Using Failure Information Analysis to Detect Enterprise Zombies

Zhaosheng Zhu[1], Vinod Yegneswaran[2], and Yan Chen[1]

[1] Department of Electrical and Computer Engineering, Northwestern University
{z-zhu,ychen}@northwestern.edu
[2] Computer Science Laboratory, SRI International
vinod@csl.sri.com

Abstract. We propose failure information analysis as a novel strategy for uncovering malware activity and other anomalies in enterprise network traffic. A focus of our study is detecting self-propagating malware such as worms and botnets. We begin by conducting an empirical study of transport- and application-layer failure activity using a collection of long-lived malware traces. We dissect the failure activity observed in this traffic in several dimensions, finding that their failure patterns differ significantly from those of real-world applications. Based on these observations, we describe the design of a prototype system called Netfuse to automatically detect and isolate malware-like failure patterns. The system uses an SVM-based classification engine to identify suspicious systems and clustering to aggregate failure activity of related enterprise hosts. Our evaluation using several malware traces demonstrates that the Netfuse system provides an effective means to discover suspicious application failures and infected enterprise hosts. We believe it would be a useful complement to existing defenses.

1 Introduction

Due to the persistent and ubiquitous nature of the Internet's background radiation [35], modern enterprise networks have become relentless targets of attacks from a plethora of Internet malware including worms, self-propagating bots, spamming bots, client-side infects (drive-by downloads) and phishing attacks. Estimates on the number of malware instances released vary vastly (between ten of thousands to more than hundred thousand per month) depending on census methodologies [16, 31]. However, there is consensus that malware is becoming increasingly prevalent, sophisticated, and a formidable threat not just to network communications but also as a purveyor of data and identity theft. Network security analysts in today's enterprise networks rely primarily on a combination of network intrusion detection systems (NIDS) [36, 41] and antivirus (AV) systems to shield enterprise networks from this deluge of malware.

A NIDS passively monitors packets on the network wire and uses rules to discover suspicious activities, such as scans and exploit attempts, directed against systems in the network. Knowledge-based and behavior-based detection are two

Y. Chen et al. (Eds.): SecureComm 2009, LNICST 19, pp. 185–206, 2009.

fundamental approaches to intrusion detection [14]. Knowledge-based intrusion detection systems [41] use signatures of well-known exploits and intrusions to identify attack traffic. However, reliable and accurate performance requires constant maintenance of the knowledgebase to reflect the latest vulnerabilities. In contrast, behavior-based intrusion detection techniques [27] compare current activity with a predefined model of normal behavior and flag deviants from known models as anomalies. A drawback with many behavioral approaches is the inherent difficulty of building robust models of normal behavior whose incompleteness results in high false alarm rates.

Contemporary AV software monitors end hosts by performing periodic system scans and real-time monitoring, checking existing files and process images with a dictionary of malware signatures that is constantly updated. Certain vendors also incorporate heuristic detection engines that identify infections based on static traits (*e.g.,* whether it is packed) or approximate behavioral profiles of known malware. Despite their ubiquity and sophistication, most AV systems have been shown to have unsatisfactory detection rates [10] especially in early days of an outbreak. Our experience at honeynets shows that the median day-zero detection rate for 30 AV vendors is around 82% [26]. The proliferation of the recent Conficker A and B worms offers further testament to the inefficacy of current AV systems. By leveraging a well-publicized Windows RPC vulnerability (MS08-67) [32], Conficker has successfully infected millions of hosts [17, 18], and in the early days of the outbreak only 3/39 AV engines were able to detect this binary as being malicious [43]. Like most malware, Conficker disables AV updates after infection, so subsequent signature updates by AV vendors were not particularly effective in curtailing this worm. In summary, the remarkable success of a scan-and-infect worm such as Conficker (seven years after Code Red I [13]), underscores why network security analysts need better tools to understand, react to, and cope with infections in their enterprises.

Our approach. In this paper, we introduce a new behavior-based approach to detect infected hosts within an enterprise network. Our objective is to develop a system that is independent of malware family and requiring no apriori knowledge of malware semantics or command and control (C&C) mechanisms. We devise an approach that is motivated by the simple observation that many malware communication patterns result in abnormally high failure rates. While prior efforts have tried to exploit this in the specific context of portscans [28] or studied types of failures [44, 39], we extend this to broadly consider a large class of failures in both transport and application levels. We have developed a prototype system called Netfuse that correlates network and application failures to detect infected hosts within enterprise networks. The event correlation engine of our system is inspired by prior systems such as BotHunter [23]. While BotHunter relies on exploit signatures from Snort, an important distinction of our approach is that it requires no specific knowledge of malware. Instead, Netfuse relies on application knowledge that it obtains from network protocol analyzers such as Wireshark [8] and L7 filters [5]. In some sense, Netfuse could be considered a behavior-based detection system whose model for malicious behavior is

derived from underlying protocol analyzers. However, its novelty lies in its use of multipoint failure monitoring for support vector machine (SVM)-based classification of malware failure profiles. We believe that Netfuse could be a useful sensor input to BotHunter.

The Netfuse system has several integral components. First, it has a protocol failure analysis component that is built on the Wireshark protocol analyzer. It specifically analyzes transport failures (TCP RSTs, ICMP) and application failures on common ports TCP/25 (SMTP), TCP/80 (HTTP), UDP/53 (DNS) and TCP/6667 (IRC). Furthermore, it uses L7 filters to detect when common protocols are observed in nonstandard ports (*e.g.*, HTTP or DNS activity on a high-order port) and routes them to the appropriate Wireshark protocol handler. Second, it has a lightweight DNS monitor that monitors DNS activity between enterprise clients and the DNS server. Finally, it has a clustering and correlation component that aggregates alerts observed by the two sensors producing a condensed summary of failure activity that classify anomalous activity. For every IP with failure activity, it computes four different scores: (*i*) composite failure (*ii*) divergence (*iii*) persistence and (*iv*) failure entropy. This information is used by an SVM driven classification engine to detect suspicious hosts. Furthermore, a cluster summary is produced that aggregates suspicious hosts with similar failure profiles. The combination of these scores and clustering enables security analysts to easily comprehend failure patterns in the enterprise and quickly identify suspicious hosts in the network. We find that our approach is effective in isolating the presence of a vast majority of contemporary malware without specialized signatures.

Contributions. The contributions of our work are as follows:

1. We describe application-aware failure monitoring as a new approach for identifying infected hosts and uncovering anomalies in enterprise traffic.
2. We develop a prototype implementation of the Netfuse monitor using Wireshark and L7 filters. An important aspect of the implementation is multipoint failure monitoring.
3. We develop an SVM-based classifier to identify infected hosts.
4. We use multiple network traces of malware and benign traffic to evaluate detection rates and the false positive rate of Netfuse.

The remainder of the paper is organized as follows. In Section 2, we provide an analysis of network and application failures that motivate the Netfuse system. In Section 3, we introduce our Netfuse prototype implementation. In Section 4, we describe our classification and clustering algorithm. Then we describe our in-situ and online experiences with the Netfuse system and analyze results in Section 5. We survey related work in Section 6. We summarize our results and discuss future work in Section 7.

2 An Empirical Survey of Application Failure Anomalies

We explore reasons behind the occurrence of application failures in enterprise traffic. We begin with a case study analysis of the failure patterns of malware

using over 30 long-lived malware (5-8 hour) traces. We then examine failure profiles of several normal applications that may cause failures similar to malware including webcrawlers, P2P software and popular video sites. Then we discuss the potential and implications of using protocol failure anomalies to detect misbehaving clients in the enterprise network.

In the following, we define the term *failure* to broadly refer to both network and application failures. Network failure corresponds to presence of packets, indicating transport-level failures such as TCP RSTs and ICMP unreachable messages in the trace. Application failures indicate higher-level protocol failures as shown in Table 1.

Table 1. Commonly observed protocol failure messages

Protocol	Layer	Failure Types
DNS	Application	NXDOMAIN (No such domain)
HTTP	Application	400 Bad Request, 404 Not Found, 403 Forbidden, 411 Length Required 500 Internal Server Server, 501 Not Implemented
FTP	Application	Transient Negative Completion reply Permanent Negative Completion reply
SMTP	Application	Domain service not available, mailbox unavailable Syntax error, command not implemented Machine does not accept mail, mailbox unavailable User not local, requested mail action aborted
IRC	Application	No such nick, No such server No such channel, Cannot send to channel

2.1 Malware Trace Analysis

The first part of our analysis is a study of application failure patterns observed in contemporary Internet malware. We started with a corpus of 32 different malware instances that we each executed in a controlled virtual machine (VM) environment for several hours. The sources of the malware include our honeynet [43], malicious email attachments, and the Offensive Computing website [6]. To obtain accurate and complete results of network interaction, it was necessary to collect long-lived traces and to allow the hosts to communicate with the outside world. We collected `tcpdump` traces of all network activity, and we analyze the failure patterns found in these traces below.

We find that contemporary malware instances generate a diverse set of failures, in both the transport and application levels. Interestingly, we find that these failures could be attributed to a small set of causes, *i.e.*, broken C&C channels, scanning and spam delivery attempts. Furthermore, the volume of failure activity seems to be strongly correlated with the volume of overall network activity. For example, scanners tend to generate a lot of flows, many of which generate transport failures. Likewise, many malware instances periodically retry failed communication attempts, which results in larger network traces with redundant activity.

Among the 32 malware instances, eight did not generate failures. These include two worms, three IRC botnets, and three spyware instances. As the three IRC bots contacted the server successfully and did not receive any MOTD commands from the server, there were no failures. Likewise, the well-behaved spyware binaries simply contacted a few active websites.

Table 2 illustrates the distribution of failures by protocol for each of the malware instances that generated transport or application failures. First, we note that 24/32 botnet and worm instances generate some sort of failure (either application or transport). We find that most of them (18/24 instances) trigger DNS failures. Furthermore, malware with spam capabilities (notably Storm) also tends to produce high volumes of SMTP failures. Finally, malware with P2P C&C channels and malware with scanning behavior are also associated with abnormally high ICMP failures. We examine the failure breakdowns within each protocol in greater detail below and provide explanations for their causes.

DNS failures. In our analysis, we found that 18 malware traces contained DNS failures. All of these were due to unresolved domain names or NXDOMAIN responses from the DNS server. In many cases, particularly for IRC bots, these arise because the C&C server gets taken down by ISPs or is otherwise blocked by law enforcement. While many well-behaved applications terminate connection attempts after a few failed tries, we find that malware tends to be remarkably persistent in its repeated attempts to contact its C&C server. We also observed that for certain malware, there is built-in redundancy in that they will query a set of domain names for the remote server. Although some domains do not resolve, C&C communication will still continue based on the successful DNS lookups.

Table 2. Failure profile summary (in hourly rates) of 24 malware instances

Malware	Class	DNS rate	HTTP rate	ICMP rate	SMTP rate	TCP rate
Look2me	Spyware	5				
Wsnpoem	Spyware	15				
Bobax	HTTP botnet	148				191
Kraken I	HTTP botnet	348				
AgoBot	IRC botnet	5312				
Gobot	IRC botnet			891		9539
Sdbot	IRC botnet	2188				
Sdbot II	IRC botnet	53				
Spybot I	IRC botnet	283				1506
Spybot II	IRC botnet	16				50
Spybot III	IRC botnet	16				
Wootbot	IRC botnet					275
Irc.Webloit	IRC botnet					477
Nugache	P2P botnet					291
Storm I	P2P botnet	26		5432	284	73
Storm II	P2P botnet			27151		
Allaple	Worm	9		33413		5738
Grum	Worm	60		160		31330
Kwbot	Worm	37				
Mytob	Worm	221		385		53
Netsky	Worm	51012				
Protoride	Worm	503				151
Virut	Worm	222	10	409		14
Weby	Worm		67			24

In fact, for some bots, such as Kraken, DNS failures could be considered part of normal behavior. This malware uses a dynamic C&C-based communication structure that constructs a new list of C&C rendezvous points each day. The fully qualified domain name (FQDN) of the C&C server is constructed from a dynamically generated hostname (based on the date) and one of the following four base domain names: `.mooo.com, .dynserv.com, .dyndns.org,` and `.yi.org`. As long as the botmaster and the malware use the same algorithm to generate domain names, it is very easy for the botmaster to change the C&C server names and IP addresses to evade detection. While resolutions for most of these DNS domains are expected to fail everyday, the botmaster simply has to register one of the daily domains when he wants to instruct the bots to perform a task. Hence, a lot of DNS lookup failures are observed in the trace. For example, our trace shows that the host received 1740 DNS failures in about 5 hours, which is highly anomalous for a normal host. A similar strategy is also adopted by the recent Conficker worms [18].

SMTP failures. In our analysis, we found that SMTP failures result from spamming behaviors. A typical example is the Storm botnet, which also uses SMTP to generate emails for spam as well as propagation. Hence, its trace includes a flurry of SMTP activity and a lot of failures. Certain SMTP servers immediately close the connection after the TCP handshake. Other failures occur early in the SMTP connection setup, most common reason being "550 Recipient address rejected: User unknown". In our traces, we found hundreds of SMTP failures from several email servers. But these failures were not persistent, *i.e.*, Storm does not retry a rejected username on the same SMTP server. In certain traces of the Storm botnet, this spam behavior stops after an hour, suggesting that certain malware instances do eventually learn from failures (albeit after a long time). We feel that any malware that generates spam is bound to produce such failures. Besides Storm, there were other malware instances that attempted to send spam email, *e.g.*, Bobax, but could not succeed in establishing communication with the remote SMTP server.

HTTP failures. We found that the HTTP failures in our traces could be attributed to two reasons: (1) sending mal-formed packets for DoS attacks and (2) querying for a configuration file that has since been removed from the control server. For example, malware Mimail.L sends the following request to the target HTTP server to launch a DoS attack: "GET / HTTP/1.0" to port 80, followed by 2048 bytes of data to port 80. As a result, it receives a flurry of "HTTP 400" errors from the server implying "Bad or Malformed HTTP request". Certain other failures are due to the missing files in controlling servers. For example, clients infected with the Weby malware will try to get a configuration file from several servers. Since this file is removed in the servers, it results in "HTTP 404/File not found" errors, which are quite persistent. In our 5-hour trace, there were 335 "HTTP 404/File not found" failures.

IRC failures. For botnets that use the IRC protocol for communication and control, the following failure modes are common. Sometimes, the channel is

removed from a public IRC server, which results in IRC application failures like "no such channel". In certain other cases, the channel might be full due to too many bots, which would result in a "Cannot join channel" message.

TCP layer failures. We consider unproductive TCP flows *i.e.*, which do complete a TCP handshake and/or terminate the connection with RST prior to sending any payload. The prevalence of such unproductive flows (which also results from scanning behavior), is another characteristic of malware. For some malware instances, we observed that there were continuous TCP layer failures in certain ports. For example, some IRC botnet clients receive failures in the IRC port (TCP/6667) from the remote servers (either because the server has been taken down or because it is too busy). Certain Bobax clients receive failures in the SMTP (TCP/25) port from remote email servers because the client network has been blacklisted. While scanning is usually good evidence of malware, we find that persistent TCP failures from the same remote host could be another useful indicator of malware. For example, we observed that many IRC botnets generate TCP failures from being unable to contact a previously active C&C server that has since been taken down.

ICMP failures. In our analysis, we found that ICMP failures result from scanning behavior and communication patterns of P2P botnets such as Storm. As we discuss below, this is quite unlike normal P2P applications, such as BitTorrent and eMule, that generate few ICMP failures.

2.2 Failure Patterns of Normal Applications

The second part of our analysis studies failure patterns of normal applications. As studying failure patterns of all applications is outside the scope of this study, we focus on applications that one might typically expect to produce failure patterns similar to what was observed in the malware corpus that we analyzed. The goal of this study is to understand the degree to which malware failure patterns could be used to distinguish malware traffic from other benign enterprise traffic. Our Netfuse system uses these network failures as symptoms to detect suspicious hosts. Thus, these results could inform the feasibility and design of the Netfuse system and help us prioritize failure patterns that are used for detection. Specifically, we focus our investigation on three classes of applications, which at the first glance may cause similar failures: web crawler, P2P applications (BitTorrent, eMule), and online video service (youtube).

We collected several long-lived traces for each of these normal applications, in order to get a good understanding of the types of failures they generate. Table 3 provides a summary of these traces.

Webcrawler. webcrawlers, popularly known as webspiders or webrobots, are automated scripts that systematically scan all web-pages in a site looking for specific types of content. These are commonly used by search engines to build automated meta-data (indexes) of public web-pages, but are also used for mirroring websites, data mining, and by other web-based applications such as mashups

Table 3. Normal application trace summary

Type	Site	Size	Time	Pkts	# URLs
Webcrawler Mirror	news.sohu.com	3.1 GB	2 days	3577674	25334
	amazon.com	1.9 GB	2 days	2058630	23111
	bofa.com	144 MB	12 hours	186711	4141
	imdb.com	252 MB	16 hours	333583	8113
P2P	BitTorrent	6.1 GB	18 hours	7338627	n/a
P2P	eMule	1.3 GB	1 day	1982682	n/a
Video	youtube.com	16MB	2 hours	25498	n/a

Table 4. Failure profile summary (in hourly rates) for normal applications

Application Name	HTTP	ICMP	TCP
	rate	rate	# ports / rate
Web crawler(sohu)	1.4		1/0.4
Web crawler(amazon)			1/1.4
Web Crawler(imdb)	0.04		1/0.2
Web Crawler(bofa)	0.8		1/0.9
BitTorrent	0.6		382/333
eMule		68	839/370

and portals. Since webcrawlers have become very popular and they follow hyperlinks in an automated fashion, one might expect such systems to frequently stumble upon many failed links and generate HTTP failures. Hence, we pick them as the first class of application to study.

We used the default settings and -m (mirror) option in `wget` [3] that forces wget to act as a webcrawler, recursively following all links in a given site. until all the pages have been downloaded. We collected traces from crawling four popular websites in the US and China including `bofa.com, amazon.com, imdb.com` and `sohu.com`. Each crawl took 1-2 days and involved 144 MB to 3 GB of data transfer. We found that the webcrawler produced very few HTTP and transport failures. As an example, for the website `news.sohu.com`, there were only 18 transport layer (TCP) failures and 66 HTTP failures in 2 days. Other websites also show the similarly low failure patterns as shown in Table 4. As one might expect, we find that in webcrawlers, HTTP failures are restricted to "HTTP 404/File not found" messages.

P2P applications. We select two popular peer-to-peer (P2P) software programs for our analysis: BitTorrent and eMule. BitTorrent and eMule are P2P file sharing protocols used to transfer large amounts of data such as media files, software, and OS distributions. A single large file is broken up into pieces, which are replicated and distributed among a set of peers. In BitTorrent, the publisher of the file acts as the *first seed*, and every peer who downloads the data also uploads the content to other peers. A client wishing to download the file first obtains the meta-data file, called the *torrent*, which specifies where to download the pieces. Thus, a single HTTP request for a large file is translated into several small data requests to various peers in the network. eMule is similar in concept but implements a different protocol based on Kademlia [4].

Since the status of peers in both of these networks can dynamically change (from online to offline), we expect these P2P applications to have many failures. We used BitTorrent to download a popular Linux distribution (Fedora 10) and monitored the activity of this peer for one day. It turns out there were very few (11) ICMP failures and HTTP failures, but many TCP failures. Likewise, we used eMule to download another popular Linux distribution (Ubuntu) and monitored its activity for a day. It had many ICMP and TCP failures. An important difference between transport-level failure profiles of BitTorrent and the malware we analyzed is that for BitTorrent the TCP failures happen on a large set of ports. This did not occur in the malware traces, *i.e.*, failures were restricted to fewer ports and typically occurred in one or two ports. As an example, most TCP failures with the Storm worm were dominated by its activity on port 25/TCP (arising from its spam campaigns and unrelated to its P2P communication).

Online video service. `YouTube.com` is one of the largest and most popular websites that provide online video hosting service. Users can upload, view, and publicly share video clips. In this experiment, we collected traces by opening videos from `youtube.com`, and then keeping the browser open for several hours. In analyzing the trace, we found that there were no transport-layer failures. While we did find several "HTTP 304/Not Modified" errors, we did not find any other application-level failures. Since "HTTP 304/Not Modified" messages were not found in the malware traces, we infer that this might be an error code to be considered in a whitelist.

2.3 On the Potential of Failure Analysis to Uncover Suspicious Activities

We summarize the results of our exploratory empirical analysis on the utility of failure profile analysis. After our analysis of a collection of traces from both malware and benign applications, we find several notable differences in failure pattern between malware and normal applications that could be exploited in network-based detection systems.

1. Failures in malware occur frequently in both the transport and application levels. In general, failures are rare for normal applications, except for certain P2P protocols that can generate high volumes of transport failures. Thus *high volume* of failure traffic could be a useful indicator of malware.
2. DNS failures and in particular NXDOMAIN errors are common among malware applications and relatively infrequent in normal applications. Furthermore, these failures tend to *persist* (repeat with high frequency) in malware.
3. Failures in malware applications tend to be restricted to a few ports and often a few domains. Thus, malware failure patterns tend to have *low entropy*.

3 Architecture

In the prior section, we explored the possibility to using failure information to detect suspicious hosts in the enterprise network. Here, we describe the system framework and our prototype implementation of a system that realizes our ideas.

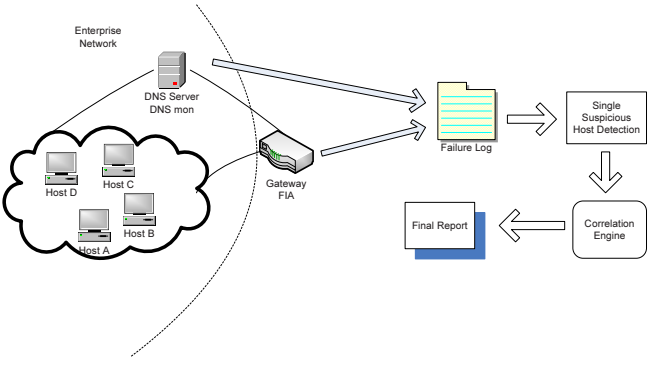

Fig. 1. Netfuse multi-point monitoring architecture

3.1 System Overview

As shown in Figure 1, Netfuse is composed of three parts: the failure information analysis (FIA) engine, DNSMon and the correlation engine. The FIA engine will typically be deployed on the perimeter of the enterprise networks. The major function of this component is to extract the failure information by looking at all packets that transit the enterprise gateway router. It will generate the failure information if any, by including both flow-level and application-level information (if available). The DNSMon system monitors interaction between enterprise clients and the local DNS server.

After the failure information is collected, the correlation engine implements a diagnostic algorithm to classify hosts according to their failure profiles and to group those suspicious hosts with similar failures. It then generates a classification report that identifies suspicious hosts based on four different criteria: failure volume, failure entropy, failure persistence, and failure uptick. We implemented our prototype FIA engine by modifying the wireshark network protocol analyzer. Our correlation engine is implemented in Python and uses a publicly available clustering package [1].

3.2 Building an FIA from Wireshark

Wireshark([8]) is an open-source network protocol analyzer that is based on libpcap library. Hence, Wireshark can analyze packets captured from a live network connection or read from a captured pcap trace file. It is distinguished by its flexible design that makes it easy to add dissectors for new protocols and built-in support for hundreds of popular protocols.

We modified Wireshark to automatically extract failure information. The failures we consider include transport-level and selected application-level protocols such as FTP, HTTP, SMTP, DNS, and IRC. For each ICMP failure, we record the error type and client IP address. For TCP failure, we record client and server IP addresses and corresponding port numbers. For DNS failures, we record the

failure type, domain name, and client IP address. For FTP, IRC, HTTP and SMTP failures, we record the server IP address, error code, client IP address, and detailed failure information that may be helpful to an administrator. We also capture the packet associated with each failure message. We focus on these five protocols simply because they were the most popular in the enterprise traffic that we monitored. However, the design of Wireshark makes it straightforward to track failures in other protocols. Finally, as we are interested only in identifying potentially infected local hosts, we configure our system to only track inbound failure messages.

3.3 L7-Based Automatic Protocol Inference

One problem with Wireshark is that it does not have built-in protocol inference capability. It does not detect when a well-known protocol, *e.g.*, HTTP, is used in nonstandard ports. Wireshark expects each dissector to be tied to one or more ports and relies on the user to explicitly decode the packet by choosing a dissector when the packets are observed in unspecified ports. This is a fundamental limitation especially for malware analysis, as malware often transmits packets in nonstandard ports to evade monitoring systems.

To improve the fidelity of the FIA engine, we enhance Wireshark with L7 filter protocol signatures. L7-filter [5] is a classifier that can identify packets based on packet payload. It uses regular expressions to automatically classify packets as belonging to certain common protocols. We provide below examples of L7 protocol signatures for HTTP and IRC:

– **HTTP Protocol:** `http/(0.9|1.0|1.1)[1-5][0-9][0-9][\x09-\x0d-~]*(connection:|content-type:|content-length:|date:)|post[\x09-\x0d-~]*http/[01].[019]`

– **IRC Protocol:** `^(nick[\x09-\x0d]*user[\x09-\x0d]*:|user [\x09\x0d]*:[\x02-\x0d]*nick[\x09-\x0d]*\x0d\x0a)`

We modified the connection struct in Wireshark to maintain a dissector tag for each connection. Every connection starts without any pre-specified dissectors. When a packet arrives, we first check to see if the connection has been allocated to a dissector. If not, we check to see if the packet matches one of the L7 filter signatures. If it finds a suitable dissector, then the connection struct is updated so future packets can be accelerated, bypassing the L7 regular expression check. Once the packet is parsed with the appropriate dissector, the output is examined for any failure messages that are stored in a log file. The FIA engine is installed as a monitor on the span port of the gateway router of the enterprise networks and logs inbound failure responses from remote servers. Figure 2 illustrates the modified Wireshark packet processing engine.

3.4 Multipoint Deployment

We begin with a simplified overview of a domain name lookup using the domain name service. As in our deployment, DNS servers are typically located inside the enterprise network. Local enterprise clients submit name resolution requests to the

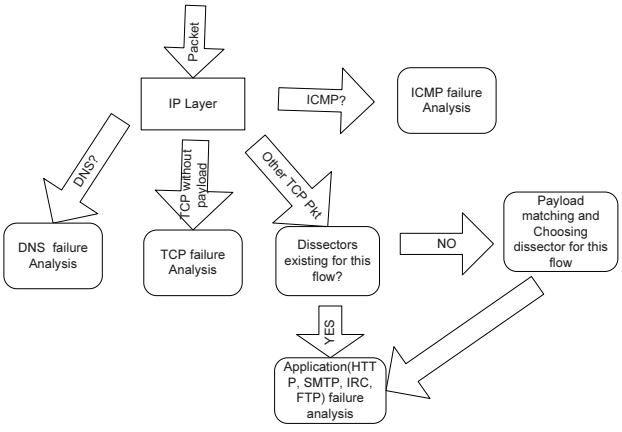

Fig. 2. Modified Wireshark packet processing engine

local DNS server (resolving name server). The resolving server checks its cache and if the name does not exist queries the authoritative name server on behalf of the local client. (The resolving server might have to query additional servers to obtain the name of the authoritative server for a specific domain.) Finally, the resolving name server responds back to the client with the appropriate IP address or NXDOMAIN if the name does not exist, or other type of DNS failure.

A side-effect of the hierarchical DNS system is that it poses additional challenges for any network-based monitoring system as monitoring the gateway only provides a view of the interaction between the resolving name server and external DNS servers. While suspicious domain lookups could be identified, they cannot typically be tracked back to the client that originated the name lookup. Netfuse addresses this problem by integrating an additional lightweight monitor (which we call DNSMon) that tracks activity between the local clients and the resolving name server. DNSMon produces regular alert logs that summarize DNS failure activity of all enterprise hosts. By combining DNSMon alerts with the data collected at the gateway monitor, we get a comprehensive log of network failure activity. Next, we describe how the Netfuse correlation engine processes this information to intelligently isolate suspicious enterprise hosts.

4 Correlation and Clustering Engine

Here, we first describe the algorithm that we implement for ranking suspicious hosts based on failure profiles. Next, we describe our algorithm for classifying groups of hosts with similar failure profiles. Finally, we discuss some techniques that we implemented for reducing false positives in our enterprise network.

Based on our empirical experience from analyzing malware traces, the current prototype system implementation is focused on failures that occur in the transport-layer and five application-layer protocols: HTTP, FTP, SMTP, DNS, and IRC. As Wireshark has dissectors for hundreds of protocols, it is not difficult

to extend the system to support additional protocols. We now describe how our detection algorithm works based on failure input from these protocol analyzers.

4.1 Detecting Suspicious Hosts

The primary inputs to the diagnostic algorithm are failure logs obtained from the FIA engine and DNSMon described in Section 3. First, we classify and aggregate failure information based on host IP address, protocol, and failure type. Next, we compute the following four different scores for each host in the enterprise network with failure activity: (*i*) composite failure, (*ii*) failure divergence, (*iii*) failure persistence and (*iv*) failure entropy. The scores are each normalized to be in the range of 0 and 1. Finally, we use an SVM-based learning technique to classify suspicious hosts. We begin by describing the four scoring functions in greater detail.

Composite Failure Score. This score estimates the severity of the observed failures by each host based on volume. For every host, the failure profile can be represented as a vector$\{N_i\}$, where N_i represents the number of failures of the i_{th} protocol. We proceed as follows to compute the composite failure score for each host.

Step 1: In Section 2, we observed that malware tends to have a large number of failures. So the first step in our analysis is a filtering step that culls hosts with the fewest number of failures. Let α_i, β_i, and γ_i represent the number of application level failures, number of TCP RSTs and number of ICMP failures respectively of host i. Furthermore, let $\mu(\beta)$ and $\sigma(\beta)$ represent the average and standard deviation of TCP failures for a host. Likewise, let $\mu(\gamma)$ and $\sigma(\gamma)$ represent the average and standard deviation of ICMP failures for a host.

Specifically, we consider only hosts that satisfy either of the following three constraints: (1) $\alpha_i > \tau$ (where τ is a constant, set to be 15 for our experiments); (2) $\beta_i > \mu(\beta) + 2 * \sigma(\beta)$ (TCP RST count more than two standard deviations from the mean); or (3) $\gamma_i > \mu(\gamma) + 2 * \sigma(\gamma)$ (ICMP failure count more than two standard deviations from the mean). The final two constraints remove backscatter traffic [33], which artificially inflates the TCP RST and ICMP failure counts for IP addresses in the network.

Step 2: Next, we compute a composite score for each of the remaining hosts as follows: score($host_i$) $= \sum_{i=1}^{n} N_i/T_i$, where T_i is the total number of failures for i_{th} protocol across all hosts.

Step 3: Finally, we sort all the hosts according to the score calculated in the second step. Hosts with higher scores are more suspicious than hosts with lower scores.

Failure Divergence Score. The objective of the failure divergence score is to measure the degree of uptick in a host's failure profile. In particular, we would like to measure the delta between a host's current (daily) failure profile and past failure profiles. We expect that newly infected hosts would show a strong

and positive divergence in their failure patterns while other hosts (clean hosts and those that have been infected for a while) would demonstrate a more stable failure profile.

To quantify this we adopt a well-known statistical forecasting technique, exponentially weighted moving averages (EWMA) [7], that uses a weighted moving average of past observations as the basis for predicting the failure profile for the next day. EWMA uses an exponential distribution to weigh recent observations more heavily than past observations and it is controlled by the parameter α, where α is the smoothing factor, and $0 \leq \alpha \leq 1$. In our measurements, we set α to be 0.5. We compute divergence as follows for each host in the network. Let E_{ijt} correspond to the expected number of failures for host i, on protocol j on day t. We compute E_{ijt} as shown in Figure 3. We then compare the actual value X_{ijt} with E_{ijt} by calculating the distance as follows: 1-(E_{ijt}-X_{ijt})/(E_{ijt}+X_{ijt}). Finally, we normalize by dividing by the maximum divergence score across all hosts in that day to obtain a score in the range [0,1].

$$E_{ij0} = X_{ij0} \tag{1}$$

$$E_{ijt} = \alpha X_{i,j,t-1} + (1-\alpha) E_{i,j,t-1} \tag{2}$$

$$Dist_{it} = \sum_{j=1}^{n} 1 - \frac{E_{ijt} - X_{ijt}}{E_{ijt} + X_{ijt}} \tag{3}$$

$$Divergence_{it} = \frac{Dist_{it}}{\forall k \;\; max(Dist_{kt})} \tag{4}$$

Fig. 3. Simple exponential prediction model and divergence computation

Failure Entropy Score. The failure entropy score measures the degree of diversity in a host's failure profile. This is based on the insight derived from Section 2 that failures in many malware applications tend to have a high degree of redundancy, *e.g.*, failures are often restricted to a few ports or domains such as in a bot that tries to repeatedly contact a C&C server that is currently inactive.

For TCP failures, we track entropy in the server distribution and host distribution of each client receiving TCP RST failures. For every server H_i, we record the number of N_i failures from it. We repeat the same for each server port P_i. For DNS failures, we track entropy in the domain names that are associated with failures. For each domain name D_i appearing in failure response, we record the number N_i. For HTTP, FTP IRC, and SMTP failures, we track entropy in the disribution of various failure types (*e.g.*, HTTP/404) within each protocol and remote servers that issue the errors. For each host H_i and each error type E_i, we calculate the corresponding number N_i. We do not consider ICMP failures in the entropy computation.

For those protocols that have two distribution sets, we calculate the average entropy [2] for each set. We begin by computing weights for each host i and protocol j. Then, for each host i, we compute the significance (s) of protocol j as $s_{ij} = N_{ij} / \sum k = 1^n N_{kj}$ (*i.e.*, number of failures of host i in protocol j divided by the total number of failures in protocol j across all hosts). The weight of

protocol j for host i is simply its normalized significance $w_ij = s_ij \ / \ \sum_{k=1}^{n} s_ik$. The weighting function ensures that for each host, protocols that are responsible for a large portion of its failures will dominate its entropy value. Next, for each host i and protocol j, we calculate the entropy p_{ij}. The failure entropy score for the host is simply the weighted average entropy score, *i.e.*, $\sum_{i=1}^{n} w_i * p_i$.

Failure Persistence Score. The final score is failure persistence, which is motivated by the observation from our case study that malware failures tend to be long-lived. Prior approaches have used autocorrelation techniques to detect long-lived periodic behavior of malware additivity [24]. While we could leverage similar statistical approaches to measure persistent malware activity, we adopt a simpler approach to measure persistence. We simply split the time horizon into N parts (where N is set to 24 in our prototype implementation), and compute the percentage of parts where the failure happens. High failure persistence values provide yet another useful indicator of potential malware infections.

SVM-based Algorithm to Classify Suspicious Hosts. Support vector machines are a recent and well-studied family of supervised learning algorithms used for classification of multidimensional data. Given a training data set, SVMs work by building a hyperplane (or a predictor function) that efficiently seperates positive and negative examples. In our case, we are interested in the maximal margin classifier, *i.e.*, a hyperplane that separates positive and negative examples with maximal distance. In many environments, SVMs have been shown to outperform traditional linear classifiers. Indeed, we had a similar experience in testing different classifiers on our data set. For this system, we use a publicly available tool WEKA [9] to implement our SVM-based classification. The input to the system is a series of four-dimensional vectors where each vector corresponds to the four scores of a individual host. We train the system using a set of malware traces and clean traces for which we have ground truth. The classification problem is identifying the set of suspicious hosts in the network.

4.2 Detecting Failure Groups

After we get the result of suspicious hosts, we want to know whether they are infected by the same malware. For example, we want to know whether they belong to the same botnet. This information can help the network administrator rapdily assess what has happened inside the network. To enable this, we developed a clustering algorithm to detect failure groups which we discuss below. We begin by defining the scoring function that is used for comparing failure profiles.

Scoring Function. According to the description above, each type of failure can be represented as a set of (F_i,N_i), where F_i is the failure property and N_i is the number of failures with this property. Given this representation, we can define the similarity between two hosts as follows. The pseudocode for the algorithm is provided in Algorithm 1. For each protocol, the algorithm compares the number of failures for hosts i and j. The similarity score is incremented by protocol failure count of each host minus the difference between the larger and smaller failure count. It should be apparent that hosts with identical failure profiles would end

```
   Let (F_i,N_i) be the set of one host, and (F_j,N_j) be the set of the other.
   procedure Similarity((F_i,N_i), (F_j,N_j))
   Let sum = 0 be the total number of failures of these two sets ;
   Let sim = 0 be the number of failures that show similarity;
1  foreach (F_ik,N_ik) in set (F_i,N_i) do
2      foreach (F_jl,N_jl) in set (F_j,N_j) do
3          if F_ik = F_jl then
4              sim = sim + (N_ik + N_jl − abs(N_ik − N_jl))
5              sum = sum + N_ik + N_jl
       end
   end
6  Return sim/sum;
```

Algorithm 1. Function to calculate similarity between two failure profiles

up with higher similarity scores. Finally, the similarity score is normalized by dividing by the total number of failures between the two hosts.

Clustering Method. The similarity metric enables us to cluster hosts into distinct groups based on their respective failure profiles. We apply hierarchical clustering based on Peter Kleiwig's publicly available clustering package [1]. The unique aspect of this tool is its flexibility, which lets us choose between seven different clustering algorithms. We chose Ward's minimum variance clustering method, which is widely used for hierarchical clustering. The clustering generates a dendrogram that illustrates similarity among hosts in the network based on their failure profiles. Then instead of fixing a threshold to cut them into clusters, we implement Silhouette Validation Method [37] to find the optimal cut index.

5 Evaluation

To evaluate the performance of Netfuse, we conducted comprehensive tests to measure its detection and false positive rates. The traffic that we use includes five traces shown in Table 5: three malware trace sets and two clean traces from a research institute network, which we refer to as the institute trace. First, we built a model from the training trace. Then to test the classification performance, we use traces from different malware sets and mix them with the institute traces.

Table 5. Training and testing data set

	5-day Institute Trace	**12-day Institute Trace**
Malware Trace I	Training	Testing
Malware Trace II		Testing
Malware Trace III		Testing

1. Malware Trace I: We reuse 24 traces from Table 2 which we combine with clean traces to build the classification model.

2. Malware Trace II: This data set contains five malware families that are not included in the training set (Peacomm, Kraken, Rbot, Mimail and Bifros) and three malware instances represented in the training set. We created a VMware-based virtual machine (VM) environment running eight isolated Windows XP virtual machines, infecting each with a different malware instance. We let these systems run for 10 hours and collected traces of all their network activity. We repeated the experiment three times collecting a total of 24 traces (three per malware). We use this trace to evaluate the classification system and the clustering component.

3. Malware Trace III: This data set contains more than 5,000 malware traces that were obtained from a sandnet. This corpus is particularly attractive because it represents a large and diverse collection of malware. However, a deficiency of sandnet traces is that the malware binaries are often run only for a short period and many of them do not generate any network activity. From this large corpus, we downselected 242 longer running traces based on duration and trace size.

4. Benign Institute Trace: We deployed our system online in the research institute network and continuously ran it for over three weeks. The network is rigorously monitored by NIDSs and has more than one hundred systems (mix of Linux, Macs and Windows PCs). Being a relatively small, well-administered network with a diverse mix of traffic makes it a good candidate for evaluating false positives. We use two traces from this network (a 5-day trace for training and a 12-day trace for testing). In our analysis of clients that generate many failures, we stumbled upon a group of misconfigured Tor nodes that are part of another project. These hosts are grouped together by the clustering engine and classified as benign by the SVM classifier.

5.1 Classification and Detection Results

We will first describe the training process. Then we use the built model to test the performance of our system, including detection rate and false positives.

Training Process. In the training process, we use the SVM algorithm to build a classification model. First, we combine malware trace I with the 5-day institute trace to construct the input data set. Intuitively, a larger training set implies a more accurate model. An example of a rule generated by the SVM algorithm is $-4.266 \times$ (normalized) divergence_score $-0.042 \times$ persistence_score $+0.664 \times$ entropy_score $+0.561 \times$ failure_score $+ 1.8486$. For our evaluation the detection rate for training is 97.2% and the false positive rate is 0.3%.

Performance Evaluation. To measure the detection performance and estimate false positive rates, we mixed different malware traces I, II, and III with 12 days of institute traces. We then processed them through the Netfuse classifier, which took under one hour to process the failure logs for 12 days. In each case, we counted the number of malware traces that were identified (true-positives)

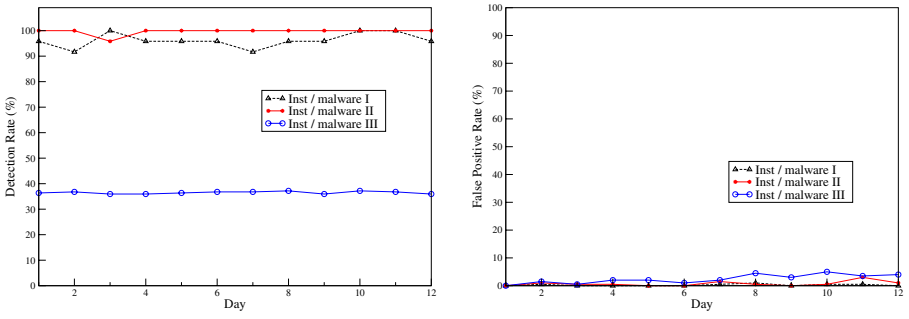

Fig. 4. Detection (left) and False Positive (right) rates on Institute/malware I,II,III mixture traffic

and the number of benign clients that were classified incorrectly. The results are shown in Figure 4. The detection rate is more than 92% for traces I and II. For trace 3, the detection rate varies between 35% and 40%, *i.e.*, around 90/242 malware instances detected. The lower detection rate for trace 3 could be attributed to two reasons. First, the trace set includes many types of malware, including adware that often have traffic profiles similar to benign applications. Second, the traces are quite short (around 15 minutes long). Despite this, Netfuse is able to detect over a third of the malware without any specialized signatures. The false positive rate is consistently lower than 5%.

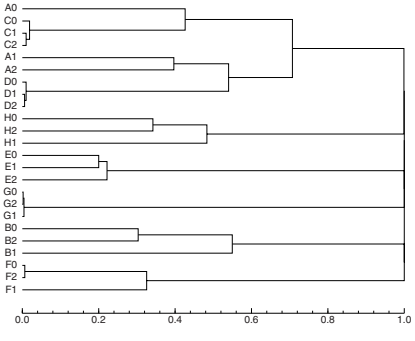

Fig. 5. Malware clustering dendrogram

Table 6. Malware clustering summary

Bot Trace	Packets	Clustered	Accuracy
Peacomm	999905	3/3	100%
Bifrose	30635	3/3	100%
Mimail	279962	3/3	100%
Kraken	49505	3/3	100%
Sdbot	312796	3/3	100%
Spybot	79750	3/3	100%
Rbot	1175083	3/3	100%
Weby	9000	3/3	100%

Clustering Results. After we identify suspicious hosts, we group them according to their failure profiles to simplify analysis of the network administrators. We use malware trace II to test the clustering engine. As shown in Table 6, we find that in all cases the clustering is quite robust. The corresponding dendrogram is provided in Figure 5, where 24 hosts are infected with eight malware instances listed A-H.

6 Related Work

Over the last three years, botnets have become one of the hot areas in networking and security research. In [40] Rajab *et al.* use a multifaceted approach to conduct a comprehensive study on the prevalence of IRC botnets. Dagon *et al.* ([12]) use DNS sink-hole redirection to measure botnet properties and develop a diurnal model for botnet propagation. In [21], Grizzard *et al.* study the structure of botnets and discuss how the single point weakness will force botherders to a P2P structure using the Storm botnet as an example. Vogt [42] *et al.* discuss a recent trend toward smaller botnets and raise the threat of superbots, *i.e.*, an army of distributed botnets that can be coordinated to act as a single network. More recently, Holz *et al.* discuss the emerging threat of fast-flux service networks [25]. Bayer *et al.* [11] propose a scalable algorithm to cluster the malware according to the host behavior profiles.

Inspired by these measurement and modeling studies, there has been a considerable research thrust in building better botnet detection systems. The Rishi [20] system detects IRC botnets by matching IRC bot nickname patterns. BotHunter was the first system to use dialog correlation to detect botnet activity. BotSniffer uses spatio-temporal correlation to detect botnet C&C activity [24]. The BotMiner system [22] combines clustering techniques with heuristics developed by BotHunter and BotSniffer to classify malware based on both malware activity patterns and C&C patterns. The motivation for Netfuse and its correlation approach bears certain similarities to these systems. However, these systems fundamentally differ from Netfuse in that they ignore application-layer failures and focus on successful communication patterns of bots.

Others have developed machine-learning approaches to detect botnets [30, 19]. Bayesian network classifiers are used in [30]. In this paper, authors use machine learning techniques to distinguish between non-IRC traffic, botnet IRC traffic and non-botnet IRC traffic. A different framework, which uses an entropy classifier and a machine-learning classifier, to detect chat bots is provided in [19]. It shows that message sizes and inter-message delays are sufficient to differentiate humans from chat bots. We consider these efforts complementary to our system. Statistical traffic anomaly detection techniques have also been demonstrated to have the potential of identifying botnet-like activity. The exPose system [29] uses statistical rule-mining techniques to extracting significant communication patterns and identify temporally correlated flows, such as worm flows. Threshold random walk is a well-known algorithm that uses hypothesis testing to identify portscanners and Internet worms [28].

Finally, we are also informed by traffic characterization studies such as Pang *et al.* [34] and efforts to automate characterization of enterprise use patterns [15]. A comprehensive analysis of DNS query traffic and its use in identifying network anomalies is provided in [38]. While our system is tuned toward the botnet detection problem, Netfuse could be easily extended to be used as a traffic characterization tool.

7 Conclusion

We propose failure information analysis as a new paradigm for detecting application-layer failures and suspicious activities in the enterprise. We are motivated by the goal of automatically discovering infected hosts in the enterprise. We use an empirical analysis case study to highlight certain differences in bot-like malware and production enterprise traffic that could be exploited to identify infection activity. Using this framework, we develop a prototype system called Netfuse that has three integral components: FIA, DNSMon and the correlation engine. The correlation engine uses four different scores (composite failure, divergence, failure entropy, and failure divergence) to classify suspicious hosts and a clustering component aggregates hosts with similar failure profiles to simplify analysis. We evaluate the system using several malware traces. Our evaluation and analysis shows that Netfuse is an efficient and effective system for discovering embedded malware. In future work, we plan to address the problem of adapting Netfuse to deal with knowledgeable adversaries.

Acknowledgements

This material is based on work supported by the Army Research Office under Cyber-TA Grant No. W911NF-06-1-0316 and by the National Science Foundation Grant No. CNS-0716612. This work was also partially supported by DOD (Air Force of Scientific Research) Young Investigator Award FA9550-07-1-0074 and a grant from NU-Motorola Center for Seamless Communication. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding sources. We also wish to acknowledge help from Michael Hodgsett, Matt Jonkman and the useful feedback received from our Securecomm reviewers.

References

1. Data clustering, http://www.let.rug.nl/~kleiweg/clustering/
2. Entropy, http://en.wikipedia.org/wiki/Information_entropy
3. Gnu wget, http://www.gnu.org/software/wget/
4. Kademlia, http://en.wikipedia.org/wiki/Kademlia
5. L7-filter: Application Layer Packet Classifier for Linux, http://l7-filter.sourceforge.net/
6. Offensive Computing, Community Malicious code research and analysis, http://www.offensivecomputing.net/
7. Simple Exponential Smoothing, http://en.wikipedia.org/wiki/Exponential_smoothing
8. Wireshark: The World's Most Popular Network Protocol Analyzer, http://www.wireshark.org/
9. WEKA-Machine Learning Software in Java (2008), http://weka.wiki.sourceforge.net/Primer-?token=2b7a093d07966047b281eeec0da1b9fd

10. Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F., Nazario, J.: Automated classification and analysis of internet malware. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 178–197. Springer, Heidelberg (2007)
11. Bayer, U., Comparetti, P.M., Hlauscheck, C., Kruegel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: Network and Distributed System Security Symposium, NDSS (2009)
12. Dagon, D., Zou, C., Lee, W.: Modeling botnet propagation using time zones. In: Network and Distributed System Security Symposium, NDSS (2006)
13. Moore, D., Shannon, C., Brown, J.: Code-Red: A case study on the spread and victims of an Internet worm. In: Proceedings of the Internet Measurement Workshop (2002)
14. Debar, H.: An Introduction to Intrusion Detection Systems. In: Proceedings of Connect (2000)
15. Estan, C., Savage, S., Varghese, G.: Automatically Inferring Patterns of Resource Consumption in Network Traffic. In: Proceedings of ACM SIGCOMM (2003)
16. F-Secure. Kapersky Security Bulletin 2008: Malware Evolution January - June 2008 (2008), http://www.viruslist.com/analysis?pubid=204792034
17. F-Secure. Calculating the Size of the Downadup Outbreak (2009), http://www.f-secure.com/weblog/archives/00001584.html
18. Fitzgerald, P.: Downadup: Geolocation, Fingerprinting and Piracy (2009), https://forums.symantec.com/t5/Malicious-Code/Downadup-Geo-location-Fingerprinting-and-Piracy/ba-p/380993
19. Gianvecchio, S., Xie, M., Wu, Z., Wang, H.: Measurement and classification of humans and bots in internet. In: USENIX Security (2008)
20. Goebel, J., Holz, T.: Rishi: Identify bot contaminated hosts by irc nickname evaluation. In: Hot Topics in Understanding Botnets (HotBots) (2007)
21. Grizzard, J.B., Sharma, V., Nunnery, C., Kang, B.B.: Peer-to-peer botnets: Overview and case study. In: Hot Topics in Understanding Botnets (HotBots) (2007)
22. Gu, G., Perdisci, R., Zhang, J., Lee, W.: Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In: Proceedings of the 17th USENIX Security Symposium (2008)
23. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting malware infection through IDS-driven dialog correlation. In: Proceedings of 16th USENIX Security Symposium (2007)
24. Gu, G., Zhang, J., Lee, W.: Botsniffer: Detecting botnet command and control channels in network traffic. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium, NDSS 2008 (2008)
25. Holz, T., Gorecki, C., Rieck, K., Freiling, F.C.: Measuring and detecting fast-flux service networks. In: NDSS (2008)
26. SRI International. Malware Threat Center (2008), http://mtc.sri.org
27. Javitz, H., Valdes, A.: The SRI IDES statistical anomaly detector. In: Proceedings of IEEE Symposium on Research in Security and Privacy (1991)
28. Jung, J., Paxson, V., Berger, A.W., Balakrishnan, H.: Fast portscan detection using sequential hypothesis testing. In: Proceedings of the IEEE Symposium on Security and Privacy (2004)
29. Kandula, S., Chandra, R., Katabi, D.: What's going on? Learning communication rules in edge networks. In: Sigcomm (2008)

30. Livadas, C., Walsh, R., Lapsley, D., Strayer, W.T.: Using machine learning techniques to identify botnet traffic. In: Proc. IEEE LCN Workshop on Network Security, WoNS 2006 (2006)
31. Trend Micro. Trend Micro Threat Roundup and Forecast - 1H 2008 (2008), http://us.trendmicro.com/us/threats/enterprise/security-library/threat-reports/index.html
32. Microsoft. Microsoft Security Bulletin MS08-067 – Critical (2008), http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx
33. Moore, D., Voelker, G.M., Savage, S.: Inferring internet denial-of-service activity. In: Proceedings of the 10th Usenix Security Symposium (2001)
34. Pang, R., Allman, M., Bennett, M., Lee, J., Paxson, V., Tierney, B.: A first look at modern enterprise traffic. In: IMC (2005)
35. Pang, R., Yegneswaran, V., Barford, P., Paxson, V., Peterson, L.: Characteristics of Internet background radiation. In: Proceedings of the 4th ACM SIGCOMM Internet Measurement Conference (2004)
36. Paxson, V.: Bro: A system for detecting network intruders in real-time. In: Proceedings of the 7th USENIX Security Symposium, San Antonio, TX (January 1998)
37. Rousseeuw, P.: Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. Journal of Computational and Applied Mathematics 20 (1987)
38. Plonka, D., Barford, P.: Context-aware clustering of dns query traffic. In: Proceedings of ACM Internet Measurement Conference (2008)
39. Plonka, D., Barford, P.: Context-aware Clustering of DNS Query Traffic. In: Proceedings of the 8th ACM SIGCOMM Internet Measurement Conference (2008)
40. Rajab, M.A., Zarfoss, J., Monrose, F., Terzis, A.: A multifaceted approach to understanding the botnet phenomenon. In: Proceedings of the 6th ACM SIGCOMM Internet Measurement Conference (2006)
41. Roesch, M.: The SNORT Network Intrusion Detection System (2002), http://www.snort.org
42. Vogt, R., Aycock, J., Jacobson Jr., M.J.: Army of botnets. In: Network and Distributed System Security Symposium, NDSS (2008)
43. Yegneswaran, V., Porras, P., Saidi, H., Sharif, M., Narayanan, A.: SRI's Multiperspective Malware Infection Analysis Page (2009), http://www.cyber-ta.org/releases/malware-analysis/public/
44. Zdrnja, B., Brownlee, N., Wessels, D.: Passive Monitoring of DNS Anomalies (2007)