Implementing and testing a virus throttle[1]

Jamie Twycross, Matthew M. Williamson
Trusted Systems Laboratory
HP Laboratories Bristol

HPL-2003-103
May 21st , 2003*

E-mail: jamie@milieu3.net, matthew.williamson@hp.com

In this paper we build on previous theoretical work and describe the implementation and testing of a *virus throttle* - a program, based on a new approach, that is able to substantially reduce the spread of and hence damage caused by mobile code such as worms and viruses. Our approach is different from current, signature-based anti-virus paradigms in that it identifies potential viruses based on their network behaviour and, instead of preventing such programs from entering a system, seeks to prevent them from leaving. The results presented here show that such an approach is effective in stopping the spread of a real worm, W32/Nimda-D, in under a second, as well as several different configurations of a testworm.

* Internal Accession Date Only Approved for External Publication

[1] Proceedings 12th USENIX Security Symposium, 4-8th August 2003, Washington, DC, USA

© Copyright Hewlett-Packard Company 2003

Implementing and testing a virus throttle

Jamie Twycross
jamie@milieu3.net

Matthew M. Williamson
matthew.williamson@hp.com

Hewlett-Packard Labs, Bristol, U.K.

May 13, 2003

Abstract

In this paper we build on previous theoretical work and describe the implementation and testing of a *virus throttle* - a program, based on a new approach, that is able to substantially reduce the spread of and hence damage caused by mobile code such as worms and viruses. Our approach is different from current, signature-based anti-virus paradigms in that it identifies potential viruses based on their network behaviour and, instead of preventing such programs from entering a system, seeks to prevent them from leaving. The results presented here show that such an approach is effective in stopping the spread of a real worm, W32/Nimda-D, in under a second, as well as several different configurations of a test worm.

1 Introduction

CNET dubbed 2001 the "*Year of the Worm*" [14] and 2002 has only seemed to confirm this, with high-speed mobile code forming the majority of entries in the top ten of most infectious programs. Indeed, on the weekend this paper was being completed, the W32/SQLSlam-A worm [22] infected what is currently estimated at 75,000 machines in 30 minutes [16] and caused major network disruption. Finding effective ways to prevent such activity is a high priority.

The work we present here provides just that - an extremely effective way of halting the propagation of certain classes of worms. Our approach, *virus throttling* [27], [30], is based on the observation that under normal activity a machine will make fairly few outgoing connections to new machines, but instead is more likely to regularly connect to the same set of machines. This is in contrast to the fundamental behaviour of a rapidly spreading worm, which will attempt many outgoing connections to new machines. The idea behind the virus throttle is to put a rate limit on connections to new machines such that normal traffic remains unaffected but suspect traffic is slowed, quickly detected and stopped.

Our approach is different from current approaches in *three* key ways: it focuses on the *network behaviour* of the virus and prevents certain types of behaviour, in our case the attempted creation of a large number of outgoing connections per second. It is also unique in that, instead of stopping mobile code from entering a system, it restricts the code from *leaving*. Lastly, because connections over the allowed rate are *delayed* and not dropped, the system is tolerant to false positives and is therefore robust.

In this paper we give a detailed description of an implementation of the virus throttle and our experimental setup, and present the results of a number of tests. These tests show that the virus throttle is able to very quickly detect and prevent worms spreading from an infected machine. For example, the throttle is able to stop the W32/Nimda-D worm [21] in under *one* second. Furthermore, since the throttle prevents subsequent infection, the effect on the global spread of the virus will depend on how widely it is deployed. Our results also show that when 75% of machines are installed with the throttle, the global spread of both real and constructed worms is substantially reduced. Throttled machines do not contribute any network traffic in spite of being infected, significantly reducing the amount of network traffic produced by a virus.

The next section, Section 2, paints the background against which our work stands, briefly reviewing what mobile code is and how it propagates and discussing current approaches to limiting this propagation. It then goes on to outline the foundations on which our perspective is based, and briefly reviews related work. Section 3 describes in detail the design and implementation of our virus throttle, the performance of which, along with our experimental setup, we describe in Section 4. Concluding remarks are made in Section 5.

2 Background

In this section we offer a brief review of mobile code and current approaches to limiting its spread. We then go on to introduce the conceptual framework upon which our work rests, and end the section with a summary of related work.

2.1 Mobile code

We are interested in a class of software broadly known as *mobile code* [10]. For our purposes we define mobile code pragmatically as any program that is able to transfer itself from system to system with little or no human intervention. Many examples of such mobile code can be found in real life, the most common of which are the many viruses and worms that are becoming an increasingly prevalent feature of the Internet [5, 14]. While mobile code can propagate through different media, for example, removable storage, since we are particularly interested in propagation across *networks*, we will restrict our discussion to code that spreads across this medium. Although technical difference do exist between virus and worms [6, 7], in what follows we will use these terms and mobile code interchangeable.

An archetypal piece of mobile code can in general be seen as repeating a cycle composed of several distinct stages. The code will perform some form of *scan* to attempt to locate target machines which are vulnerable to infection, and will then attempt to *exploit* any target machines found. If successful, the exploit will allow the mobile code to *transfer* a copy of itself to the target machine, which will itself begin its own scan/exploit/transfer cycle.

2.2 Current approaches

Current approaches to virus protection involve preventing a virus from entering a system, predominantly through signature-based detection methods [8]. These methods concentrate on the physical characteristics of the virus i.e. its program code, and use parts of this code to create a unique signature of the virus. Programs entering the system are compared against this signature and discarded if they match. In terms of the three-stage scan/exploit/transfer cycle described above, current approaches can be seen as focusing on the *transfer* stage.

While this approach has up to now been fairly effective in protecting systems it has several limitations which, as the number of virus samples increases, decrease its effectiveness. It is fundamentally a reactive and case-by-case approach in that a new signature needs to be developed for each new virus or variant as it appears. Signature development is usually performed by skilled humans who are only able to produce a certain number of signatures in a given time. As the number of viruses increases, the time between initial detection and release of a signature increases, allowing a virus to spread further in the interim. Furthermore, comtemporary viruses are using techniques such as polymorphism and memory-residency to sidestep signature detection entirely.

2.3 Agents and complex systems

An alternative and fruitful approach can be gained by viewing the mobile code as an autonomous agent acting within a complex system [19]. Such a paradigm shift leads to an emphasis on different concepts and also allows a vast amount of literature on complex and adaptive systems from fields concerned with these entities to be drawn upon. For example, when viewed as an agent the question of how the agent behaves within the environment it inhabits becomes as important as the purely mechanistic details of its construction on which the current approach described in the last section is based.

The distinction between mechanism and behaviour can lead to some simple but potentially powerful conclusions. While a virus is able to instantiate an effective spreading mechanism in an extremely large number of ways, each requiring a separate signature, the number of ways in which a virus can behave to spread effectively is perhaps much more limited. This is especially the case with the class of high-speed worms which are becoming increasingly prevalent and which, due to their high-speed nature, need to scan a large number of hosts per second. Behaviour is a much more powerful discriminator than that employed in current, mechanistically-orientated signature-based methods, as it potentially allows the automatic identification and hence removal of an entire class of worms.

Considering behaviour also leads to some less obvious insights, one being that it could be more productive to focus on preventing viruses leaving a system, as opposed to stopping them entering, the strategy taken by current methods. While such a seemingly altruistic approach may at first sight appear ineffective, recent work by Williamson and Leveille [29] and other work discussed in the next section indicates that it can be extremely effectual in preventing the spread of viruses across networks.

2.4 Related work

Our approach is related to the "*behaviour blocking*" of Messmer [15] which seeks to specify policies defining normal or acceptable behaviour for applications. If an application breaches such a policy it is reported to an administrator. The approach we take differs in that it is able to automatically respond to abnormal behaviour, taking the administrator out of the loop in this respect. The benign facet of this response is particularly important as it makes the throttle more tolerant to false positives. Another example of a benign response used in an intrusion detection application is given by Somayaji and Forrest [20], although their application implements this response in relation to abnormal sequences of syscalls. Bruschi and Rosti [2] discuss various ways in which hosts can be prevented from participating in network attacks and describe a tool, AngeL [3], which can be used to prevent systems from participating in such attacks. AngeL, however, relies on a signature-based algorithm to detect attacks, inspecting network packets for predefined sequences of data, for example, shellcode or unusual HTTP requests, in contrast to our behaviour-based approach.

3 The virus throttle

After outlining the context of our work in the last section, we now go on to give a detailed explanation of the design and implementation of a virus throttle, of which an initial description and proof-of-concept based on theory and simulation was presented in [27]. The main focus of this section will be on a TCP implemention of the virus throttle, although we have also tested UDP, SMTP and Exchange throttles with similar designs.

3.1 Design

The virus throttle is a program that limits the rate of outgoing connections to new machines that a host is able to make in a given time interval. For the purposes of simplicity in this section we will assume that the host has *one* unique address - its source IP address, although the implementation described below allows for multiple source IP addresses. Connections to a remote machine are established through what is known as a three-way-handshake in which the initiator of the connection, the source machine, sends a TCP SYN packet to the target machine, identified by a destination IP address. The target machine then sends back a SYN-ACK packet, which the source machine replies to with an ACK packet [17]. By controlling the number of SYN packets transmitted from the source machine we can control the number of connections it is able to make.

A note should be made about the relationship between connection attempts and SYN packets. At the application layer, a connection is usually initiated by opening a socket [23]. This results in the sending of an initial TCP SYN packet and, if no response is received within a certain time, the sending of further, identical, SYN packets. This continues up to a maximum time, the socket timeout, when the socket will give up and return control to the application. Thus, in attempting to open a single connection a machine may actually transmit several SYN packets. In our implementation we make no attempt to differentiate initial SYN packets from retries, and count the retries as separate connection attempts.

A machine will establish many such connections in the course of normal usage, for example when requesting a web page or the delivery of email. Many worms also use such connections when scanning in order to establish the existence and configuration of remote machines, with high-speed worms such as W32/Nimda-D [21] or CodeRed [4] initiating large numbers of connections to different targets per second. The virus throttle rests on the observation that the patterns of connections due to normal usage are very different from the patterns of connections created by such mobile code. Our research has suggested that under normal usage often no more than *one* connection to a target not recently connected to is made per second, and that the majority of connections are made to destination addresses that have recently been connected to [27].

The virus throttle parses all outgoing packets from a machine for TCP SYN packets. The destination address of an intercepted SYN packet is then compared against a list of destination addresses of machines to which connections have previously been made, which we term the *working set*. The working set can hold up to 5 such addresses. If the destination address is in this working set the connection is allowed immediately. If the addresses is not in the working set and the working set is not full i.e. it holds less than 5 addresses, the destination address is added to the working set and the connection is once again allowed to proceed immediately. If none of these two conditions are met, the SYN packet is added to what we term the *delay queue* and is not transmitted immediately.

Once every second the delay queue is processed and the SYN packet at its head and any other SYN packets with the same destination address are popped and sent, allowing the establishment of the requested connection. The destination address of this packet is also added to the working set, the oldest member of which is discarded if the working set is full. If the delay queue is empty at processing time and the working set is full, the oldest member of working set is also discarded, allowing for the potential establishment of one connection per second to a target not recently connected to.

This design, summarised schematically in Figure 1, allows hosts to create as many connections per second as they want to the 5 most recently connected-to machines. Any further connection attempts will be delayed for at least a second, and then attempted. Delaying connections rather than simply dropping them is important as such a benign response [20] allows the virus throttle a certain amount of leeway in its conception of normal behaviour and a response that, if incorrectly targeted at legitimate connection attempts, will introduce an often imperceptible delay in the connection, instead of prohibiting it entirely.

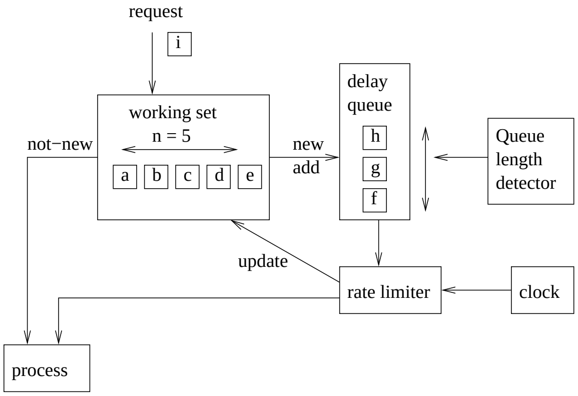

Figure 1: A schema of the control flow of the virus throttle

In [27], it was argued that if the rate of connections to new targets is high, as is the case with many worms, the delay queue would rapidly grow resulting in attempted connections being severely delayed. When implementing the throttle, we took this idea further and introduced an upper limit to the size of the delay queue which, once reached, will disallow all further connection requests by the host. Thus the throttle can be said to behave benignly within certain limits. From observation of the normal behaviour of a range of users we saw that the delay queue rarely grew bigger than a handful of packets and concluded that the size of the delay queue could offer an indication of the presence of a worm on the host: large delay queue sizes would almost certainly indicate an application behaving in a suspect manner. In our throttle we set this upper limit on the delay queue size to 100 packets.

Lastly, as mentioned previously, we have also implemented UDP, SMTP and Exchange versions of the throttle. The UDP throttle works in exactly the same way as its TCP counterpart except that instead of parsing outgoing network traffic for TCP SYN packets, it looks for outgoing UDP packets, each of which is considered a separate connection attempt. The email throttle is described fully in [28].

3.2 Implementation

In this section we describe how the virus throttle was implemented, beginning with a short overview of the structure of the Linux network stack. Necessarily, this section is fairly technical, and the reader is referred to more complete works such as [1, 18] for further information.

The Linux network stack strictly consists of two data structures, the ptype_all linked list and the ptype_base hash table, containing pointers to packet handler functions [13], but, at a conceptual level, can simply be thought of as a list of packet handler functions. Outgoing packets are placed onto the list by a packet handler and traverse the list until they reach its head, after which the packets are passed to the hard_start_xmit function supplied by the appropriate network device driver for transmission on the wire. The virus throttle works by replacing the pointer to the hard_start_xmit function registered by the network device driver with a pointer to a function of its own. This means that, since the hard_start_xmit function is called every time a packet has finished traversing the network stack and is ready to be transmitted on the network interface, we are able to intercept and control every packet leaving the network device. In essence, the throttle can be viewed at this level as an ethernet device driver wrapper.

To accomplish this wrapping the virus throttle is implemented as a Linux 2.4.18 kernel module which, in its init_module routine, fetches the pointer to the net_device structure registered by the current network device driver. The net_device structure contains entries to pointers to the packet handling functions registered by the network device driver, including the hard_start_xmit function, whose pointer we replace with a pointer to our own transmit function.

Our own transmit function is a simple packet parser which looks for TCP SYN packets, the packets used to establish stream-based connections between applications. If the packet being parsed is *not* a TCP SYN packet, it is passed on to the original hard_start_xmit function for transmission as usual. If, however, it is a TCP SYN packet it is, as described in the previous section, either allowed to pass immediately, in which case it is handed off to the original hard_start_xmit function and possibly added to the working set or, if the working set is full, is added to the delay queue for later transmission. Both the working set and delay queue data structures are instantiated as linked lists using the linked list structure provided by the Linux kernel. The working set list stores the destination address of the packet, while the delay queue list stores the source and destination addresses of the packet, a copy of the sk_buff data structure associated with the packet, and the time it was enqueued. The limit on the size of the delay queue is implemented by monitoring the size of the delay queue in the packet parser and setting a flag if this size increases beyond the specified upper limit. The packet parser will never allow a connection attempt to commence if this flag is set.

Processing of the delay queue is handled by a kernel thread which wakes up a specified number of times per second, in our case once per second. The delay queue is then processed as described in the previous section, with packets deemed suitable for transmission dequeued and passed on to the original hard_start_xmit function. Due to the fact that the delay queue and working set are data structures shared by the enqueueing packet parsing routines and the dequeueing delay queue routines, these two data structures are carefully protected by spinlocks.

In order to allow for the fact that one host may have several different IP addresses or that the virus throttle may be placed on some intermediate system such as a bridge or gateway and hence see packets transmitted with a number of different source IP addresses, our implementation actually keeps an array of working set and delay queue data structures. Each entry in this array corresponds to a working set and delay queue for a particular source IP address.

4 Testing

Now that we have detailed the design and implementation of the throttle, in this section we move on to describe how we evaluated its performance. Initially, we outline our experimental setup and then go on to present the experiments we performed and their results.

4.1 Experimental setup

In order to effectively test our throttle in a range of different scenarios we first had to develop a secure testbed on which the virus throttle could be exposed to real and constructed mobile code. It is this testbed that we now describe, more details of which can be found in [25].

4.1.1 Testbed

The testbed consists of a rack mounted single-chassis HP Blade Server bh7800 [12] providing the physical infrastructure for 3 separate LANs and housing what includes three 25-port 10/100 switches and 16 bh1100 Server Blades. Each Blade has a 700MHz Pentium III processor with a 30GB hard-disk, onboard graphics and 3 network interfaces. One network interface is provided to the management LAN by a remote management card (RMC) daughterboard, while the other two provide connections to the remaining two LANs.

As well as being physically separate, the functional roles of the three LANs have also been separated. The *management LAN* provides access to the RMC daughterboard on each Server Blade which can be used for tasks such as power-cycling the Blade. The second LAN is designated the *administrative LAN* and is used to handle the installation and configuration of the Blades, and the coordination of experiments and collation of data from these experiments. The third LAN, the *experimental LAN.* is the network on which the experiments are actually performed. This setup is summarised in Figure 2.

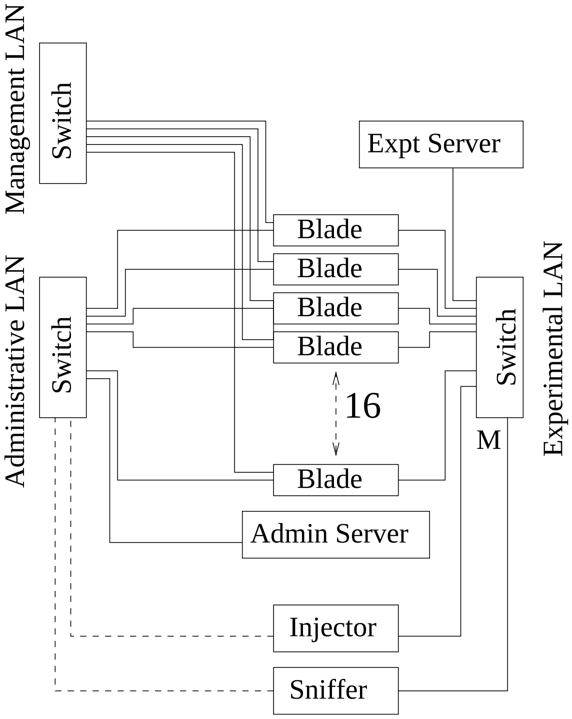

Figure 2: The configuration of the testbed

In addition to the Blade Server, we are also employing four 1.8GHz Pentium 4 boxes, two of which act as servers on the administrative and experimental LANs respectively. The administrative LAN server, as well as running services such as PXE, DHCP and FTP necessary for the configuration of the Server Blades also coordinates our experiments. A third machine acts as our data collector and is in essence a network sniffer. A monitoring port [11] has been configured on the switch on the experimental LAN and the sniffer machine, running tcpdump [24], listens on this port and keeps a copy of all traffic on this LAN. The fourth machine is our infector which, at the start of some of our experiments, we bring up and use to inject a copy of the virus under study into one of the machines on the experimental LAN. This machine is then brought down and plays no further role in the experiments. For coordination purposes, the sniffer and infector machines have network connections to both the experimental and administrative LANs, and particular care has been taken to ensure that no traffic from one of these LANs is able to cross to the other LAN.

A slimmed-down Redhat 7.3 Linux installation with a custom 2.4.18 kernel forms the operating system on the Server Blades. As the experiments involve a variety of operating systems and settings, to ease configuration and save time we run VMware Workstation 3.2 [26] on each Server Blade. VMware allows the running of one operating system, the guest operating system, inside another operating system, the host operating system, in our case Linux. For the experiments below we use a standard installation of Windows 2000 Professional as the guest OS with VMware configured in bridged-only networking mode. Setting a VMware guest OS to use bridged-only networking means that the guest will largely use its own network stack, as opposed to that of the host OS if host-only networking is configured. This results in more realism in the network behaviour of applications running under the guest

OS but, since bridged-only networking employs a proprietary VMware driver located fairly low down in the Linux network stack, places constraints on the implementation of the virus throttle not necessarily present in a production version of the throttle.

This configuration of the testbed allows us to automate the process of running experiments to a large extent, essential when it is necessary to repeat experiments a number of times to reduce the noise inherent in the random nature of the programs we are studying. Obviously, when a network has machines known to be infected with viruses, the security and isolation of this network is of prime importance. As well as the measures described above and careful configuration and testing of all software, the testbed is housed in a secure laboratory physically preventing connections to all other networks and strict policies concerning data transfer are imposed.

4.1.2 Test worm

In order to test the effectiveness of the virus throttle at allaying the spread of worms which scan at different rates a test worm was developed. The test worm consists of a basic stream-socket server [23] which listens for connections on a specified port. When it receives a connection attempt the server starts a scanner whose properties, such as scan rate and address range, can also be specified. This scanner then scans the IP address space by attempting to make connections to addresses on the port the test worm server is listening on. As the type of scanning used is TCP connect scanning [9], if the scanner discovers the IP address of a machine running a test worm server it will trigger the server to start the scanner on this machine. This scheme allows a fairly realistic simulation of the scan/exploit/transfer worm lifecycle but, since no actual exploit or file transfer is involved and since any machines infected by the worm have, *a priori*, to already have been infected with the worm, guarantees that the test worm will never spread autonomously.

4.2 Results

Using the testbed setup described above we have been able to securely observe a variety of different viruses spreading across the experimental network and have performed a number of different experiments with the virus throttle, which we now go on to detail.

4.2.1 Stopping speed

We were first interested in the time it would take worms scanning at a number of different rates to cause the delay queue to reach its upper limit, 100 packets in our implementation. Remembering that once the delay queue has reached this limit we disable all further connection attempts, this time is effectively the time it takes the throttle to stop a worm. The experimental LAN on the testbed was configured with two machines, a gateway and a Server Blade running a Windows 2000 Professional guest OS. Table 1 records the time taken for the delay queue to reach 100 and the number of connection attempts made before this time when the Blade Server is infected with the real W32/Nimda-D virus [21] and the test worm configured to scan at various rates. As this table shows, the virus throttle takes only 0.25 seconds to stop Nimda, which scans at a rate of around 200 connections per second, from spreading, and under 5 seconds to stop any application that scans at a rate of 20 connections or more per second. In the Nimda case, the throttle only allows one packet out on to the wire before networking is shut down, and a maximum of 5 packets for the test worm configured to scan at 20 connections per second. The stopping times for the SQLSlammer worm [22] of a UDP implementation of the throttle have also been included in Table 1 and show that the throttle is able to stop this worm in 0.02 seconds.

4.2.2 Mobile code propagation

In order to test the effectiveness of the virus throttle at reducing the propagation of a real worm, the testbed was set up with one class C network containing 16 Server Blades running a VMware-encapsulated copy of Windows 2000 Professional vulnerable to the W32/Nimda-D virus [21]. A machine which acted as the default gateway and DHCP server for this network was also configured. One of the Windows machines was then infected with W32/Nimda-D and its progress through the network observed by gathering data using the sniffer configured as described above. This data was then analysed and the time at which each system became

connections per second	**stopping time**	**allowed connections**
	Nimda	
120	0.25s	1
	Test Worm	
20	5.44s	5
40	2.34s	2
60	1.37s	1
80	1.04s	1
100	0.91s	1
150	0.21s	0
200	0.02s	0
	SQLSlammer	
850	0.02s	0

Figure 3: Average time taken by the throttle to stop real and test worms

infected with the Nimda virus determined. By varying the number of machines on the network which had the throttle installed and by repeating each experiment 10 times, we were able calculated the average time taken for a given number of machines to be infected assuming that a certain percentage of the machines on the network had virus throttles installed. These results are shown in Figure 4.

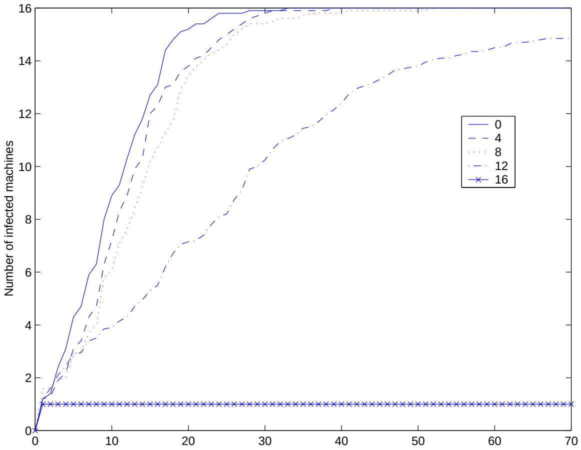

Figure 4: Infection times for different numbers of installed virus throttles (Nimda)

As Figure 4 shows, when no throttles are installed on any of the 16 machines Nimda takes on average just over 20 minutes to spread to all 16 machines. This time is slightly reduced when 25% and 50% of machines have throttles installed. However, when at least 75% of machines are installed with the throttle, Nimda is only able to spread to on average around half the machines on the network in the same amount of time, and has not spread to all 16 machines even after over 70 minutes. This represents both a decrease of 50% in the number of machines infected and a substantial increase the time taken for the worm to spread. When every machine is installed with the throttle, the worm is unable to spread at all.

Aside from the damage caused by the malicious payloads of many high-speed worms, these worms often cause denial-of-service attacks through the amount of network traffic they generate. The Sapphire worm caused network uplinks to become saturated due to the sheer quantity of traffic infected machines generated, whereas Nimda, while generating substantially less quantities of network traffic, caused routers and firewalls to fall over due to their inability to process the increased number of connection requests generated by infected machines. Figure 5 shows, for the experiment described at the start of this section, the traffic load over time on the test network. Here, the effect of having a network protected by the throttle is even more marked, with an approximately 25% reduction in viral traffic when only 25% of the network in protected by the throttle. When 50% of the machines are throttled the traffic load is reduced by over half, and when 75% are throttled to around 10% of its unthrottled average. Having all machines installed with the throttle quickly reduces all viral traffic to *zero*. These results show the ability of the virus throttle to substantially reduce the amount of network traffic generated by mobile code.

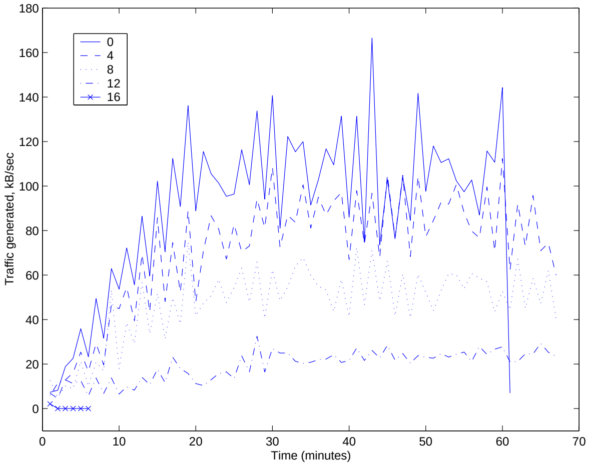

Figure 5: Traffic loads for different numbers of installed virus throttles (Nimda)

4.2.3 User trials

While the previous section shows that the virus throttle is extremely effective in slowing and stopping the spread of certain classes of worms, an equally important aspect in determining its overall utility is establishing its impact on the performance of applications which legitimately establish connections in the course of normal user behaviour. If the throttle, for example, prevents users from accessing their emails, or slows this down to an unacceptable speed, it will soon be uninstalled.

From preliminary tests in which the throttle was installed on a number of researchers' machines, we received no reports of impaired network performance. We are currently involved in a series of much larger user trials in which the throttle is installed on the gateway used by a considerable number of users running a wide range of network-capable applications under several different operating systems. Initial results also point to no noticeable degradation in network performance.

To further explore the interaction of the throttle with legitimate network-aware applications, we have also implemented a throttle simulator which takes packet traces as input. To gather these packet traces we configured a monitoring port on a 80-port switch on HP's internal network and ran a packet sniffer on this port. This has allowed us to gather large quantities of data over extended periods of time from a range of different machines. Separating out these traces on a machine-by-machine basis and then running them through the simulator allows us to rapidly assess the applicability of the throttle to different classes of machines running different services. Preliminary results from the simulator show that the majority of network traffic from a throttled machine passes onto the network undelayed, and that none of the large quantities of legitimate network traffic we have gathered is mistaken for viral traffic. A fuller discussion of the applicability of our approach is given in [27].

5 Conclusion

In this paper we have, after presenting the necessary background, described in detail the implementation and testing of a virus throttle. From the tests described above we have been able to show that the virus throttle is highly effective in detecting, slowing and stopping both real worms such as W32/Nimda-D and a test worm configured to scan at different rates. Our results also show that the virus throttle can substantially reduce the global spread of a worm, and hence the amount of network traffic produced.

In conclusion, this paper has demonstrated virus throttling to be a powerful tool in the prevention of high-speed worm propagation. We believe that throttling, when combined with current signature-based methods, will be an essential ingredient in any multilayered anti-virus solution.

References

[1] D. P. Bovet and M. Cesati. *Understanding the Linux kernel.* O'Reilly & Associates, 2002.

[2] D. Bruschi and E. Rosti. Disarming offense to enable defense. In *Proc. of the New Security Paradigms Workshop 2000*, pages 69–75, Ireland, 2000.

[3] D. Bruschi and E. Rosti. AngeL: A tool to disarm computer systems. In *Proc. of the 2001 Workshop on New Security Paradigms*, 2001.

[4] CERT Advisory CA-2001-19. CERT Coordination Center. http://www.cert.org/advisories/CA-2001-19.html.

[5] D. Chess. The future of viruses on the Internet. Presented at the Virus Bulletin International Conference, October 1-3, 1997.

[6] F. Cohen. Computer viruses - theory and experiments. In *Proc. of the 7th Security Conference*, pages 143–158, 1984.

[7] F. Cohen. A formal definition of computer worms and some related results. *Computers and Security*, 11(7):641–652, 1992.

[8] F. Cohen. *A short course on computer viruses.* John Wiley & Sons, Inc., 1994.

[9] fyodor. The art of port scanning. *Phrack Magazine*, Volume 7, Issue 51, 11 of 17, 1997.

[10] R. A. Grimes. *Malicious mobile code.* O'Reilly & Associates, 2001.

[11] HP Procurve Series 2500 switches - management and configuration guide. Hewlett-Packard Company, 2000.

[12] HP Blade Server bh7800 service guide. Hewlett-Packard Company, 2002.

[13] kossak and lifeline. Building into the Linux network layer. *Phrack Magazine*, Volume 9, Issue 55, 12 of 16, 1999.

[14] R. Lemos. Year of the worm. http://new.com.com/2102-1001-254061.html.

[15] E. Messmer. Behaviour blocking repels new viruses. http://www.nwfusion.com/news/2002/0128antivirus.html.

[16] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the Sapphire/Slammer worm. http://www.cs.berkeley.edu/ nweaver/sapphire/.

[17] J. Postel. Transmission control protocol. RFC 793, DARPA, 1981.

[18] A. Rubini and J. Corbet. *Linux device drivers.* O'Reilly & Associates, 2001.

[19] H. G. Schuster. *Complex Adaptive Systems.* Scator Verlag, 2001.

[20] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proc. of the 9th USENIX Security Symposium*, pages 185–197, 2000.

[21] W32/Nimda-D. Sophos Anti-Virus. http://www.sophos.org/virusinfo/analysis/w32nimdad.html.

[22] W32/SQLSlam-A. Sophos Anti-Virus. http://www.sophos.org/virusinfo/analysis/w32sqlslama.html.

[23] W. R. Stevens. *UNIX network programming.* Prentice Hall, 1990.

[24] tcpdump. http://www.tcpdump.org.

[25] J. Twycross. Studying mobile code: an experimental setup. Technical report, Hewlett-Packard Labs, 2002.

[26] VMware Workstation 3.2. VMware Inc. http://www.vmware.com.

[27] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *Proc. of the ACSAC Security Conference*, Las Vegas, Nevada, 2002.

[28] M. M. Williamson. The design, implementation and testing of an email throttle. Submitted to the Annual Computer Security Applications Conference, Las Vegas, N.V., 2003.

[29] M. M. Williamson and J. Leveille. An epidemiological model of virus spread. To appear in the Proceedings of the Annual Virus Bulletin Conference, 2003.

[30] M. M. Williamson, J. Twycross, J. Griffin, and A. Norman. Virus throttling. In *Virus Bulletin*, U.K., 2003.