Throttling Viruses: Restricting propagation to defeat malicious mobile code

Matthew M. Williamson
HP Labs Bristol, Filton Road, Stoke Gifford, BS34 8QZ, UK
matthew_williamson@hp.com

Abstract

Modern computer viruses spread incredibly quickly, far faster than human-mediated responses. This greatly increases the damage that they cause. This paper presents an approach to restricting this high speed propagation automatically. The approach is based on the observation that during virus propagation, an infected machine will connect to as many different machines as fast as possible. An uninfected machine has a different behaviour: connections are made at a lower rate, and are locally correlated (repeat connections to recently accessed machines are likely).

This paper describes a simple technique to limit the rate of connections to "new" machines that is remarkably effective at both slowing and halting virus propagation without affecting normal traffic. Results of applying the filter to web browsing data are included. The paper concludes by suggesting an implementation and discussing the potential and limitations of this approach.

1 Introduction

This paper presents a technique to automatically control the spread of computer viruses. This addresses an important problem, as while the speed of propagation of viruses has increased dramatically over recent years [8], the speed of responses has not increased as quickly.

In order to combat these fast spreading viruses, there is a need to respond automatically to the virus before it has been identified (which is often the work of a human). Automatic systems are problematic for most security applications because of the problem of false positives. Most responses are irrevocable e.g. quarantining files, shutting down, patching etc. and if these actions were carried out on every false positive error the performance would be poor.

A possible solution to this is to use "benign" responses, those that slow but do not stop the virus. This can then hopefully slow the propagation of the virus until an irrevocable human-driven response can be applied.

The technique relies on the observation that under normal activity a machine will make a fairly low rate of outgoing connections to new or different machines, and that connections are locally correlated e.g. it is more likely to connect to the same machine regularly than to different machines. This contrasts with the fundamental behaviour of a rapidly spreading virus. An infected machine will attempt to make as many outgoing connections as possible (high rate), to as many different machines as possible (no local correlation). This observation makes sense for desktop machines and for servers (which primarily handle incoming connections), and makes less sense for machines running notification services. Evidence from [9, 10] supports this observation, showing that most machines interact with a few other machines.

The idea is to implement a filter on the network stack that uses a series of timeouts to restrict the rate of connections to new hosts such that most normal traffic is unaffected. Any traffic which attempts connections at a higher rate is delayed. Newness is defined by temporal locality (comparing the connection to a short list of recently made connections). The delays introduced by the timeouts are such that false positives (occasional periods when normal traffic is at a higher rate than allowed) are tolerated with small delays, but malicious traffic (at a rate much higher than allowed) is heavily penalised. The paper will show that for web browsing traffic, normal traffic is not impeded by even quite low allowed rates (0.5–1 connections per second (cps)), a rate that would severely impede most viruses (the Code Red virus [2, 5] propagated at a rate of at least 200 cps[1]).

This approach is related to "behaviour blocking" [12] where a policy for the allowed behaviour of an application is defined, and infringements of that policy are detected and reported. Products in this space are sold by [13] and [6]. Techniques for filtering network traffic are also relevant, either based on source IP addresses (ingress/egress filtering [7]), or based on statistical measures of network traffic (e.g. products from [11]). This work differs from all of these approaches because it consists of an automatic response with a benign action. This makes it faster and more tolerant to

[1] Measured with our test setup on a slow machine (Win2K, 500 MHz).

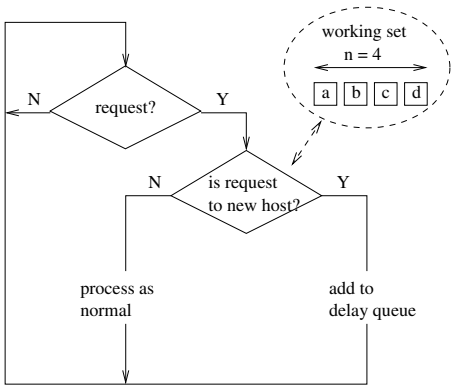

Figure 1. Processing loop for new requests. Whenever a request is made it is compared against a list of recently used hosts (shown in the dotted ellipse) to determine whether it is new or not. If the connection is to a new host, (e.g. to host 'e' which is not in the working set) it is placed on a delay queue to await processing, while if it is not new (say to host 'b'), it is processed immediately.

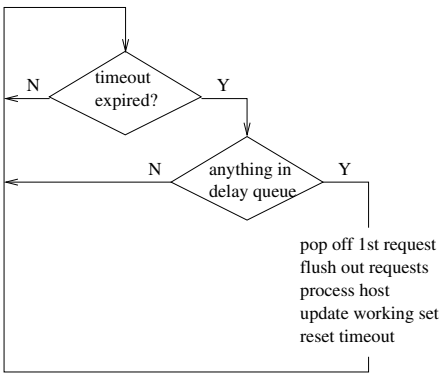

Figure 2. Processing loop for rate limiting. Whenever the timeout expires, if there is anything in the delay queue, the first request on that queue is processed. This involves flushing out any other requests to the same destination from the queue, making the connections, updating the working set with the new destination, and resetting the timeout.

false positive errors in detection.

Work in network intrusion detection e.g. [9, 10] exploits the locality of interactions between machines to detect anomalies in network traffic. The locality makes the problem of learning the "normal" behaviour more tractable. In these systems, detection errors are picked up by a human operator. This work differs because the outgoing connections of each machine are monitored rather than traffic on the network as a whole. In addition this work explicity exploits locality and uses benign responses to handle errors.

Other related work by [1] suggests ways to limit machines so that they cannot participate in network attacks, and [14] give an example of benign responses used in an intrusion detection application.

The rest of the paper describes the filter in detail, showing how it can effectively block high rate traffic. It then examines the filter behaviour with normal traffic, using the example of web browsing. The results back up the assumptions of low rates and local correlation. Data from other protocols is presented to support the generality of this approach. The paper then suggests a possible implementation on the Windows platform and concludes by discussing some of the vulnerabilities of this approach.

2 Filter algorithm

The aim of the filter is to limit the rate of connections to new hosts. The overall system is split into two parts that run in parallel: a system to determine whether requests for connections are to new hosts; and a system based on a series of timeouts that limits the rate of those connections.

The first part is shown in Figure 1. Whenever a request is made, a check of the newness of the host is performed. If the host is new it is added to a "delay queue" waiting to be processed by the other system, and if not new it is processed as normal. Newness is determined by comparing the destination of the request to a short list of recently made connections. The length of this list or "working set" n can be varied so altering the sensitivity of the system. For example, if $n = 1$, requests other than consecutive connections to the same host will be determined as new.

The rate limit is implemented as shown in Figure 2. Every time a timeout expires, if there is anything in the delay queue (new hosts waiting to be processed), then the first request is removed to be processed. At the same time any other connections to the same destination are also processed. The connection to the host is made, and the working set is updated. This involves removing a host from the set, and inserting the host that was just processed. The replacement strategy could either be first-in-first-out or least-recently-used. The timeout is only reset when there is something in the queue so that if the next new connection is attempted after the timeout has expired it will be processed immediately.

The overall system thus allows traffic that is locally correlated to pass unrestricted (controlled by the length of the working set n), and restricts connections to new hosts to one per timeout period.

If the time between timeouts is d, then the system limits the rate of connection to new hosts $r_{allowed} = 1/d$. If an attempt is made to make connections at a higher rate r_{attack}, then every d seconds the size of the delay queue l_d will grow by $(r_{attack}d - 1)$ or at average rate

$$\dot{l}_d = (r_{attack}d - 1)/d = r_{attack} - r_{allowed} \quad (1)$$

or

$$l_d(t) = (r_{attack} - r_{allowed})t \quad (2)$$

where t is time. Since the hosts are popped off the delay queue once per timeout, the overall delay to a particular connection is equal to d times the length of the queue when that host was at the back of it, or

$$t_{delay} = dl_d(t_0) = d(r_{attack} - r_{allowed})t_0 \quad (3)$$

where t_0 is the time that the connection was placed on the queue.

Taking (2) and (3) together, if the attack rate is a lot greater than the allowed rate, then the delay queue will grow at roughly the attack rate, and the delay to the individual connections will grow as the queue length grows. Malicious code that has a high attack rate will thus be severely delayed.

An infected program can thus be quickly detected by monitoring the size or rate of increase of the delay queue. It can then be suspended or stopped, so stopping the further propagation of the virus. Figure 3 shows the size of the delay queue against time for different attack rates. If the maximum size of the delay queue is set to 200, a virus spreading at 50 cps will be halted in 4 seconds, after making only 4 connections to other machines[2].

Figure 3 also shows that for low rates of r_{attack} the queue size and thus the delays grow slowly. This means that if a normal program has a brief period where it's rate is greater than $r_{allowed}$, there will be some delay, but the delays should be small.

The system thus acts as a filter that quickly and heavily delays high rate traffic, but is tolerant to fluctuations of rate that are just higher than $r_{allowed}$. By monitoring the delay queue, rapid spreading behaviour can be quickly detected and the offending program stopped. Rates lower than $r_{allowed}$ are not affected. The parameters of the algorithm are the allowed rate $r_{allowed}$ and the length n of the working set. The following sections examine the algorithm using data from web browsing traffic, concentrating on the effects of these parameters.

[2]There may be a natural limit to the size of the queue. For example the Code Red virus uses 99 parallel threads to propagate itself. Each thread waits for its connection before attempting another. This means that the queue length will quickly reach about 99 then stay stable at that value. The output rate will still be one connection per timeout period.

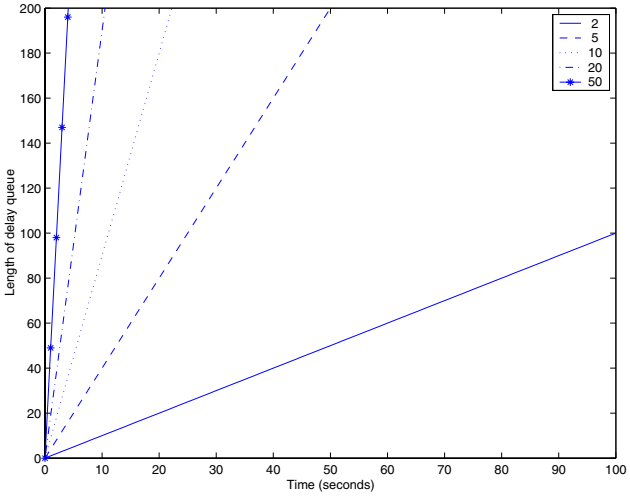

Figure 3. Figure showing the size of the delay queue against time for different attack rates in cps. This data was plotted for $r_{allowed} = 1$ cps. The size increases linearly with time with a gradient equal to $r_{attack} - r_{allowed}$. The delay to each connection is d times the queue length (in this case $d = 1$ second). Normal traffic, which might occasionally be faster than $r_{allowed}$, will also be put on the delay queue. However, the relatively low rate will mean that the queue will not grow large, and the traffic will not be greatly delayed.

3 Evaluation under normal traffic

Given that the filter can effectively delay traffic that is at a much higher rate than allowed, this section considers the ability of the filter to allow normal traffic through without delay. The normal traffic chosen is web traffic (http) because it is a common network protocol and also because a number of recent high profile (i.e. damaging) viruses used http to propagate ([2, 3]).

The browsing behaviour (time, url visited) of five fellow researchers was collected over a five month period, as detailed in Table 1. From the url the host e.g. `www.google.com` can be extracted. While this is not an enormous amount of data, it is enough to demonstrate the practicality of the idea.

The first issue is the effect of the working set size n, used to determine whether a given host is new or not. The data was run through the filter using a first-in-first-out (FIFO) replacement strategy on the working set. The total number of "new" hosts expressed as a percentage of the total number of requests is shown in Figure 4.

Four of the users have very similar profiles with one out-

Table 1. Details of the web browsing data. The data captures all of the browsing for users 1 and 2, and is a partial record for the others.

User	Time period (months)	Number of requests
1	5.67	162373
2	5.2	23535
3	4.5	7144
4	4.3	10010
5	3.46	7095

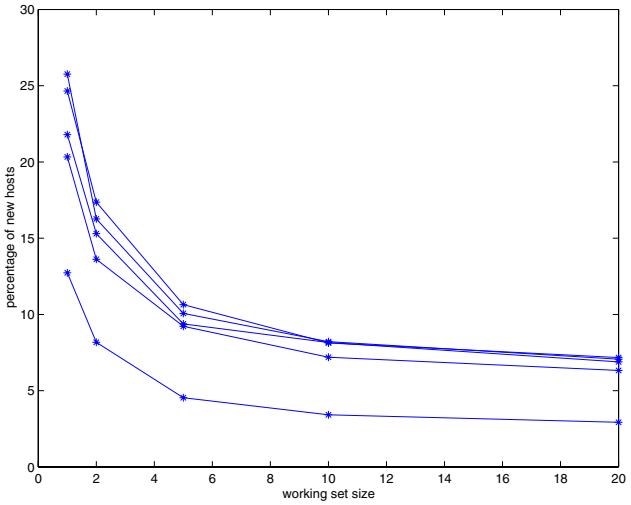

Figure 4. Graph showing the number of new hosts (expressed as a percentage of the total number of requests) as a function of working set size n, for a FIFO replacement strategy. The different lines correspond to different users. All of the users show the same pattern, the number of new hosts is reduced as n increases, and flattens out for larger n. The number of new hosts is low ($<$ 10%) showing that the data is locally correlated.

lier. As n increases, the number of new hosts decreases, with the effect tailing off for $n > 5$. This supports the assumption that the connections are locally correlated and makes sense for web browsing: the loading of each page may make multiple requests to the same site for images etc., and to different sites for advertisements. The relatively small size of working set required is also good news for implementation, showing that large amounts of memory are not needed. Other replacement strategies such has least-recently-used have also been tested, with results very similar to the FIFO case.

A second issue is the rate of connections to these new

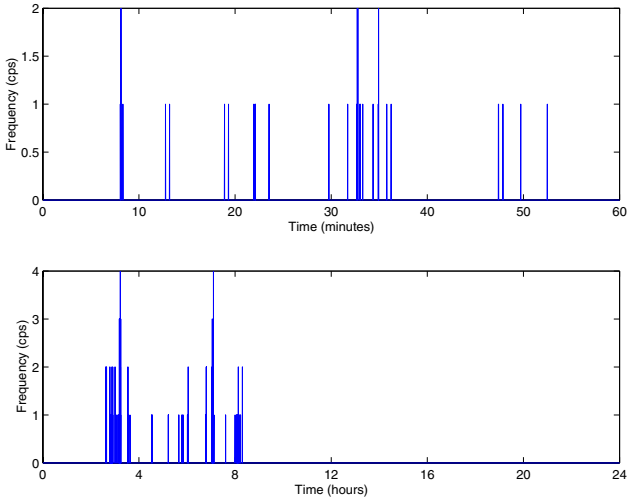

Figure 5. Plot showing local frequency against time for a one hour (top plot) and a 24 hour period. The browsing only occurs during working hours and is bursty.

hosts. This can be measured by calculating the time sequence of new hosts as above, and then sliding a time window along the sequence, measuring the local frequency. Figure 5 shows the result of this calculation, giving the local frequency against time for a one hour and 24 hour period. The data has a low frequency (generally 1–2 cps) and is bursty, with sporadic loading of pages during working hours.

A histogram of the local frequencies for a single user are shown in Figure 6, removing the zero frequency that accounts for 99.97% of the time. For small working set sizes (n) there are more new hosts, and they occur at higher frequencies, but for larger set sizes there are less new hosts, at lower frequencies. In all cases the maximum frequency is low i.e. < 5 cps, and most of the traffic is concentrated at low frequencies e.g. 1–2 cps. This suggests that a reasonable value for the allowed rate should be 1–2 cps.

The third issue is the effect of the allowed rate $r_{allowed}$. Figure 7 shows the effect of simulating the complete filter (calculation of new hosts, delay queue and timeouts) on a single users data with different values of allowed rate $r_{allowed} = 1/d$. The plot shows the number of requests that are delayed (as a percentage of the total number of requests) against the amount that they are delayed, for different values of $r_{allowed}$. The delays are low, and affect few requests: less than 1% of requests being delayed for $1 < r_{allowed} < 5$. Even for smaller rates the delays are not excessive. The data suggests that reasonable performance would be obtained with an allowed rate of 0.5–1 cps. Malicious code propagation would of course be limited to the

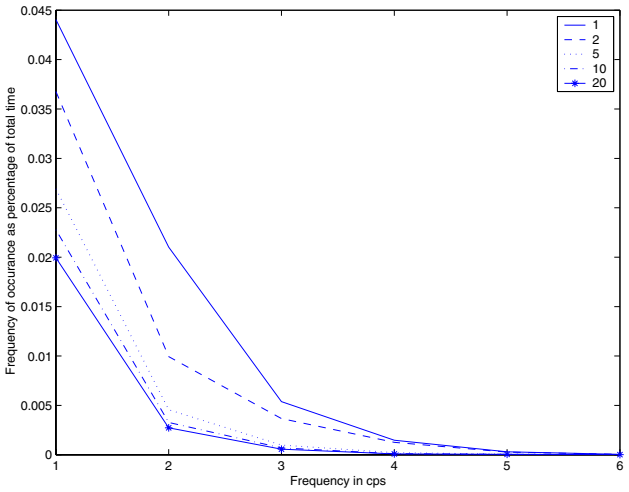

Figure 6. Local frequency plot for user 1 for different working set sizes (n). The x-axis is the local frequency observed in the data, and the y-axis measures the percentage of the total time that that frequency was observed. The zero frequency has been removed, as it accounts for 99.97% of the data. The plot shows that increasing working set size reduces the number of new hosts, with the effect becoming less as window size increases. It also shows the low frequency content of the data, with the highest frequency being 5 cps for $n = 1$, and 2–3 cps for $n > 5$.

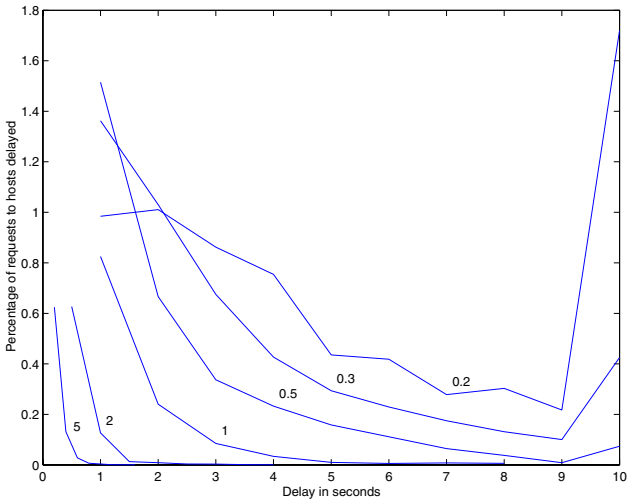

Figure 7. Plot of simulation of filter, showing the proportion of requests that were delayed (as a percentage of the total number of requests) plotted against the time that they were delayed (in seconds). The different lines correspond to different values of $r_{allowed}$. This data is for user 1, with a working set size of 5. The delays are small and few requests are affected, particularly for higher values of $r_{allowed}$ (< 1 % for $1 < r_{allowed} < 5$).

same rate.

The question remains as to which requests are actually delayed. Table 2 shows the top 6 delayed hosts for user 1. The table shows the host, the percentage occurrence of that host in the entire trace and the percentage of requests to that host that were delayed by one second. The top 6 hosts are the most delayed, however they are not very common. e.g. about a third of the requests to `ads.inet1.com` were delayed, but requests to that site form just 0.1% of the total data. The most delayed sites are advertisements, presumably because when a page loads, multiple requests are sent to the same host for html, images, frames etc.. Ads are loaded from a different host so are more likely to be delayed. The lower set of 6 hosts are ordered by occurrence. While these hosts are visited frequently ($> 10\%$ of all requests), they are delayed by low amounts (less than 5 % or 1 in 20). A similar pattern is observed for other users and other time delays.

To summarise, this preliminary analysis suggests that for web browsing the rate of connections to new hosts is generally low (< 2 cps) and that connections are locally correlated (increasing the size of the working set reduces the number of "new" hosts). It suggests that the algorithm will give low delays on normal browsing for quite small allowed rates (0.5–1 cps). In addition the most delayed sites are quite likely to be advertisements.

Even given this analysis, it is hard to determine what these delays would "feel like" to a user of the machine. An implementation is the subject of ongoing research [16], and is described in section 5.

4 Other protocols

The data presented so far has shown that traffic created by web browsing is locally correlated and thus is suitable for throttling. This section considers whether this is the case for other protocols.

Data was collected from the complete network behaviour from five machines over a 24–48 hour period. This data was then filtered to recover instances of TCP connections being initiated or UDP packets being sent. This data was then processed using the working set to determine how the number of "new" connections varied with working set size. The data was split up by destination port, choosing the nine most common ports. The actual numbers of data points for

Table 2. Delayed hosts that were delayed by 1 second for user 1, using $n = 5$ and $r_{allowed} = 1$ cps. The hosts are ordered by % occurance (number of requests to this host as percentage of all requests made) and % delayed (number of requests delayed as percentage of requests made to that host). The top six hosts are the most delayed, and interestingly they are mostly advertisement sites, and occur relatively rarely. The bottom six are the most common sites, with correspondingly lower delay rates e.g. 1 in 50 calls to `portal.hp.com` will be delayed.

Host	% occurance	% delayed
'ads.inet1.com'	0.1	33.3
'gserv.zdnet.com'	0.2	32.7
'www.vibrantmedia.com'	0.1	16.4
'ummail4.unitedmedia.com:80'	0.2	16.3
'www.computercreative.com'	0.0	15.9
'a708.g.a.yimg.com'	0.3	15.6
'gserv-cnet.zdnet.com'	17.5	5.7
"portal.hp.com'	13.5	2.2
'secure.webconnect.net'	8.9	0.1
'visit.geocities.com'	4.3	5.9
'www.ebay.co.uk'	1.4	0.4
'a248.e.akamai.net'	0.8	1.9

Table 3. Table showing number of data points for each port in the data set, together with an indication of what protocols use each port

Port	Number	Usage
53	388	dns
80	2191	http
137	12263	Microsoft windows naming (netbios)
138	23946	Microsoft file sharing
139	367	Microsoft file sharing
143	1023	imap
445	262	Microsoft directory service
8088	2290	web proxy
11000	1101	local usage ?

each port is shown in Table 3.

Figure 8 shows the results of this analysis. For some protocols, all connections are to the same machine e.g. mail (143) and web proxy (8088). Some of the traffic is locally correlated (e.g. dns (53) and http (80), Microsoft directory service (445) and file sharing (139)). Two protocols stand out as not having a locality property, Microsoft file sharing (138) and naming (137). On closer inspection the traffic on these protocols is bursty with the machine making rapid interactions with about 20 hosts, repeated every 50 seconds or so. The average rate of interactions is thus low. It thus might be possible to throttle these protocols too by developing a filter that allows bursty activity but restricts the long term average rate of connections.

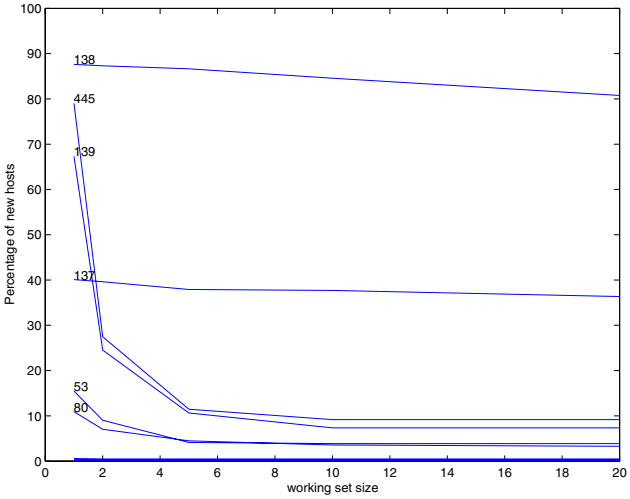

Figure 8. Graph showing percentage of new hosts against working set size for traffic on different protocols. Each line is labelled except the three at the bottom which are 143, 8088, and 11000. All of the protocols exhibit some form of locality (number of new hosts reducing with working set size) except 137 and 138.

To summarize, the data from other protocols confirms the observation that machines make outgoing connections to few other machines at a fairly low rate. This shows that throttling is immediately applicable to some protocols, but that others might require slight modifications to the algorithm.

5 Implementation

This section describes how the virus throttle could be implemented on the Windows platform. The implementation is shown in Figure 9, and is similar in architecture to that used by "personal firewall" software, e.g. [17]. The network software of a PC has a layered architecture, and the filter is best implemented as an extra layer, or shim. Thus all the traffic from the computer can be processed by the filter. The logical way to implement the delays is to delay the initial connection attempt (e.g. the first SYN packet of the connect handshake in TCP). Since no packets will leave the machine while a connection is being delayed, any networking timeouts will not be a problem. If the malicious code sets its own timeout and restarts connection attempts, these will be added to the queue as normal.

As described in Section 2, when an application is in-

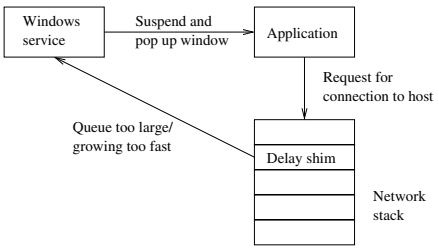

Figure 9. A possible implementation of this system on the Windows platform based on the architecture of a software firewall. The delay queue and working set is implemented as a "shim" in the network stack. The queue becoming too large suggests the presence of an infected application, and a windows service is used to suspend the application and pop up a window asking the user for directions.

fected by a virus and is attempting to propagate vigorously, the filter can detect this very quickly by monitoring the size or rate of increase of the delay queue. A suitable response is then to suspend the offending application and pop up a window to alert the user. A windows service is required for this functionality. This has two important functions: firstly the spreading of the virus is stopped (the process in suspended); and secondly the user can (hopefully) determine whether this is a real problem or an error.

For protocols like e-mail where the address used by the virus is not the machine address, the implementation needs to be more sophisticated. This is because a single mail server will handle sending mail to many different addresses, so monitoring connections to new machines will not catch an e-mail virus. The solution is a more detailed examination of the packets sent to determine the destination address for each e-mail, and applying the same filter (with longer timeout settings) to those addresses. This could be implemented in the network stack, using a proxy, or at the mail server.

6 Conclusions

This paper has presented an approach to throttling the spread of viruses by targeting their propagation. The fundamental assumption used is that for a virus to spread effectively it needs to contact as many machines as possible, as fast as possible. This is in contrast to the normal behaviour of machines, where connections to new hosts occur at a lower rate, and those connections are temporarily correlated (the probability of returning to a recently accessed host decays with time).

The algorithm developed has two parts: a method for determining whether a connection to a host is new or not, using a short list of past connections; and a method for ensuring that the rate of connections to new hosts is limited, using a series of timeouts. Data from web browsing behaviour was analysed verifying the assumptions above and showing that the rate limit can be set as low as 1 cps without causing undue delay. The nature of the filter ensures that delays to normal traffic are significantly less than those on high rate traffic, and that by monitoring the size of the delay queue the virus propagation can be quickly detected and stopped. Data from other protocols was also analyzed lending support for the generality of this approach.

This approach should be effective against scanning viruses such as Code Red [2] and Nimda [3], and also against email viruses such as I Love You [4]. Reducing the propagation rate by large factors[3] and stopping the offending application when the delay queue is too long would greatly reduce the threat of these viruses.

Computer security is an arms race, and each security advance changes the likely future threats. The most obvious consequence of widespread deployment of this technique is for viruses to become more stealthy (spread at low rates). This is not altogetherly bad as the longer the virus takes to spread the more effective human-mediated responses will be.

In a recent article [15] suggested a variety of extremely virulent theoretical worms. These included the Warhol worm that has sophisticated scanning algorithms to search for susceptible machines effectively, and the Flash worm that pre-computes the addresses of susceptible machines and is estimated to spread through the entire Internet in seconds. The technique described in this paper would be effective against full-speed versions of both worms, and a stealthy Warhol worm would not spread quickly. Unfortunately since a Flash worm knows exactly what hosts to attack, even a stealthy one will spread very quickly. For example a Flash worm spreading at 1 cps will infect 10m machines in around 100 seconds. This is slower than the 30 seconds that the full speed version might take, but is not really slow enough. Combating this type of threat is fundamentally difficult.

Further work consists of building an implementation to validate the behaviour of the system with real data and checking the sensitivity of applications to delays in their network connections.

In conclusion, this paper has shown how an automatic response to computer viruses using benign responses can be used to both slow and stop the propagation of viruses while at the same time tolerating normal behaviour. The system should be effective against all but the most sophisticated of viruses.

[3] For example 200 in the case of Code Red, assuming a propagation rate of 200 cps and a rate limit of 1 cps

References

[1] D. Brushi and E. Rosti. Disarming offense to facilitate defense. In *Proceedings of the New Security Paradigms Workshop*, Cork, Ireland, Sept. 2000.

[2] CERT. CERT Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL, July 2001. Available at `http://www.cert.org/advisories/CA-2001-19.html`.

[3] CERT. CERT Advisory CA-2001-26 Nimda Worm, Sept. 2001. Available at `http://www.cert.org/advisories/CA-2001-26.html`.

[4] CERT. CERT Advisory CA-2000-04 Love Letter Worm, May 2002. Available at `http://www.cert.org/advisories/CA-2000-04.html`.

[5] eEye Security. .ida code red worm, 2001. `http://www.eeye.com/html/Research/Advisories/AL20010717.html`.

[6] Entercept. Entercept web server, 2002. `http://www.entercept.com/products/`.

[7] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing, May 2000. RFC 2827.

[8] R. A. Grimes. *Malicious Mobile Code: Virus Protection for Windows*. O'Reilly & Associates, Inc., 2001.

[9] L. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 296–304. IEEE Press, May 1990. `http://seclab.cs.ucdavis.edu/papers/pdfs/th-gd-90.pdf`.

[10] S. A. Hofmeyr. *A Immunological Model of Distributed Detection and its Application to Computer Security*. PhD thesis, Department of Computer Science, University of New Mexico, Apr. 1999.

[11] Mazu Networks. Enforcer, 2002. `http://www.mazunetworks.com`.

[12] E. Messmer. Behavior blocking repels new viruses. Network World Fusion News, Jan. 2002. Available from `http://www.nwfusion.com/news/2002/0128antivirus.html`.

[13] Okena. Stormwatch, 2002. `http://www.okena.com/areas/products/products_stormwatch.html`.

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

[15] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium (Security '02)*, 2002. Available at `http://www.icir.org/vern/papers/cdc-usenix-sec02/`.

[16] M. M. Williamson and A. Norman. Throttling viruses II: Implementation. Technical report, Hewlett-Packard Labs, 2002. In Preparation.

[17] ZoneAlarm. Zone alarm personal firewall, 2002. `http://www.zonelabs.com/`.