist identification services, and web searching services. Whenever the module receives a suspicious process and its contacting domains, it sends a query to multiple network services to gather network features and cache them for a while. Also, we use TinySVM library [19] to create the *process reputation model*. For a *network information trading analysis module*, we capture network packets usingthe Pcap library.

3.3. Correlation engine implementation

We implement the *correlation engine* as an independent process (using TinySVM library [19]) and it will wait for a message from each detection module, and finally decide whether the process is malicious or not.

4. Evaluation

In this section, we first show our real-world data collection of both benign and malicious programs. We discuss our model training. After that, we state our real-world bot detection results. We then discuss our false positive test. Finally, we report the efficiency, performance overhead, and scalability of each module in the EFFORT system.

4.1. Data collection and usage

4.1.1. Benign data collection

We install our modules into 11 different real-life hosts to collect the information of process activities and network behaviors for several days. These 11 machines are used by diverse users (including some lab members, friends in different majors, housewives) in the office or home. These machines are used in real-life operation for diverse usages, e.g., some users use network heavily for finding (or sharing) some information from (or through) networks, and some users mainly use for scientific simulation or regular document editing. The collection has been done in working hours on business days. We carefully examine to make sure that there are no malicious programs (especially bots) in the hosts, thus we consider that the collected data can be used as benign examples.

We explicitly collect data in two periods for different purposes: training and testing. We have collected training data from 6 machines and the collected data is denoted as **SET-1**. Later we have collected testing data on 8 machines (among them, 3 machines are also used for training data collection, but 5 machines are newly added). The data for testing is denoted as **SET-2**. Note that we intentionally test several new machines that we have no training data collected. This is to demonstrate that our training models are pretty generic and not sensitive or limited to specific programs or machines. Some detailed information of **SET-1** and **SET-2** is summarized in Table 3.

In terms of normal Windows programs installed on these machines that have generated network communications, they are very diverse, covering application programs such as browsers, multimedia applications, Windows Office programs and P2P applications. In training dataset **SET-1**, we have 61 distinct programs. And we have 71 programs in **SET-2** to evaluate our system (false positive test). Program examples are listed in Table 4.

4.1.2. Bot malware data collection

To test the false negative or detection rate on real-world bots, we build a virtual environment to run several collected real-world bots. The environment consists of three virtual machines which are individually served as an

Table 3
Benign dataset summary. (Programs represent the number of programs producing network connections).

		SET-1 (collected Nov. 2010)			SET-2 (collected Apr. 2011)		
Host ID	*Usage*	*Programs*	*Connection trials*	*Collection time*	*Programs*	*Connection trials*	*Collection time*
A	Office	7	252	<1 day	–	–	–
B	Home	8	10,927	4 days	–	–	–
C	Office	19	5740	5 days	–	–	–
D	Office	16	7859	3 days	37	113,067	12 days
E	Office	9	5098	4 days	12	20,711	6 days
F	Home	27	55,586	7 days	22	48,264	6 days
G	Office	–	–	–	16	12,169	4 days
H	Office	–	–	–	11	5,134	4 days
I	Office	–	–	–	8	17,373	4 days
J	Office	–	–	–	13	4,776	5 days
K	Home	–	–	–	10	14,455	5 days

Table 4
Example benign programs in **SET-1** and **SET-2**.

Date set	*Program examples*
SET-1	Google update, MS Word, Tortoise SVN, Vmware, Gom player, Bittorrent, Java, WinSCP, WeDisk (p2p program), Emule, Internet Explorer, Chrome, FireFox, iTunes, Thunderbird, EndNote, Skype, Putty, AdobeUpdater, MS Sidebar, Visual Studio 2010, GoogleTalk, QQ (Chinese chatting program)
SET-2	Google update, Gom player, Tortoise SVN, Internet Explorer, Chrome, MS Word 2010, Outlook 2010, Gom Audio, McAfee update, FireFox, Skype, SohuNews, MS Powerpoint, Google Talk, Eclipse, AirVideo, QQ, Kmplayer, Bittorrent, Emule, Windows media player, Dropbox, Windows live toolbar, MS Clip Organizer, Windows Error Reporter, Alzip, MS windows defender, Windows Task Manager, Vmware, MS Office Protection, Adobe synchronizer, SeaPort (MS search enhancement broker program), Sophos Security update, Putty, WeDisk

infected host, a controller, and a monitor machine. All of them install `Windows XP SP3` operating system with basic software installed, such as Internet Explorer browser and Microsoft Messenger. At the infected host, we create independent snapshot for each individual malware instance to ensure no cross-infection between different malware. Our host-based modules are also installed to collect the information of process activities and network behaviors for these bots. At the monitor machine, we install a fake DNS server to redirect all the DNS queries. At the controller side, we install various malware controllers we could find to manipulate the infected machine. We intend to reconstruct realistic attack scenarios that a botmaster sends commands to his zombie army.

We have used a total of 17 different bots (including Peacomm/Storm, Waledac, PhatBot). Their names, C&C protocols, and sample functionalities are summarized in Table 5. Since we just have binary samples of most bots except three (B1, B2, and B5), we install and simply run them. Among them, 3 botnets (B1, B2, and B5) use IRC protocol, 2 botnets (B4 and B10) use HTTP protocol, 2 botnets (B3 and B4) use P2P protocols, and other 9 botnets use customized protocols. In addition, three botnets (B3, B4, and B7) use encrypted protocols to evade network-level detection. In terms of their actively spreading time in the wild (i.e. when they infect victims highly), it varies from 2003 (B7) to recent (B4, B16). Since these collected botnets can cover diverse cases (e.g. from old one to currently working one, different types of protocols including encryption, and various kinds of malware functionalities), we believe that they can fairly validate our system's detection rate (or false negative rate).

We note that the separate collection of bot malware data and benign program data in our evaluation does not affect the accuracy of false positive/negative testing because our monitoring/recording/detection granularity is per-process instead of per-host (and obviously a bot program is a separate, different program from a normal program). Thus, we can easily mix the bot malware data and benign testing data **SET-2** to simulate real-world scenarios of bots running on some normal machines.

Table 5
Bots for Evaluation.

ID	*Name*	*Protocol*	*Sample functionalities*
B1	PhatBot	IRC	Steal Key, spam mail send, network scan
B2	JarBot	IRC	Kill Process, Steal Key
B3	Storm/ peacomm	P2P[a]	Other
B4	Waledac	HTTP, P2P[a]	Other
B5	PhaBot.α5	IRC	Other
B6	Flux	Custom	Operate/modify file, kill process, capture desktop/screen,
B7	NuclearRat	Custom[a]	Download update
B8	BiFrost	Custom	Operate file, kill process, capture screen, steal key
B9	Cone	Custom	Operate file
B10	Http-pentest	HTTP	Operate file, kill process, capture screen
B11	Lizard	Custom	Capture screen, DDoS
B12	PanBot	Custom	Flooding
B13	Penumbra	Custom	Operate file, create shell
B14	SeedTrojan	Custom	Download update
B15	TBBot	Custom	Capture screen, create shell
B16	Sality	Custom	Others
B17	Polip	Custom	Others

Custom denotes a botnet using its own protocol.
Other denotes other network/system malicious operations not categorized in the table.
[a] Represents the protocol is encrypted.

4.2. Model training

Based on **SET-1** data, we have created detection models for each module.

Process reputation model: From the collected data in **SET-1**, we find that processes have contacted 7202 different domains. In order to create the *process reputation model*, we extracted features as described in Section 2.2.1. At this time, we consider that all collected domains are benign, hence we will use them to represent the benign class. We also need malicious domains to represent the malicious class. For that purpose, we have collected recent 150 malicious domains from [22] and also extracted the corresponding features. Using these collected features, we train a SVM model for the classifier.

System resource exposure model: We analyze system resource exposure patterns of benign processes to create the one-class *system resource exposure model*. Here, we will only use information of benign processes and employ a OCSVM classifier to build the model. To do this, we use 77 collected benign processes information in **SET-1**. Representative benign processes here are the instances of browsers (e.g., Chrome, Firefox, IE), mp3 playing programs (e.g., Winamp, Gom player), p2p client program (e.g., Emule), and other programs such as MS Word, Visual Studio, Putty, Google Talk (more program examples are listed in Table 4). We extract each feature defined in Section 2.2.2 from the processes and build the OCSVM classifier.

Network information trading model: We analyze the network flow patterns and verify that most benign user programs act as clients instead of servers. We measure the ratio between incoming packets (or bytes) and outgoing packets (bytes). We apply the ratio of the incoming and outgoing packets to discriminate a malicious process from a benign process. It is obvious that when a bot delivers its own information to a master, we could detect them easily by observing the ratio.

We investigate the number of network connection trials of a host. We find in our training dataset that the maximum network connection trials of a host within a certain time window (2 s) is 41. Based on this result, we simply set the threshold τ as 49 (with 20% of error margin to be conservative).

Correlation engine model: To calculate weights for the *correlation engine*, we select 27 benign processes, which produce network connections frequently from **SET-1**. Most of them are processes of browsers, multimedia applications, and p2p client programs. We collect their detection results which are performed by our detection modules. Then, we train an OCSVM classifier using the collected results and determine the weights.

4.3. Detection results of real-world bots

We begin our false negative evaluation with the test of the *human–process–network correlation analysis module*, followed by the results of other modules.

4.3.1. Detection results of automatic connections

First, we test whether a bot program really generates automatic connections to remote servers and whether we can use the *human–process–network correlation analysis module* to detect them. To test this, we installed each bot in a host and leave it without any intervention. After a while, we find that all installed bots issue automatic connections to some remote servers (to be controlled). All of the automatic connections are captured by our *human–process–network correlation analysis module* and the detected information is delivered to other upper-layer modules.

4.3.2. Detection results of the process reputation model

The *process reputation analysis module* receives domain names that a bot process contacts. Then, the module analyzes the reputation of contacted domains. Since a bot contacts multiple domains, we analyzed all contacted domains. If the module finds any malicious domain from the contacted domains, it considers the process malicious.

The detection results on all bot programs are shown in Table 6. As shown in the Table, the *process reputation analysis module* detects 12 bots but misses 3 bots (B2, B3, and B4).

Table 6
Detection results of automatic connections.

ID	*Contacted domains*	*Detected domains*	*ID*	*Contacted domains*	*Detected domains*
B1	1	1	B10	1	1
B2	1	–	B11	1	1
B3	2	–	B12	1	1
B4	1	–	B13	2	2
B5	6	2	B14	1	1
B6	3	2	B15	1	1
B7	2	1	B16	4	2
B8	3	2	B17	5	1
B9	2	2	–		

We investigate why our module missed these three. In the case of B2 (Peacomm) and B3 (Waledac), both bots only contacted the remote server using direct IP addresses instead of domain names. Of course, we can also apply the IP addresses to our module. However, unfortunately, their contacting targets are either private IP addresses (192.168.X.X) or some hosts for which we could not get any useful information from the third parties.

B4 (JarBot) contacts a regular IRC server and the server has been operated for several years and we could not find any malicious keyword from search results.

4.3.3. Detection results of the system resource exposure model

Receiving the information of an automatic connection trial from the *human–process–network correlation analysis module*, the *system exposure analysis module* begins examining the target process.

When we test the functionality of each malware listed in Table 5, we find that the *system exposure analysis module* detects most of the malicious operations because many of these malware programs access system resources anomalously. The detection results are summarized in Table 7 (marked with “S”).

It only misses 2 malicious operations of “B6 (Flux)”, the first operation is to *operate file* which moves/creates a file in the host, and the second operation is to *capture screen* which takes a snapshot of the current screen.

When we analyze their resource exposure patterns, we find that their operations are very similar to normal programs. In the case of *operate file*, malware just creates a file under its permission and reports its success to a remote server. In the *capture screen* case, malware captures the current screen, saves in its local folder, and delivers captured screen information to a remote server. Both operations (in the point of host view) are very similar to resource exposure patterns of normal applications – creates a file and saves it in its local folder. However, we believe that these operations will be detected by the *network information trading analysis module*, because they distribute more information to the outside.

4.3.4. Detection results of the network information model

After notifying an automatic connection, the *network information trading analysis module* captures network traffic between the process (not a host) that issued an automatic connection and some remote server(s). If the process sends more packet/bytes than receives packets/bytes, our module considers it anomalous.

As listed in Table 7 (marked with “N”), the *network trading information analysis module* detects most malicious operations. It misses 8 malicious operations related to *download updates* and *file modification or operation*. In the case of the *download updates*, the process gains more data, so that our module cannot detect an anomaly. In addition, sometimes a botmaster sends commands frequently to an infected host, but does not require an answer. In this case, a bot also obtains more data. In terms of the aggregated view, our module detects all massive outgoing connection trials, such as DDoS and flooding.

Table 7
Detection results of all modules (shaded cells represent functionalities provided by malware. Each "P", "S", and "N" denotes each *process reputation analysis*, *system resource exposure analysis*, and *network information trading analysis module* detect the functionalities, respectively.

Functionality	*B1*	*B2*	*B3*	*B4*	*B5*	*B6*	*B7*	*B8*	*B9*	*B10*	*B11*	*B12*	*B13*	*B14*	*B15*	*B16*	*B17*
Operate file						P,N		P,S	P,S,N	P,S,N			P,S,N				
Modify file						P,S											
Kill process		S,N				P,S		P,S		P,S,N							
Capture Desktop						P,S,N											
Capture screen						P,N		P,S,N		P,S,N	P,S,N				P,S		
DDoS											P,S,N						
Flooding												P,S,N					
Create Shell													P,S,N		P,S,N		
Download update							P,S							P,S			
Steal Key	P,S,N	S,N						P,S,N									
Spam Mail Send	P,S,N																
Network Scan	P,S,N																
Other Operation			S,N	S,N	P,S											P,S,N	P,S

4.3.5. Correlated detection results

If any of the above modules determines its decision, the decision result is delivered to the *correlation engine*. Based on all delivered results, the *correlation engine* makes a final decision for a process.

When we test malicious operations, the *correlation engine* can detect all malicious operations by bots. As we discussed before, even though some module misses an anomaly of an attack, other modules will complement it, thus our combined results can still detect all attacks and the combined results are shown in Table 7.

4.4. False positive test results

In order to determine whether our modules misjudge benign processes as malicious or not, we have tested 1165 benign processes in **SET-2**, representing 71 distinct normal programs. They are general Windows applications programs such as browsers (e.g., IE, Chrome, Firefox), P2P software (e.g., Bittorrent, Emule, Skype), AV tools (e.g., McAfee, MS windows defender, Sophos), IM tools (e.g., Skype, QQ, Google Talk), office programs (e.g., Word, Powerpoint), multimedia programs (e.g., Gom, Kmplayer) (more shown in Table 4).

Among all 1165 processes (that have network communications), our *human–process–network correlation analysis module* detects 490 processes that produce automatic network connections. Thus, these 490 processes are further investigated by other three modules. *Process reputation analysis module* detects 2 processes as suspicious, *system resource exposure analysis module* considers 14 processes suspicious, and *network information trading analysis module* detects 237 processes as suspicious.

In the case of *process reputation analysis module*, 2 detected processes are browser processes and they visit some web sites. One of the sites sells some Chinese software programs and it is registered very recently (2011). In addition, several visited web sites are enlisted in blacklists and we find some malicious keywords such as spam from web search results for the sites.

The 14 processes detected by *system resource exposure analysis module* are 5 Chrome browser processes, 3 Firefox browser processes, 1 Internet Explorer browser process, 1 Safari browser process, 2 Gom player (multimedia player) processes, 1 Seaport (MS search enhancement broker) process, and 1 MathType (mathematical equation editor). Most of the 10 browser processes are detected because they create new processes and access system folders. Gom player processes try to access other users' folders, Seaport process accesses Windows system folders, and Mathtype process create new processes. Thus, our *system resource exposure analysis module* considers them suspicious.

Network information trading analysis module detects 253 processes as suspicious, and most of the processes are browser and program update processes. They send some information to remote hosts with gaining less (or no) information. Since we have not captured payload information of network traces, we could not understand why they send out more information. However, we infer that they might send information of current status of a host to remote servers or send query to remote mail servers to check new update.

Even though each module misjudges some benign processes as suspicious, it does not mean that our *correlation analyzer* regards them as malicious. Since *correlation analyzer* correlates all results from three modules, it determines a process as malicious only if two or three modules misjudge at the same time. In our test, we find 8 processes are misjudged by multiple modules, i.e., we have 8 false positives reported. Among them, 7 processes (4 Chrome browser, 1 Internet Explorer, 1 Gom player, and 1 Seaport processes) are detected by *system resource exposure analysis* and *network information trading analysis module*, and 1 process (Chrome browser process) is detected by *system resource exposure analysis* and *process reputation analysis module*. Thus, the false positive rate of our system is only 0.68% (8 out of 1165). We consider it very low that only 8 false positives found in about a week of testing time on all 8 real-world heavily-used machines. If we can apply a whitelisting approach, our false positives could be easily significantly reduced.

To further have a sense on whether the accuracy of EFFORT is reasonable in the real world, we note that a state-of-the-art related study, AccessMiner [20], reported to have a malware detection rate of 89.5% and a false positive rate of 3.4%. Of course, we do not attempt to directly compare EFFORT with AccessMiner to claim which one is better, because they are evaluated on different datasets and they use different features. The point is that, we can clearly see EFFORT has a very decent accuracy, comparable to (or even better) than some state-of-the-art work.

4.5. Efficiency of EFFORT

To show the efficiency of EFFORT, we measure how many processes are deeply/heavily investigated in real world. It can be understood by measuring how many automatic connections there are because our system only focuses on processes generating automatic network connections. We analyze the number of automatic connections and processes producing them in **SET-2**.

We find 231,613 connection trials in **SET-2**. Among them, 136,265 connections are automatically generated. The rate is quite high (58.83%). However we find that most of them are heading to hosts in local networks. For example, *spoolsv.exe* and *taskeng.exe* are the Windows system processes handling network or printing services and they contact network printers or sharing folders in local networks. In addition, *synergyc.exe* is a process to share mouse or keyboard with multiple hosts. They generate a lot of automatic connections to poll hosts or printers in the local networks.

We consider that we could ignore these automatic connections contacting local trusted networks, because it is very unlikely that a botmaster runs a server in the local network. There are 111,699 connections to local networks, and by removing them we finally have 24,566 connections, which is 10.6% of all connections. Among 24,566 connections, 10,154 connections (more than 40% of them) head to well-known web sites such as Google.com and Yahoo.com. The connections visiting to these domains could be ignored if a whitelist approach is employed. To be conservative, we let EFFORT keep investigating all the automatic connections.

4.6. Performance overhead of EFFORT

We have measured the overhead of each module to further verify the efficiency of EFFORT. In this measurement, we first show how our module in host (i.e., *system resource exposure analysis module*) affects the system and other applications, and then we will show the overhead of our network modules and demonstrate how they are scalable in real-world deployment.

To measure the performance overhead of the module in host, we use two metrics: *memory usage* and *program delay*. The *memory usage* represents how our modules consume the resources of memory and the *program delay* represents how our modules make the programs slow down when our modules are running. To measure the *program delay*, we select three types of test programs: Internet Explorer which produces network connections frequently, Calculator which mainly uses CPU, and Notepad which produces some disk operations. We compare the running time of these programs between when our modules are running and not.[2] In the case of the Internet Explorer, we simply visit one web site (Yahoo.com) and close. We divide some numbers using Calculator and read/edit/save a file using Notepad.

Table 8
Overhead of human–process–network correlation analysis module.

Item	*w/o module* (ms)	*With module* (ms)	*Overhead* (%)
Internet Explorer	177	179.4	1.35
Notepad	4206	4218	0.29
Calculator	26	26	0

Table 9
Overhead of system exposure analysis module.

Item	*w/o module* (ms)	*With module* (ms)	*Overhead* (%)
Internet Explorer	177	185.1	4.51
Notepad	4206	4463	6.12
Calculator	26	26.3	1.15

We send queries of 100 domains, which are randomly selected from **SET-2**, to *whois server, blacklist servers, and search engines* to measure the performance of *process reputation analysis module* and measure how long they take to get responses. We run this test 10 times (i.e., in each run, we send 100 randomly selected queries) and measure the average time to receive all responses.

Overhead of human–process–network correlation analysis module: As shown in Table 8, the overhead of this module is 1.35% at maximum and even 0% (i.e. our module does not affect other programs at all). In addition, this module only consumes 1.81 MB of memory. Thus, we believe that the overhead of this module is nearly ignorable.

Overhead of system exposure analysis module: We expect this module will show relatively high overhead. Since it has to monitor a lot of system calls which are frequently called by a process, it is very hard to reduce the overhead of this module.

When we measure the overhead, we observe that it consumes 9.18 MB memory and produces overhead around 6% at maximum and 1 % at minimum, as presented in Table 9.

The overhead seems to be not so high and even very low in some case. In addition, our module does *not* need to monitor *all* processes *all* the time. The *system exposure analysis module* only investigates a process when the process issues automatic connections to remote sites. In our real-world test on dataset **SET-2**, the automatic connections to remote sites happen very rarely, around 10% of all connection trials. Moreover, since a process is likely to generate multiple automatic connections (averagely 278 automatic-connections/process in the case of **SET-2**), the number of processes that the module needs to investigate is very small. Hence, we also consider that the overhead from this module is low.

Overhead of process reputation module: When we measure the time to collect information for this module, we find that collecting information related to *whois* and *blacklist* takes about 1.335 s per sending one query, and

[2] When we performed this test, we run a test program 5 times and calculate the average value.

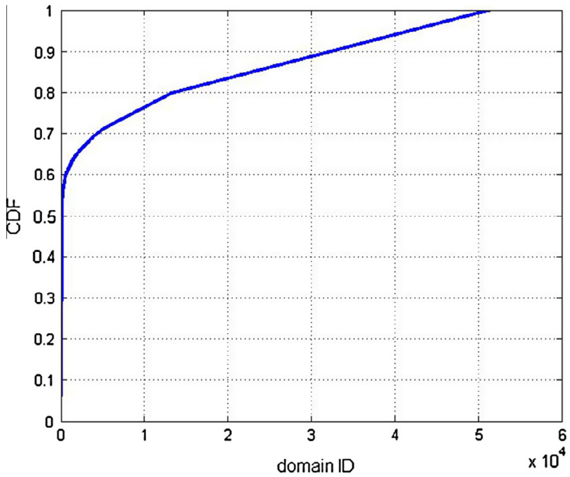

Fig. 2. CDF plot of percentage of queries to domains (*x*-axis denotes each individual domain, *y*-axis denotes cumulative distribution of percentage of queries to a domain).

gathering information from *search engine* takes 0.149 s per sending one query.

The results seem reasonable. However one concern is that is it scalable when there are many machines in the network communicating with the Internet? We argue that we do not send a query for every DNS request, because it is very likely that this domain (particularly when it is normal, popular, or frequently visited) is already queried before (either by this host or any other host in the network) thus we already have the cached information. To understand whether it is true or not, we investigate DNS query patterns of a large campus network, which consists of around 25,000 active unique hosts during monitoring, as shown in Fig. 2. We have captured all DNS queries produced by the campus network users for a day. The number of captured DNS queries is around 200,000 and they are targeted to around 50,000 domains. We find that most DNS queries are targeted to a relatively small stable set of domains. Among these 50,000 domains, top 10% of them cover more than 70% of all DNS queries and top 30% cover around 80% of all DNS queries. It is inferable that *process reputation analysis module* does not need to collect information for *all* DNS queries and the frequency of information collection might be very low. Thus, even though the number of hosts in a network can increase, we may not worry too much about the increased distinct DNS queries.

Overhead of other modules: Unlike the previous modules, the other modules of the *network information trading analysis module*, and the *correlation analyzer* exists in the other host(s) and they mainly monitor light-weight network traffic or receive results from other modules. Thus they do not affect the performance of the host/network directly.

5. Related work

There have been several approaches to detect bots at the network level. They detect bots mainly based on network traffic pattern analysis [17] or aggregating network flows [32] or through network behavior correlation analysis [11,10,12]. Our work is different from the above, because we design both of new network level sensors and host level sensors.

Detecting bots at the host level is also popular due to its effectiveness. They employ several interesting techniques to detect bots such as tainting memory and system resources [4,28] and examining the system call sequences/graphs [18]. Although they detect malware accurately, they could cause high overhead on the host. Our work designs several new host level sensors without analyzing *all* running processes *all* the time, but only investigating the process when necessary.

There are also several interesting studies related to detect bot malware. Liu et al. [21] proposed a host-based approach of executing malware in a virtual machine to detect bot-like behaviors or characteristics such as making automatic network connections. EFFORT is a host–network cooperated approach to protect real-world user machines. In addition, our approach and major detection features/modules are different. Lanzi et al. [20] proposed an approach of detecting malware in the host by investigating the access patterns of the process. Our work differs from it because we use different features at host level (e.g., socket creation) and detection models. Moreover, our work analyzes the process only when necessary. Zeng et al. [33] also proposed to detect bots combining information from both host and network levels. This work uses network sensors to trigger host analysis, thus it suffers from the same limitations of previous network-based detection approaches. If a bot can evade the network level monitoring, it evades their detection system. EXPOSURE system has been proposed to detect malicious DNS based on several features [1]. Although some domain registration features of the *process reputation analysis module* are similar to EXPOSURE, they are only small parts in our engine. Cui et al. provided an approach to detect malware by monitoring automatically generated connections [6]. Our work differs from it in that we consider more factors and issues (such as foreground, helper process for DNS relaying) and we do *not* use whitelisting. Moreover, this is only one module in our whole detection framework.

In [30], Trestian et al. uses the Google search engine for network traffic measurement, and our approach of identifying reputation of a process also employs search engines. Our work differs in its main goal (for security) and detection features/models. Also, while they only use IP address for their query, we use the process and domain name as well.

6. Limitations and future work

As we all know that no detection system is perfect. Our EFFORT system is no exception. For instance, a bot might decide to (mis) use benign domains (e.g., Google, Flickr) as a side channel [25,3] for concealed C&C communication to evade our *process reputation analysis module*. However, the efficiency, bandwidth, or realtimeness of such C&C is likely restricted compared to regular C&C, which could downgrade the utility of bots. The bot can also choose not to access unnecessary system resources to avoid the detection of our *system resource exposure analysis module*. However, this will essentially render the bot less useful or no profit to the botmaster. The bot can also increase the incoming traffic or decrease the outgoing traffic in order to evade our *network information trading analysis module*. However, the former (increasing incoming traffic from C&C server) could increase the exposure (and detection probability) of C&C server to regular security monitoring tools/IDSes that mostly monitor inbound traffic. And the later (decreasing outbound traffic) will likely significantly decrease the utility of bots in performing malicious activities (information theft, spamming or DDoS). To evade our *human–process–network correlation analysis module*, the bot program should not produce automatic network connections itself. Instead, it may fork another process to do so or even consider compromising a benign program (may have frequent human interactions) to do so. However, keep in mind that we do not use whitelisting and furthermore, there will be clear coordination and communication between these processes and they could be correlated together in analysis to discovery the process group. In short, we do acknowledge these limitations but we think the chance that a bot evades all our modules is very slim unless they sleep or behave like normal programs (in which case we still achieve the goal of deterrence because they are not usable to botmasters then). Although not perfect, we believe EFFORT raises a higher bar, and this is a **timely effort** and a **right direction** in the malware battle.

Our reputation module mainly assumes that bots will use DNS to contact their master. However not all bots may use DNS, some bots use IP address directly. Our reputation model is easily to be extended to handle IP address as well.

In the future, we will further improve the *process reputation analysis module* with more robust features and intelligent context analysis. For instance, we plan to improve the current *context* analysis in the case of malware keywords appearing in the search results.

7. Conclusion

In this paper, we study various features at network and host levels and choose promising features that enable to detect bots both effectively and efficiently, a very challenging research problem in the domain. We propose a novel host–network cooperated detection approach with correlative and coordinated analysis and develop a first-of-its-kind prototype system EFFORT. In our extensive evaluation on real world data, we show that our system can detect bots accurately with very low false positive and low overhead.

Appendix A

A.1. SVM classifier

For the *process reputation model*, we use a SVM classifier. Here we briefly talk about the SVM classifier.

To start with the simplest case, we assume that there are two classes and they can be separated by a linear function. More formally, given training examples x_i and a classifier y_i, if we assume that those two classes are denoted as 1 and -1 (i.e. $y_i \in \{-1, 1\}$), the training examples which lie on the hyperplane satisfy the following equation.

$w \cdot x + b = 0$, where w is a normal vector and $b/\|w\|$ is a perpendicular distance from the hyperplane to the origin.

From the above equation, we can find the hyperplanes which separate the data with maximal margin by minimizing $\|w\|$ under the constrains of $y_i(x_i \cdot w + b) - 1 \geqslant 0$. To solve this equation, we will apply a *Lagrangian formulation*, and then we will have a primal form – L_p – of the *Lagrangian* [2]. It is described as the following equations.

$$L_p \equiv \frac{1}{2}\|w\|^2 - \sum \alpha_i y_i (x_i \cdot w + b) + \sum \alpha_i, \tag{A.1}$$

where α_i is a *Lagrangian multiplier* and $\alpha_i \geqslant 0$.

Now, we have to minimize L_p with respect w and b, and it gives us two conditions of $w = \sum \alpha_i y_i x_i$ and $\sum \alpha_i y_i = 0$. In addition, we can substitute these conditions into L_p, since they are equality in the dual formulation. Thus, we can get dual form – L_d – of the *Lagrangian* like the following equation.

$$L_d = \sum \alpha_i - \frac{1}{2} \sum \alpha_i \alpha_j y_i y_j x_i \cdot x_j \tag{A.2}$$

Finally, we can get our SVM classifier through maximizing L_d. If we cannot separate the data by a linear function, we have to extend the original set of training examples x_i into a high dimensional feature space with the mapping function $\Phi(x)$. Suppose that training examples $x_i \in R^d$ are mapped into the Euclidean space H by a mapping function $\Phi{:}R^d \rightarrow H$, we can find a function K such that $K(x_i,x_j) = \Phi(x_i) \cdot \Phi(x_j)$ (a.k.a. "kernel function"). We can replace the inner-product of the mapping function by the kernel function and solve the problem with similar approach of the linearly separable case.

A.2. One-Class SVM (OCSVM)

The One-Class SVM (OCSVM) has been proposed to create a model with only one side of information [24]. In this paper, we use OCSVM to build the *system resource exposure model*. The OCSVM maps training examples into a feature space and finds a hyperplane which can best separate training examples from the origin. For instance, given training examples x_i, if we assume that there are two planes denoted as "+" and "−" across an origin, the OCSVM will assign all known examples x_i into a one of the planes (i.e. "+" plane or "−" plane).

Similar to generic multi-class SVM classifier, the OCSVM needs to find a hyperplane with maximal geometric margin and it is described as solving the *Lagrangian* equations of (1) and (2) in Appendix A.1 (more details about the OCSVM can be found in [24]). In this model, we will find a hyperplane to assign all benign examples (i.e. the Y_i features of the benign processes) into the "+" plane and anomaly examples into the "−" plane.

The weights for the *correlation engine* can also be determined by the OCSVM. To do this, we first collect detection results of other modules of benign processes and then we use these results as features for the OCSVM classifier. Similar to the above approach, we try to find (a) hyperplane(s) which map(s) all (or most) benign processes to one plane.

References

[1] Leyla Bilge, Engin Kirda, Christopher Kruegel, Marco Balduzzi, Exposure: finding malicious domains using passive dns analysis, in: Proc. of NDSS, 2011.

[2] J.C. Christopher-Burges, A tutorial on support vector machines for pattern recognition, Journals of the Data Mining and Knowledge Discovery 2, 1998 121–167.

[3] Sam Burnett, Nick Feamster, Santosh Vempala, Chipping away at censorship firewalls with user-generated content, in: Proceedings of the 19th USENIX Conference on Security, USENIX Security'10, 2010.

[4] Juan Caballero, Pongsin Poosankam, Christian Kreibich, Dawn Song, Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering, in: Proceedings of the 16th ACM Conference on Computer and Communication Security, November 2009.

[5] Corinna Cortes, V. Vapnik. Support-Vector Networks, Mach. Learn. 20(3), 1995 273–297.

[6] Weidong Cui, Randy H. Katz, Wai tian Tan, BINDER: an extrusion-based break-in detector for personal computers, University of Berkeley Technical Report No. UCB/CSD-4-1352, 2004.

[7] Detours, Software packaged for detouring win32 and application apis. <http://research.microsoft.com/en-us/projects/detours/>.

[8] EasyHook, Easyhook – the reinvention of windows api hooking. <http://easyhook.codeplex.com/>.

[9] Tal Garfinkel, Mendel Rosenblum, A virtual machine introspection based architecture for intrusion detection, in: Proceedings of the 10th Annual Network and Distributed System Security Symposium, February 2003.

[10] Guofei Gu, Roberto Perdisci, Junjie Zhang, Wenke Lee, BotMiner: clustering analysis of network traffic for protocol- and structure-independent botnet detection, in: Proceedings of the 17th USENIX Security Symposium (Security'08), July 2008.

[11] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, Wenke Lee, BotHunter: detecting malware infection through IDS-driven dialog correlation, in: Proceedings of the 16th USENIX Security Symposium (Security'07), August 2007.

[12] Guofei Gu, Junjie Zhang, Wenke Lee, BotSniffer: detecting botnet command and control channels in network traffic, in: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08), February 2008.

[13] Ramakrishna Gummadi, Hari Balakrishnan, Petros Maniatis, Sylvia Ratnasamy, Not-a-Bot (NAB): improving service availability in the face of botnet attacks, in: Proceedings of Symposium on Networked System Design and Implementation (NSDI), April 2009.

[14] Thorsten Holz, Christian Gorecki, Felix Freiling, Detection and mitigation of fast-flux service networks, in: Proceedings of NDSS Symposium, February 2008.

[15] Nicholas Ianelli, Aaron Hackworth, Botnets as a vehicle for online crime, in: Proceedings of 18th Annual FIRST Conference, June 2006.

[16] Xuxian Jiang, Xinyuan Wang, Dongyan Xu, Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, October 2007.

[17] Anestis Karasaridis, Brian Rexroad, David Hoeflin, Wide-scale botnet detection and characterization, in: Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets, April 2007.

[18] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, Xiaofeng Wang, Effective and efficient malware detection at the end host, in: Proceedings of 18th USENIX Security Symposium, August 2009.

[19] Taku Kudo, Tinysvm: support vector machines. <http://chasen.org/taku/software/TinySVM/>.

[20] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, Engin Kirda, AccessMiner: using system-centric models for malware protection, in: Proceedings of 17th ACM Conference on Computer and Communications Security, June 2010.

[21] Lei Liu, Songqing Chen, Guanhua Yan, Zhao Zhang, BotTracer: execution-based bot-like malware detection, in: Proceedings of International Conference on Information Security (ISC), 2008.

[22] MalwareDomains, Dns-bh malware domain blacklists. <http://www.malwaredomains.com/wordpress/?p=1411>.

[23] MicroSoft MSDN, Windows hook functions. <http://msdn.microsoft.com/en-us/library/ff468842(v=VS.85).aspx>.

[24] B. Scholkopf, J.C. Platt, J.Shawe-Taylor, A.J. Smola, R.C. Williamson, Estimating the support of a high-dimensional distribution, Technical report, Microsoft Research, MSR-TR-99-87, 1999.

[25] Kapil Singh, Abhinav Srivastava, Jonathon T. Giffin, Wenke Lee, Evaluating email's feasibility for botnet command and control, in: Proceedings of Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2008), 2008, pp. 376–385.

[26] SPAMHAUS, The SPAMHAUS Project. <http://www.spamhaus.org/>.

[27] SRI-International, An analysis of conficker C. <http://mtc.sri.com/Conficker/addendumC/>.

[28] Elizabeth Stinson, John C. Mitchell, Characterizing the remote control behavior of bots, in: Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), 2007, July 2007.

[29] Elizabeth Stinson, John C. Mitchell, Towards systematic evaluation of the evadability of bot/botnet detection methods, in: WOOT'08: Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies, Berkeley, CA, USA, 2008. USENIX Association.

[30] I. Trestian, S. Ranjan, A. Kuzmanovic, A. Nucci, Unconstrained endpoint profiling (googling the internet), in: Proceedings of ACM SIGCOMM 2008, Mayt 200.

[31] WinPcap, The industry-standard windows packet capture library. <http://www.winpcap.org/>.

[32] T.-F. Yen, M.K. Reiter, Traffic aggregation for malware detection, in: Proceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA2008), July 2008.

[33] Yuanyuan Zeng, Xin Hu, Kang G. Shin, Detection of botnets using combined host- and network-level information, in: Proceedings of Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010), June 2010.

Seungwon Shin is a Ph.D. student in the Department of Electrical and Computer Engineering at Texas A&M University working with Dr. Guofei Gu in SUCCESS Lab. He received his B.S. and M.S. degrees in Electrical Engineering from KAIST in Korea. His research interests include analyzing and detecting botnet, and protecting cloud computing environments from threats.

Zhaoyan Xu, third year Ph.D. candidate in Texas A&M University, College Station. He received his bachelor degree from Shanghai University, China. His research interest focuses on system/network security and malware analysis. He is currently working in Secure Communication and Computer System Lab at Texas A&M University.

Guofei Gu is an assistant professor in the Department of Computer Science & Engineering at Texas A&M University (TAMU). Before coming to Texas A&M, he received his Ph.D. degree in Computer Science from the College of Computing, Georgia Institute of Technology. His research interests are in network and system security, such as malware analysis/detection/defense, intrusion/anomaly detection, web and social networking security. Dr. Gu is a recipient of 2010 NSF CAREER award and a corecipient of 2010 IEEE Symposium on Security and Privacy (Oakland10) best student paper award. He is currently directing the SUCCESS (Secure Communication and Computer Systems) Lab at TAMU.