Taint Propagation in a Network

Grigoriy Kerzhner and Kyle Feuz and Neal Sorensen and Chad Mano

Department of Computer Science
Utah State University
Logan, UT 84322, USA
gik@duke.edu,neal.sorensen@usu.edu,kyle.feuz@aggiemail.usu.edu,chad.mano@usu.edu

Abstract. This paper proposes a network taint propagation algorithm and is a combination of software taint propagation and botnet infection. We propose an efficient algorithm to monitor the spread of infection in a local area network. In our approach, whenever an infected machine communicated with an uninfected machine, the new machine was considered infected as well. Initially, one machine was considered tainted by a gateway monitoring system and thus became a bot. Then, as this machine infected others in the local network to create a sub-botnet, our taint propagation algorithm monitored the spread of this infection. This algorithm effectively implemented network taint propagation with a less than 10 percent increase in traffic.

1 Introduction

This paper describes the design and implementation of a taint propagation algorithm in a local area network. Our taint propagation system is a fusion of two well studied topics: botnets and taint propagation. To begin, we will give a brief background on each of these two subject.

Botnet infection is one of the most serious problems threatening the security of the Internet today [1, 11]. In 2004, it was estimated that almost one million systems were part of a botnet; this number is only expected to rise in the future [8]. A botnet is a collection of infected hosts called bots. These bots receive commands and instructions from a leader known as the botmaster. Every bot has two basic functions. Firstly, a bot tries to infect as many other machines as possible. Secondly, a bot carries out malicious instructions from the botmaster. Traditionally, each bot in the botnet received commands from one centralized command and control server [3, 8]. Such a botnet is called a centralized botnet; however, centralized botnets are weak because they can be destroyed by disabling communication with this single command and control server. Because of this weakness, Peer-to-Peer (P2P) botnets have recently emerged as a more sophisticated attack [7].

In software development, tainted data is any data input by a user. If an application prompts input from a user, the user could type specific commands to preform malicious actions. A very simple example of this phenomenon is the System.exec() function in Java. If an application does not watch for tainted data, the user may use this function to run his or her own commands on the

host system [5]. To prevent attacks such as the System.exec() attack, user input must be followed through an application to make sure it does not cause the application to preform malicious or illegal actions. This process is called taint propagation [5].

In this paper, we describe the combination of taint propagation and the event of a botnet infection. Specifically, we worked with a small local network. Traffic was monitored between this local network and the rest of the Internet and eventually, one machine was declared infected. This initial infection is analogous to a user entering possibly dangerous input into a software application. After the initial infection, a machine in the sub-botnet was considered newly infected when an already infected machine communicated with it. Therefore, after a machine in a local network was initially infected, tainted machines communicated with non-tainted machines and the overall list of infected machines grew.

This paper presents an efficient algorithm for taint propagation in a local network in the following manner. Firstly, we discuss related work done about sub-botnets and taint propagation. Secondly, we discuss motivation for and limitations of our work. Thirdly, we present our taint propagation framework and its implementation. Then, we examine the correctness and efficiency of our taint propagation application. Finally, we conclude and propose ideas for future research.

2 Related Work

In this section, we describe related work on sub-botnets and software taint propagation to prepare the reader for the fusion of the two topics in our approach.

2.1 Infection of a Local Network: the Creation of a Sub-Botnet

To date, Bothunter is the best known system that is capable of detecting P2P botnets [4]. However, Bothunter is only a gateway monitoring system and thus may not detect the infection of a local network if there is cooperation between the local machines. This flaw exists because in order for Bothunter to detect an infection, a machine must preform several actions that indicate that the system has been compromised. However, in a local network it is possible to divide these actions between several computers. As a consequence of such a division, machines in the local network still gets infected but the infection is not detected by traditional botnet mitigation protocols. From here on, such an infected local network is referred to as a sub-botnet [9].

A botnet that is able to evade Bothunter and infect a local network is described in detail in [5,9] but we will briefly summarize its behaviour here. Firstly, one computer in the local network must be comprised from an external source. Then, this infected machine attempts to quietly infect other machines in the local switched network. Such a local infection is much harder to detect by an existing infection detection system (IDS) [10]. Furthermore, this local infection will certainly not be detected by gateway monitoring systems such as Bothunter because infections happen inside the sub-botnet and not at the entrance.

Obviously, in order to infect a local network, local machines must communicate with each other. This communication within the sub-botnet makes it

possible for us monitor infection spread throughout the network. The approach of [10]. proposes monitoring switches to detect such communication and thus detect infection in the local network. This project will use this concept of switch monitoring to implement taint propagation in a sub-botnet.

2.2 Software Taint Propagation

While software taint propagation protects an application from many different types of attacks, injection attacks are the most popular and dangerous [6]. A typical injection attack usually follows a framework similar to the System.exec() assault described in the introduction [5]. Applications vulnerable to injection attacks must accept some form of user input. The malicious user attempts to compromise the application by inputting his or her own code instead of regular user input expected by the software. Therefore, if an application is not protected against injection attacks, a malicious user could use such an attack to access confidential information the application stores in a database.

In order to protect a software application against injection attacks, any user input is considered tainted. Then, taint propagation is used to track this tainted input throughout the application. While [5] provides an in depth overview of software taint propagation, we will briefly summarize it here. Any data entered by the user is marked tainted; this is usually done by adding a special header to user input. Furthermore, whenever tainted data is combined with non tainted data, the whole result is considered tainted as well and is therefore also marked with the taint character. Consequently, whenever a function is called within an application, the application checks whether the inputs to the function contain the taint character. In such an event, special precautions are taken to ensure no malicious actions are preformed. This concept of tainting the whole whenever a component is tainted is the core of our network taint propagation approach.

3 Problem Statement and Motivation

Both network taint propagation and sub-botnets have been well documented. However, almost no work has been done on network taint propagation, the combination of these two areas of study. Specifically, we believe we are the first to propose a resource unintensive approach to network taint propagation. This approach is necessary because there is a very clear parallel between malicious use of software and malicious use of a local network. In both cases, infection occurs from an outside source. Furthermore, in both cases this tainted infection spreads throughout the environment and could be used maliciously in several places. Because of these similarities, we decided to create an efficient network taint propagation monitoring system.

There is one main limitation to our system. To avoid false negatives, any communication from an infected machine to an uninfected machine is considered malicious. This can be problematic when an infected machine communicates with another machine for non malicious purposes. In this case, our taint propagation system would declare this other machine tainted when it is not actually infected. Nevertheless, because of fairly rapid detainting and extremely low overhead of our system, we believe this limitation is acceptable.

4 Design

The taint propagation experiment was implemented on a network of four virtual switches and three virtual machines. These machines were infected and became part of a virtual sub-botnet throughout the experiment. Figure 1 is a diagram of this virtual network. The switches listened for an initial message that declared

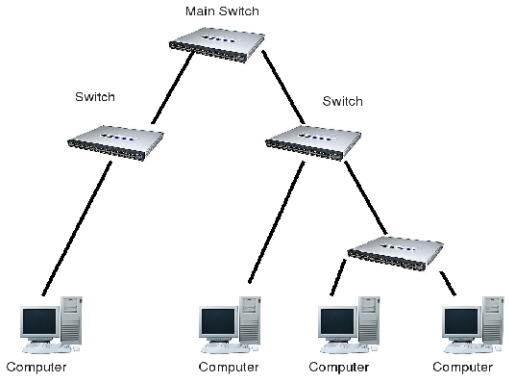

Fig. 1. Virtual network used for the taint propagation experiment

one system in the network malicious. After a machine was declared malicious, the designed software followed the following three steps. Firstly, it detected any traffic from the tainted machine. Secondly, it figured out where the malicious traffic was going. Finally, it declared the destination of this traffic as also malicious. From here, the three step process was repeated until the end of the experiment.

4.1 Behavior of the Virtual Bot

The botnet functioned in the following manner. One computer in the virtual network got the malicious binary from outside the network. This simulates an initial infection that evades Bothunter. Figure 2 is a diagram of this initial infection.

Then, this infected bot communicated with the other machines in the sub-botnet. This secondary infection is pictured in Figure 3. Specifically, the infected bot looked for local machines with a vulnerability. If a vulnerable system did not have a copy of the malicious binary, the infected bot sent this machine a copy of the binary. Once all vulnerable computers had the malicious file, the bots in the sub-botnet communicated with each other and cooperated to preform other malicious activities.

4.2 Scapy Messages: the Key to the Taint Propagation Implementation

Throughout the taint propagation experiment, we considered any communication between a malicious machine and an uninfected machine as malicious. Therefore, when an infected computer communicated with an uninfected computer for any reason, the uninfected computer was then considered infected.

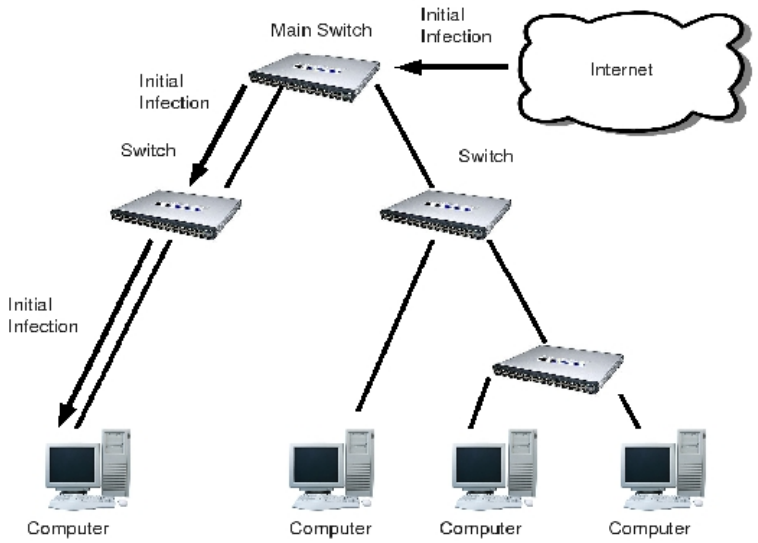

Fig. 2. An initial infection of one machine in the subnet

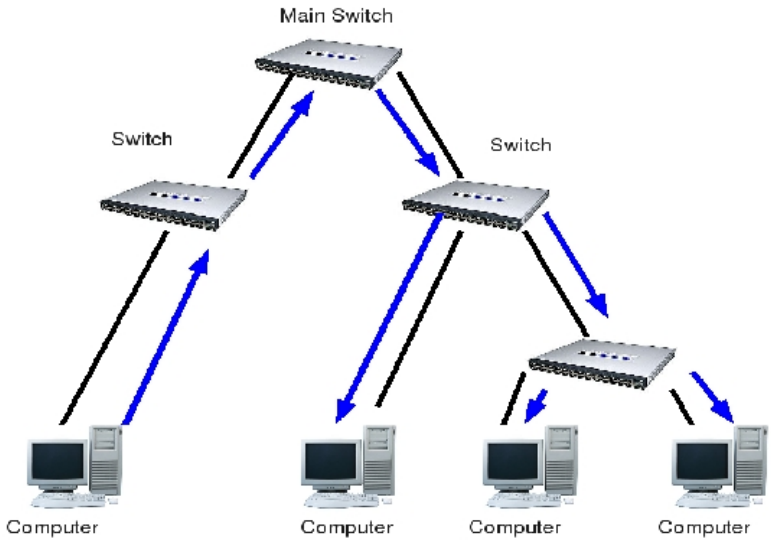

Fig. 3. The initially infected machine propagates its infection to other machines in the virtual network.

Apart from the regular network duties of each switch, the taint propagation system we designed assigned two important new jobs to each switch. Firstly, each switch kept a list of IP addresses of machines declared tainted in the local network. Secondly, the switch sent and received specially crafted packets. The purpose of these packets was to taint and detaint machines in the network.

The tool used to send and receive these packets was Scapy, a python module that makes it possible to send, receive and craft network packets. These packets were sent over the local Ethernet layer and simulated simple IPFIX messages that network switches use to communicate. Scapy packets instructed a switch to either add or remove an IP address from its list of malicious computers and were handled in the following way. Each switch listened for the Scapy messages on a specific port, 4329 in our experiment. A Scapy message had two important fields: the payload field and the header field. The payload consisted of an IP address to be added to or removed from the switch's malicious list. The header specified whether the purpose of the message was to taint or detaint. When a switch sniffed a Scapy message on the specific port, it first read the header to identify what the type of message was. Then, the switch either added or removed the IP address in the payload of the Scapy message according to the instruction in the header.

4.3 Taint propagation algorithm

Each switch was equipped with Python code to handle taint propagation tasks and the structure of this code is described in this subsection. Figure 4 is a flow chart showing the algorithm described below.

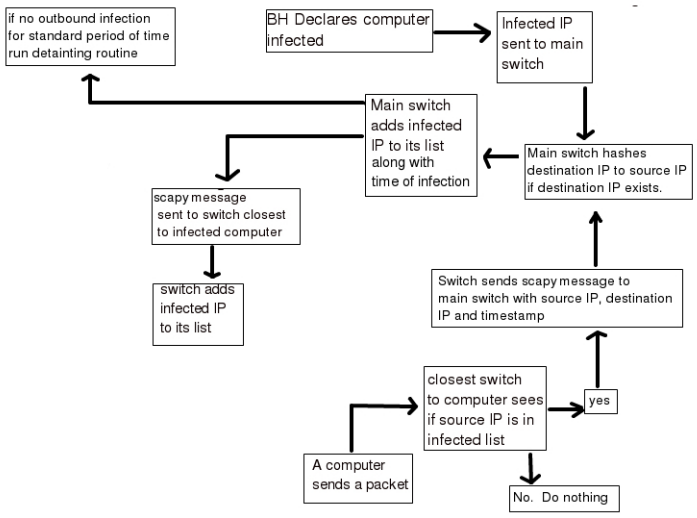

Fig. 4. Our taint propagation algorithm.

One main switch, with its code slightly changed to serve a leading duty, and three subordinate and intermediate switches listened for network traffic. The experiment started when one machine in the virtual network was declared malicious. This event simulated Bothunter detecting an infected machine in the local network. The main switch added the IP address of this newly infected machine to its malicious list. Then, the main switch sent a Scapy message to the newly infected machine with that machine's IP in its payload. Each switch on the way to the newly infected machine recorded the IP address from the payload of the Scapy packet.

Each switch also monitored regular network flow. A new machine was declared malicious in the following four step process. Firstly, when a switch sniffed a packet, it checked to see whether the source address of the packet is malicious. If that was the case, then the destination IP of the packet was declared as malicious as well. Secondly, since the main switch kept a record of all malicious IP addresses, it was notified about this newly infected machine. The switch that declared the machine malicious preformed this notification by sending a Scapy message to the main switch with the IP address of the newly infected machine in the payload. Thirdly, the main switch added this new address to its malicious list. Lastly, the main switch sent a Scapy message to the newly declared malicious machine with this newly declared IP address in the payload. Once again, every switch on the way to the machine intercepted the Scapy message and added the payload to its malicious list.

If this taint propagation algorithm was implemented with no additional code, then eventually most computers in a network would be declared malicious. To prevent this, the experiment was run again with a detainting algorithm. The main switch kept track of what time the malicious IP was added to its malicious list. If this IP had been in the list for longer than the timeout period (thirty seconds in our inital experiments), it was removed from the malicious list of the main switch. However, since other switches in the virtual network also had this IP declared as malicious, they had to be notified of the detainting as well. This was done through a Scapy message with a detaint header. This message was sent to the IP address of the machine to be detainted. Every switch that sniffed the message deleted the IP address in the payload from its malicious list.

5 Results

Taint propagation was run on the switches of the virtual network while the virtual network was infected and turned into a sub-botnet. To begin our testing, the experiment was run twice, once with detainting and once without detainting. With detainting, a computer was considered untainted 30 seconds after tainting. Two factors were monitored throughout this initial experiment: correctness and efficiency. After our algorithm was determined to be correct and fairly efficient, we experimented with different detaint times. Specifically, we looked at how the efficiency our taint propagation system changed when the detaint period was varied.

5.1 Correctness

Since each of the virtual computers in the experiment had a vulnerability, we expected them to all be considered tainted by the end of the experiment. This

turned out to be true. The taint propagation algorithm was run for 90 seconds while the virtual network was infected. When detainting was not enabled several machines were considered infected. All three virtual machines were correctly counted as tainted. Furthermore, four other addresses were tainted. These extra machines were tainted due to machines in the sub-botnet communicating with machines outside of the local network. When detainting was enabled, fewer machines were considered infected. However, since a large amount of traffic was processed during the 90 seconds, all four virtual computers were still considered tainted at the end of the experiment.

5.2 Efficiency

We defined the efficiency of our system as the number Scapy packets generated by the taint propagation system divided by total traffic in the system. Specifically, this statistic is the number of Scapy packets received by the main switch divided by the total number of packets received by the main switch. From here on, this measure will be referred to as the overhead of the taint propagation system.

All traffic was monitored by the main switch to figure out what overhead the detainting algorithm caused on the system. Figure 5 plots total traffic versus Scapy messages that implement taint propagation. According to Figure 5, the

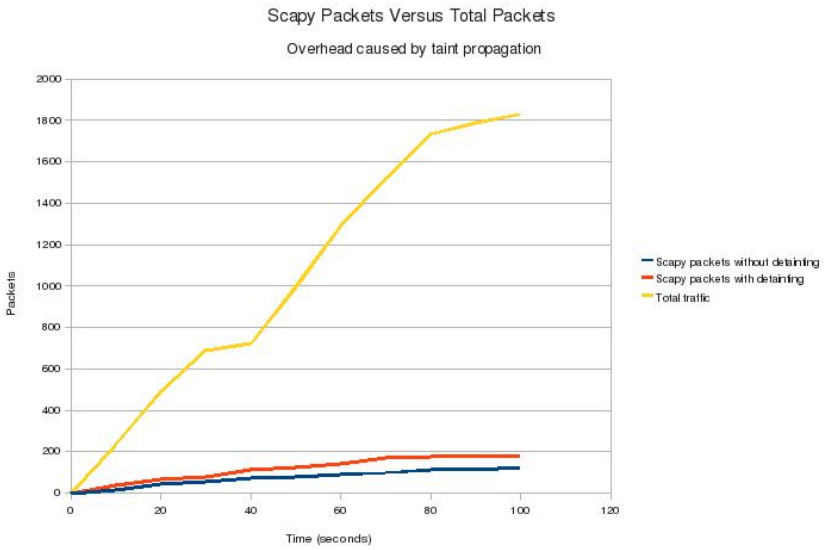

Fig. 5. Total network flow verus overhead created by taint propagation system

overhead caused by this taint propagation system was less than 10 percent. Specifically, with detainting enabled our taint propagation system caused a nine percent increase in traffic. Without detainting, taint monitoring only increased traffic by about six percent. Figure 6 presents the overhead created by our taint

propagation system. According to Figure 6, the taint propagation monitoring system was efficient throughout the experiment.

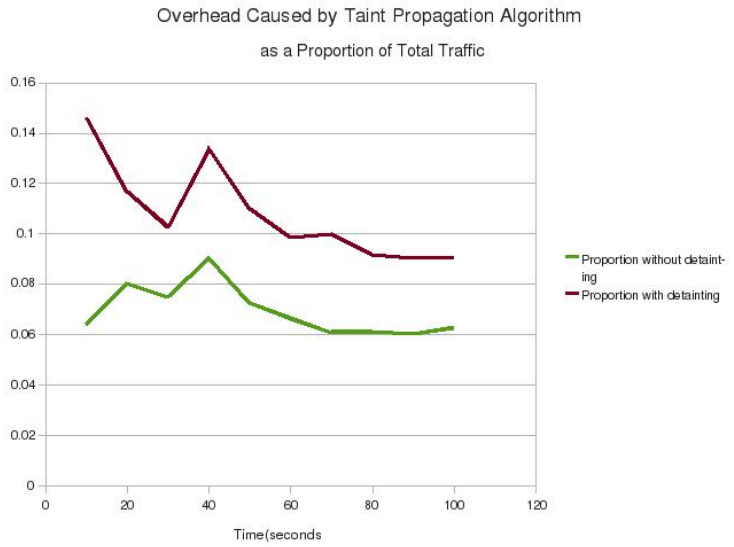

Fig. 6. Overhead caused by taint propagation system as a proportion of total network traffic

Even though our system was decently efficient in this virtual environment, we believe it would be much more efficient in the wild. This is true because in our virtual network, a large proportion of traffic was created by the botnet. Therefore, in our experiment the number of infection messages was much larger than in a regular network. While this disparity was good for testing the efficiency and correctness of the system, the disparity caused a higher than normal overhead. In a real network, one would expect many more messages that do not cause any tainting and detainting and therefore a much lower overhead.

5.3 Effect of the detaint period on the efficiency of our taint propagation algorithm

In our final experiment, we varied the time before a tainted system was considered detainted. Four different detaint periods were used: 15, 30, 45 and 60 seconds. Overhead was measured with each detaint period and the results of these measurements are shown in Figure 7. Because a longer detaint period meant fewer Scapy detaint messages sent between switches, lengthening the detaint period significantly reduced the overhead of our taint propagation algorithm.

6 Summary and Future work

In conclusion, this paper presents a unique and efficient approach to taint propagation in a sub botnet. Switch monitoring only cased a five to ten percent increase in total traffic and therefore was not extremely costly to system performance.

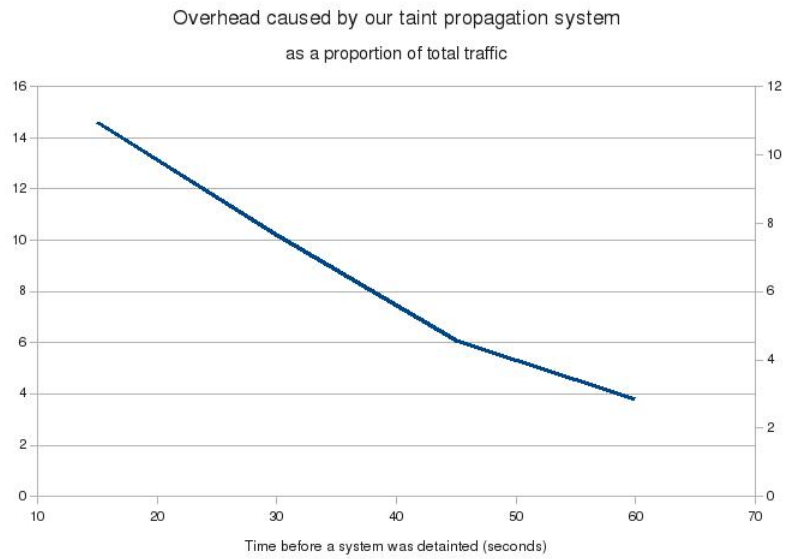

Fig. 7. Overhead of the taint propagation system versus time before detainting

A few issues arose throughout our work. Firstly, in order to implement our framework, switches must be able to run the python code at the core of our taint propagation system. Therefore, the network must contain smart switches or switches that have the taint propagation system already built in. Secondly, a major difficulty in creating a more efficient taint propagation system in a switched local network is network topology discovery. Although many algorithms have been proposed to implement network topology discovery, this process still proves difficult to implement efficiently [2]. This problem rises because because modern switches are designed to be invisible in the network. For this reason, it is difficult and costly to figure out exactly which switches are connected to which machines.

If the topology of the network was known, taint propagation could be carried out more efficiently. Perhaps the most important part of our taint propagation algorithm is intercepting communication from an already tainted computer to another machine. Without knowledge of the network topology, several switches must watch for communication from the tainted machine. This is because it is not clear exactly through which switches the tainted computer will try to communicate. If the switched network layout was known, on the other hand, only the switch closest to the tainted computer would have to be notified. This would lead to a great reduction in the redundancy of our algorithm.

References

1. Evan Cooke, Farnam Jahanian, and Danny Mcpherson. The zombie roundup: Understanding, detecting, and disrupting botnets. pages 39–44, June 2005.

2. B Donnet, T. Friedman, and M. Crovella. Improved algorithms for network topology discovery. In *Proc. Passive and Active Measurement Workshop (PAM)*, Boston, MA, USA, Mar. 2005. See also the traceroute@home project: http://trhome.sourceforge.net.
3. Julian B. Grizzard, Vikram Sharma, Chris Nunnery, Brent B. Kang, and David Dagon. Peer-to-peer botnets: overview and case study. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, page 1, Berkeley, CA, USA, 2007. USENIX Association.
4. Guofei Gu, ¡i¿georgia, Vinod Yegneswaran, and Martin. Bothunter: Detecting malware infection through ids-driven dialog correlation. pages 167–182.
5. Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.
6. Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, and David Evans. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.
7. Phillip Porras, Hassen Sadi, Vinod Yegneswaran, Phillip Porras, Hassen Sadi, and Vinod Yegneswaran. A multi-perspective analysis of the storm (peacomm) worm. available at: http://www.cyber-ta.org/pubs/stormworm/report, 2007.
8. Moheeb A. Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 41–52, New York, NY, USA, 2006. ACM Press.
9. Brandon Shirley and Chad D. Mano. A model for covert botnet communication in a private subnet. In Amitabha Das, Hung Keng Pung, Francis Bu-Sung Lee, and Lawrence Wai-Choong Wong, editors, *Networking*, volume 4982 of *Lecture Notes in Computer Science*, pages 624–632. Springer, 2008.
10. Brandon Shirley and Chad D. Mano. Sub-botnet coordination using tokens in a switched network. In *GLOBECOM*, pages 2169–2173. IEEE, 2008.
11. W. T. Strayer, R. Walsh, C. Livadas, and D. Lapsley. Detecting botnets with tight command and control. In *Proceedings of the 31st IEEE Conference on Local Computer Networks*, pages 195–202, 2006.