# The Top Speed of Flash Worms

Stuart Staniford[*] Nevis Networks
David Moore[†] CAIDA
Vern Paxson[‡] ICSI
Nicholas Weaver[§] ICSI

## ABSTRACT

Flash worms follow a precomputed spread tree using prior knowledge of all systems vulnerable to the worm's exploit. In previous work we suggested that a flash worm could saturate one million vulnerable hosts on the Internet in under 30 seconds [18]. We grossly over-estimated.

In this paper, we revisit the problem in the context of single packet UDP worms (inspired by Slammer and Witty). Simulating a flash version of Slammer, calibrated by current Internet latency measurements and observed worm packet delivery rates, we show that a worm could saturate 95% of one million vulnerable hosts on the Internet in 510 milliseconds. A similar worm using a TCP based service could 95% saturate in 1.3 seconds.

The speeds above are achieved with flat infection trees and packets sent at line rates. Such worms are vulnerable to recently proposed worm containment techniques [12, 16, 25]. To avoid this, flash worms should slow down and use deeper, narrower trees. We explore the resilience of such spread trees when the list of vulnerable addresses is inaccurate. Finally, we explore the implications of flash worms for containment defenses: such defenses must correlate information from multiple sites in order to detect the worm, but the speed of the worm will defeat this correlation unless a certain fraction of traffic is artificially delayed in case it later proves to be a worm.

## Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

## General Terms

Security

## Keywords

worms, simulation, modeling, Flash worm

## 1. INTRODUCTION

Since Code Red in July 2001 [11], worms have been of great interest in the security research community. This is because worms can spread so fast that existing signature-based anti-virus and intrusion-prevention defenses risk being irrelevant; signatures cannot be manually generated fast enough [12, 18]. Thus much effort has gone into both analyzing the dynamics of worm spread [2, 10, 18] and designing automated worm containment mechanisms [7, 12, 13, 16, 17, 20, 24, 25].

Most of this effort has been directed against random scanning worms; here a worm instance finds further victims by guessing random Internet addresses and then attacking whatever is at that address. Scanning worms are only a subset of known worm spread algorithms [23].

The premise of a flash worm is that a worm releaser has somehow acquired a list of vulnerable addresses, perhaps by stealthy scanning of the target address space or perhaps by obtaining a database of parties to the vulnerable protocol. The worm releaser, in advance, computes an efficient spread tree and encodes it in the worm. This allows the worm to be far more efficient than a scanning worm; it does not make large numbers of wild guesses for every successful infection. Instead, it successfully infects on most attempts. This makes it less vulnerable to containment defenses based on looking for missed connections [7, 16, 24], or too many connections [20, 25].

Although flash worms have not been reported in the wild, they are of interest for two reasons:

- Flash worms are the fastest possible worms and so may be created someday by worm writers needing to control a vulnerable population with extreme speed
- Because of the off-line nature of the spread map computation, flash worms are a useful thought experiment for exploring the worst case performance of containment defenses. The spread map can be adjusted to be whatever will be most difficult for the defense, and then the worm made as efficient as possible given that constraint.

The speed of flash worms is affected by several variables. If the worm tree is $K$-way (meaning each instance infects $K$ other hosts), then the number of generations to infect $N$ vulnerable hosts is $O(\log_K N)$. The total infection time is bounded by this number multiplied by the time required to infect a generation. However, more complex issues arise: a large list of $N$ addresses must be delivered from some initial nodes, and this may be a significant con-

[*] Email: stuart@nevisnetworks.com
[†] Email: dmoore@caida.org
[‡] Email: vern@icir.org
[§] Email: nweaver@icsi.berkeley.edu

*WORM04* October 29, 2004, Washington, DC, USA

straint on speed. Thus analysis of a flash design looks at both the tree shape and the means of address distribution.

A difficulty for the flash worm releaser is a lack of robustness if the list of vulnerable addresses is imperfect. Since it is assembled in advance, and networks constantly change, the list is likely to be more-or-less out of date by the time of use. This has two effects. Firstly, a certain proportion of actually vulnerable and reachable machines may not be on the list, thus preventing the worm from saturating as fully as otherwise possible. More seriously, some addresses on the list may not be vulnerable. If such nodes are near the base of the spread tree, they may prevent large numbers of vulnerable machines from being infected by the worm. Very deep spread trees are particularly prone to this. Thus in thinking about flash worms, we need to explore the issue of robustness as well as speed.

In the rest of the paper, we explore (in Section 2) how to make flash worms very fast in light of recent knowledge about worms, and how to estimate the speed of such worms without actually releasing them. We derive the performance estimates quoted in the abstract. Next, in Section 3, we study the effect of a fraction of invulnerable systems included in the spread tree by mistake. We describe some ways of making flash worms more robust to this and explore their effectiveness. In Section 4, we look at the interaction of flash worms with worm containment systems. After exploring the small amount of related work in Section 5, we conclude in Section 6.

## 2. THE DESIGN OF FAST FLASH WORMS

In earlier work [18], we performed a simple analysis of a 7-layer 10-way spread tree with a 5KB worm. We estimated the generation time at 3 seconds (based on 50% utilization of a 256Kbps link) giving a total infection time of less than 30 seconds.

The parameter space of flash worms is much larger than that one scenario. In particular, the size of the worm constrains how quickly it can be distributed, and there are tradeoffs in the tree design. A shallow tree risks that the early nodes will be a bottleneck, while making it deeper increases the number of generations required. This slows the worm and makes it less resilient to errors.

### 2.1 Lessons from Slammer and Witty

The Slammer worm [10, 22] of January 2003 was the fastest scanning worm to date by far and is likely close to the lower bound on the size of a worm. Data on observed Slammer infections (and on those of the similar Witty worm) provide us with estimates for packet rate and minimum code size in future flash worms.

Slammer infected Microsoft's SQL server. A single UDP packet served as exploit and worm and required no acknowledgment. The size of the data was 376 bytes, giving a 404 byte IP packet. This consisted of the following sections:

- IP header
- UDP header
- Data to overflow buffer and gain control
- Code to find the addresses of needed functions.
- Code to initialize a UDP socket
- Code to seed the pseudo-random number generator
- Code to generate a random address
- Code to copy the worm to the address via the socket

It is difficult to see how a worm could be much smaller. This functionality is all essential and so the only way to shrink the worm further is to tighten the instructions used to implement it.

It is also hard to see how a worm could emit packets much faster than this. Minimal code is in the loop to generate a new address and send a packet. So it is particularly interesting to see how fast Slammer emitted packets in practice. We had access to two datasets of outbound packet traces from active slammer infections: one from a national laboratory (with 13 infections), and one from a large university campus (with 33 infections). A histogram of the packet rates is shown in Figure 1. The average speed of these Slammer hosts is 4700 packets per second (pps).

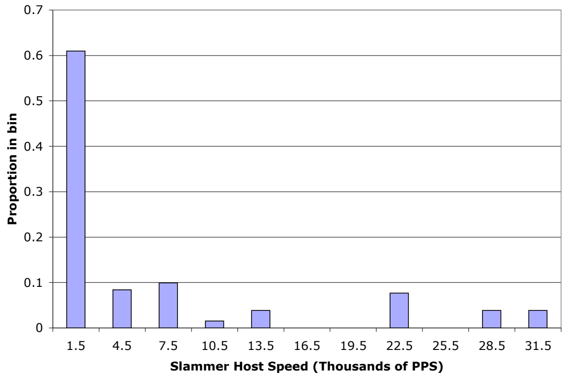

**Figure 1: Distribution of speeds of 46 outbound Slammer infections. The histogram bins are 3000 pps wide, with the center value shown on the $x$-axis.**

There are some systematic problems with the distribution in Figure 1. Firstly, the data come from only two sites which are not representative. Most obviously, these sites likely have larger bandwidth than the average site. Additionally, the 46 observations are not independent of each other since they had to share uplink bandwidth with the other infections at their site.

The reader might wonder why we don't study the distribution of *inbound* scanning rates of Slammer infections (scaling up the rates by the ratio of total Internet address space to the address space of the observing sites). The reason is that the Slammer random number generator was flawed and had a complex and system-dependent cycle structure making it very difficult to extrapolate from observations at one site to the Internet.

A data set without this problem was that for the Witty worm, a single packet UDP scanning worm which infected ISS intrusion detection systems in March 2000 and which had a good uniform random number generator. As described in more detail in [14], a representative one hour period of the Witty infection was chosen and the observed scan rate over that hour was measured for each infected host. Since the observations were made on UCSD and CAIDA's /8 network telescope, the sending rate is estimated by scaling the observed rate by 256, the inverse of the fraction of address space monitored. Figure 2 shows the cumulative distribution function of the estimated sending rates. Over 60% of the distribution is between 11 pps and 60 pps. For the average size of a Witty packet, 1090 bytes, this corresponds to being between 96kbps and 512kbps (typical broadband speeds).

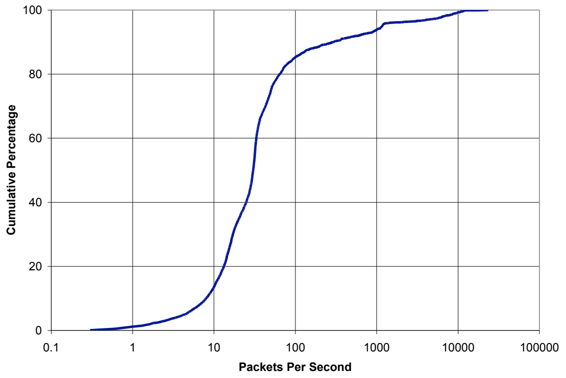

**Figure 2: Cumulative distribution of packet rates for Witty infected addresses.**

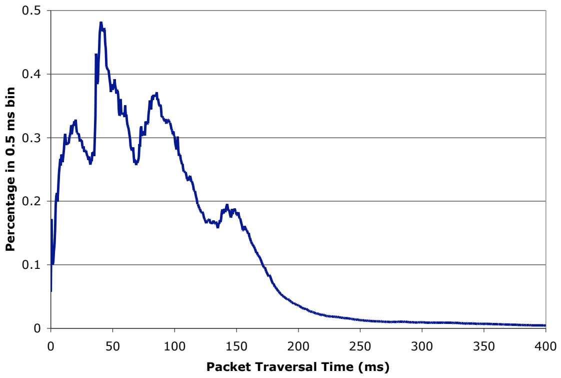

**Figure 3: Histogram of globally averaged Internet latencies for February 2004 as determined by CAIDA's Skitter infrastructure. See the text for details.**

## 2.2 First Estimates for a UDP Flash Worm

The Witty hosts show a larger range of speeds than the Slammer hosts, even noting the more than doubled size of Witty packets. It seems the well-connected sites where we have outbound Slammer data represent the upper end of the speed distribution. In this paper, however, we are interested in the *top* speed of flash worms. We can assume that an attacker is sophisticated enough to estimate bandwidth in advance (through known techniques [3]) and pick fast sites. Thus we made the assumption that our flash worms draw critical internal nodes in the spread tree from the top $5\%$ of the Witty speed distribution, with the speed scaled linearly for packet size. The $95\%$ point of this data occurs at 1000 pps or about 1 Mbps. At the Slammer packet size of $404$ bytes, this would be 2700 pps.

A key issue in single-packet flash worm design is that the time between successive packets from a fast host is small compared to the time to cross the Internet. The global Internet latency distribution is shown in Figure 3.

This figure comes from round-trip time (RTT) measurements in CAIDA's skitter datasets [1]. We used observations from 22 geographically diverse measurement boxes to over one million globally distributed destinations for the entire month of February 2004. There were over 182 million RTT measurements in this period. The 22 monitors were divided into four geographic regions: Asia/Oceania (4), North America - West (7), North America - East (6), and Europe (5). Among monitors within the same region, the RTT distributions were similar and were combined to produce a single RTT distribution per region. Since the probed destinations are geographically diverse, each of these distributions represent the expected latencies for a host in that region to reach a random host on the Internet. To compute a distribution representing the latencies between arbitrary hosts, we averaged these regional distributions based on the prevalence of hosts in each region. For publicly ! routed IPv4 addresses, the four regions above have about the same weight. Finally, we halved the RTT measurements to give one-way latencies. We checked that the exact weighting of the four regions does not grossly change the shape of the distribution.

The mean of the latency distribution is 103ms, corresponding to an average of 277 packets issued (at 2700 pps) before the first one arrives. This motivates a design in which the flash infection tree is shallow and broad. Furthermore, the attacker has control over the initial node from which the worm is launched. So this should be a host with an unusually high capacity to deliver network packets located at a site with excellent connectivity to the Internet. We assume the initial node can deliver 750 Mbps of worm IP packets ($75\%$ of a 1 Gbps link)[1]. That is a little over 240000 Slammer-sized packets per second.

In this paper, we assume that the target vulnerable population is $N = 1000000$ (one million hosts—somewhat larger than the $360,000$ infected by Code Red [11]). Thus in much less than a second, the initial host can directly infect a first generation of roughly $5,000$ - $50,000$ intermediate nodes, leaving each of those with only 20-200 hosts to infect to saturate the population. There would be no need for a third layer in the tree.

This implies that the address list for the intermediate hosts can fit in the same packet as the worm; 200 addresses only consumes 800 bytes. A flash version of Slammer need only be slightly different than the original: the address list of nodes to be infected would be carried immediately after the end of the code, and the final loop could traverse that list sending out packets to infect it (instead of generating pseudo-random addresses).

To optimize this design, let us begin by assuming fixed constant latency of $L$ (taken to be our average measured latency of 103ms), fixed constant bandwidth for the intermediate nodes of $b$ bytes per second (taken to be 1 Mbps the $95\%$ point on the Witty distribution), and the initial node bandwidth of B bytes per second (750 Mbps as discussed). The length of the worm is $W+4A$, where $A$ is the fixed number of addresses supplied to each node. If there are $n$ intermediate nodes, then $nA + n = N$, so that $n = \frac{N}{A+1}$. The total infection time is thus

$$t_I = \frac{N(W+4A)}{(A+1)B} + \frac{AW}{b} + 2L \qquad (1)$$

(assuming that the secondary nodes get sent a worm with a zero length address list).

This curve is shown in Figure 4. The optimum is to use a secondary-node address list of length $A = 107$. Thus in the se-

[1] The attacker may need to install special packet drivers on the compromised server to achieve this performance!

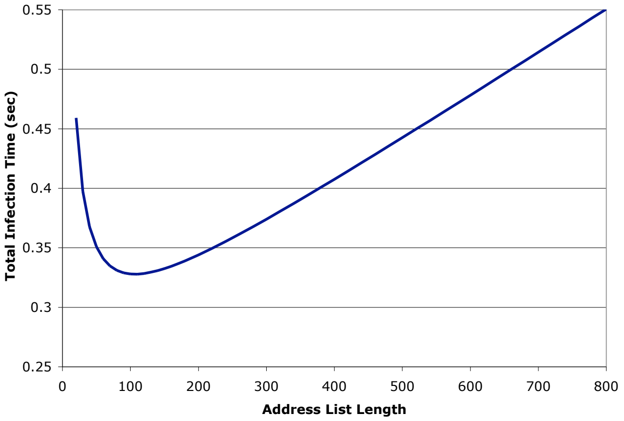

**Figure 4: Estimated speed to infect one million hosts assuming constant latency of 103ms, initial node available bandwidth of 750 Mbps, and fixed secondary node bandwidth of 1 Mbps. The worm is assumed to be 404 bytes + 4 bytes per address, and the graph plots total infection time versus the globally fixed secondary address list size.**

quel we study a worm that has $9260$ secondary nodes infecting $107$ addresses each for a total of $1000080$. The initial packets will be $832$ bytes, and with $750$ Mbps can be delivered at a little better than $118000$ pps.

Now the initial node has to place different address lists in each copy of the worm and so its code must be a little more complex; the releaser must build a special worm injector for that node. It must have a copy of the base worm, and the full address list ($4N$ bytes in size), and copy relevant parts of the address list into each copy of the worm before sending it out. The initial code can even give earlier secondary nodes more of the address list, but this turns out not to be very useful when $B >> b$.

## 2.3 Simulating UDP Flash Worm Speed

The analysis in the previous section is too crude; for two issues—Internet latency and secondary node bandwidth—it picks the average value instead of drawing from the broad distribution. To do better, we developed a Monte Carlo simulator. This models in detail the time taken in each stage of the worm:

- The linear delay before the $i'th$ packet is issued from the initial node.
- The stochastic delay for a packet to travel from the initial node to an intermediate node
- The delay associated with a stochastically picked rate of packet issue for the intermediate node.
- The stochastic latency for a packet to travel from the intermediate node to a leaf node.

For the latency distribution, we used the Skitter data in Figure 3, and for the distribution of packet rates we used the upper five percentiles of the Witty data in Figure 2. In both cases we used independent picks from the distributions in question. It was beyond the scope of this paper to optimize the worm for the latency topology of the Internet, since we do not have latency data from all points of the Internet to all other points with which to assess such optimizations. However, it is reasonable to suppose such optimization would speed up the worm a little more.

The results for our $9620 \times 107$ spread tree are shown in Figure 5, which is the cumulative distribution function for machine infection time across $100$ versions of the worm with different random seeds. The graph indicates clearly that such flash worms can indeed be extraordinarily fast—infecting $95\%$ of hosts in $510$ms, and $99\%$ in $1.2$s. There is a long tail at the end due to the long tail in Internet latency data; some parts of the Internet are poorly connected and take a few seconds to reach.

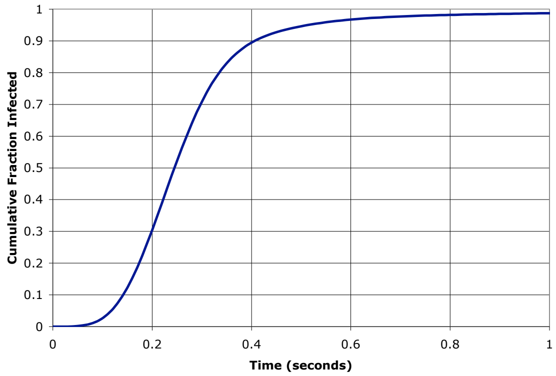

**Figure 5: The proportion of one million and eighty hosts compromised by our fastest UDP worm at different times, averaged over $100$ simulation runs.**

We note that this worm would not be limited by congestion in the Internet core. Even with all $9260$ hosts producing $1$ Mbps plus, the total bandwidth required by the worm is only O(10 Gbps), which is small on the scale of the Internet and compares favorably with the $165$ Gbps peak bandwidth usage of Slammer [10].

## 2.4 TCP Flash Worms

Can these results be extended to TCP services? If so, then our results are more grave; TCP offers worm writers a wealth of additional services to exploit. In this section we explore these issues. We conclude that top-speed propagation is viable for TCP worms, too, at the cost of an extra round-trip in latency to establish the connection and double the bandwidth if we want to quickly recover from loss.

To provide reliable data delivery, TCP requires more complex network interaction. This first affects us by requiring an extra round-trip time to establish a connection. This is true even if the target TCP receiver employs poor initial sequence numbers providing we cannot guess the sequence number in a completely blind fashion. Thus, the worm must send two additional $40$-byte packets, the initial SYN and the acknowledgment of the receiver's SYN-ACK. (In principle this latter could be bundled with the data the worm sends, but there may be TCPs that require a separate ack to complete the handshake.)

We assume that the initial window offered by the target TCP receiver is $W$ packets. While $W$ varies across the Internet, we can take it as fixed by assuming that the worm is targeting a particular vulnerable operating system and service. (In reality, $W$ is expressed in bytes, usually ranging between 8 KB and 64 KB, but here we use full-sized packets for ease of discussion.)

Assume transmitting the worm requires $k$ full-sized packets. We can divide the discussion into two categories: *small* worms, for which $k \leq W$, and *large* worms with $k > W$. Small worms fit entirely within the receiver's initial offered window, and so on establishing the connection we can immediately send the packets *en masse* without keeping state. That is, whenever the worm receives a SYN ACK, it can statelessly derive the sequence numbers needed to transmit the entire worm.

Large worms must keep state so that the worm knows which sequence numbers to use when transmitting after the initial flight. This could be avoided by transmitting stop-and-go—one packet per round trip—but at the cost of a major loss of performance. In the absence of packet loss, the sender can often get away with having more data in flight than the receiver's advertised window. This is because as long as the data streams into the receiver in order, and the receiver processes it quickly enough, the window will continue to advance as the data arrives. Thus it is possible to try to send a *large* worm in the same fashion as a *small* worm. However, this trick fails badly in the presence of packet loss: the window will cease to advance, and the loss will be *amplified* because all of the excess packets will be discarded on delivery.

How does recovery from packet loss affect worm speed? Suppose we can model packet loss as independent and occurring with probability $p$ which is constant.[2] Given this, the probability that the worm requires retransmission is $1 - (1 - p)^k$. For example, for $p = 0.005$ and $k = 8$, about 4% of the worms require retransmission. We could simply tolerate this attrition, or we could take steps to recover from it. A simple step would be to just send the worm twice (which a TCP receiver will readily tolerate). This would lower the 4% to about 0.02%, since now the probability that a given segment is not successfully received falls to $p^2 = 0.000025$.

If maximum-sized packets are 1500 bytes and $W$ is 64 KB of packets, then we can transmit $k = 43$ packets without risking packet loss amplified by packets arriving outside the window. In this case, $19\%$ of the worms will require retransmission, but retransmitting it *en masse* leads to only $0.11\%$ propagation failure, still tolerable.

If $p$ or $n$ are significantly higher than the values used above, or if in the second case $W$ is significantly lower, then sending *en masse* and recovering via redundant transmission turns less favorable. In this case, we might need to adapt more complex schemes such as using a timer-driven retransmission queue, which could slow the worm's propagation. Alternatively, the worm may be better off using the same resiliency mechanisms discussed in the next section for tolerating errors in the initial vulnerable address list. We leave this analysis for future work, but note that a few tens of KB is ample "budget" to build an effective TCP worm (perhaps one that can then bootstrap in additional functionality later). Code Red was 4KB and Nimda was 60KB. Thus, we believe the above values are fairly plausible for today's Internet and worms.

A final issue for large worms concerns the rate at which the worm transmits the $k$ packets. In the absence of the "ack clocking" provided by TCP's slow start, there is no direct guidance the sender can use in order to balance between filling the transmission pipe but also avoiding overrunning any intermediary buffers. However, the sender could round-robin through its set of current connections as a way of spatially spreading out the traffic it transmits to avoid stressing any particular path beyond its own path into the Internet core.

We believe a TCP worm could be written to be not much larger than Slammer. In addition to that 404 bytes, it needs a few more ioctl calls to set up a low level socket to send crafted SYN packets, and to set up a separate thread to listen for SYN-ACKs and send out copies of the worm. We estimate 600 bytes total. Such a worm could send out SYNs at line rate, confident that the SYN-ACKs would come back slower due to latency spread. The initial node can maintain a big enough buffer for the SYN-ACKs and the secondary nodes only send out a small number of SYNs. Both will likely be limited by the latency of the SYN-ACKs returning rather than the small amount of time required to deliver all the worms at their respective line rates.

To estimate the performance of such a small TCP flash worm, we repeated the Monte Carlo simulation we performed for the UDP worm with the latency increased by a factor of three for the handshake and the outbound delivery rates adjusted for 40 byte SYN packets. The results are shown in Figure 6. This simulation predicts $95\%$ compromise after 1.3s, and $99\%$ compromise after 3.3s. Thus TCP flash worms are a little slower than UDP ones because of the handshake latency, but can still be very fast.

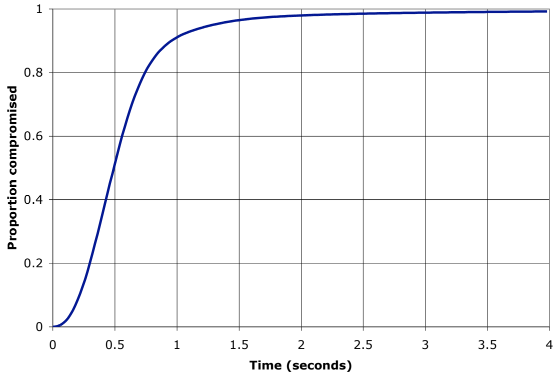

**Figure 6: Proportion of one million and eighty hosts compromised by fast TCP worm as a function of time.**

# 3. RESILIENCE TO IMPERFECT MAPS

What if a flash worm uses an imperfect list of vulnerable addresses? This could happen for several reasons. If an unreliable diagnostic procedure is used to decide what hosts are vulnerable, then the list will contain errors. Even with a perfect diagnostic procedure, the list will grow out of date as machines are patched, reconfigured, restored from old media, and so on. The older the list is, the more errors it will have. There are two kinds of list errors: a false negative, where an address is not on the list, but is in fact a reachable vulnerable address, and a false positive, where an address on the list is not really vulnerable.

The effect of false negatives is simple. If a proportion $\rho$ of vulnerable addresses are false negatives, then the flash worm can only ever saturate a $1-\rho$ proportion of addresses it should have compromised under ideal circumstances (for the worm releaser). However,

[2] This is a gross simplification, but a realistic model incorporating bursty heterogeneous loss rates [26] is beyond the scope of this paper.

the spread of the worm is otherwise unaffected, and we don't consider false negatives any further in this paper.

The effect of false positives is more significant, since key nodes high in the spread tree may be invulnerable, causing the entire subtree below that node to remain uninfected. In the rest of this section, we analyze quantitatively the impact of this on the flash worm, and discuss some cures.

## 3.1 Modeling Flash Worm Resilience

We assume that a fraction $\sigma$ addresses on the list are invulnerable, and that all nodes are equally likely to be false positives. We take the initial node to be infected at the outset with complete certainty; otherwise the attacker will choose a different initial node!

Consider a worm spread tree in which the branching factors at the $i$th level are $K_i$ for $i = 0 \ldots i_{max}$, and the branching at the leaf nodes $K_{i_{max}} = 0$. Thus our earlier simulated design had $K_0 = 9260$, $K_1 = 107$, $K_2 = 0$.

We want to compute the probability $\tau$ that a given node ends up uninfected. The worm releaser would like $\tau$ to be as small as possible, while network defenders would prefer a large value. A given node will remain uninfected if any ancestor on the tree is invulnerable, or it itself is invulnerable. To put the same thing another way, it will only get infected if it and every node above it are vulnerable. Thus for a node at level $i$, the probability of it failing to be infected is governed by

$$1 - \tau_i = (1 - \sigma)^i \tag{2}$$

$$\tau_i = 1 - (1 - \sigma)^i \tag{3}$$

To find the aggregate $\tau$, we average across all tree levels:

$$\tau = \frac{1}{N} \sum_{i=1}^{i_{max}} \left(1 - (1 - \sigma)^i\right) \prod_{j=0}^{i-1} K_j \tag{4}$$

Thus for our earlier example,

$$\tau = \frac{9260\sigma + 990820(2\sigma - \sigma^2)}{1000080} \tag{5}$$

Equation 5 is plotted in Figure 7, where it can be seen that errors in this broad shallow tree are not too serious, never more than doubling the infection failures over the invulnerable hosts. If the list is of somewhat reasonable quality, the worm will work. Similarly, this kind of worm can tolerate modest rates of packet loss without failing.

Failures become more serious in deep narrow trees. Consider a binary tree, which in twenty levels (including level zero), can cause an infection of total size 1048574 (excluding the initial node). The polynomial in $\sigma$ that arises from Equation 4 is too large to reproduce here, but we plot the infection failure rate $\tau$ as a function of the invulnerability rate $\tau$ in Figure 8.

Clearly, the binary tree is much more fragile with even a few percent of list errors greatly lowering the worm's success rate, and $20\%$ invulnerable machines causing near total failure of the worm. The reason is that the average host is only infected after many layers of the spread tree have successfully executed their infections, and the chances of one or more of those layers having an error is excellent if the list's invulnerability rate is significant at all.

The main point of Figure 8 is this: a binary flash tree with no mechanism to tolerate errors is too fragile to be useful.

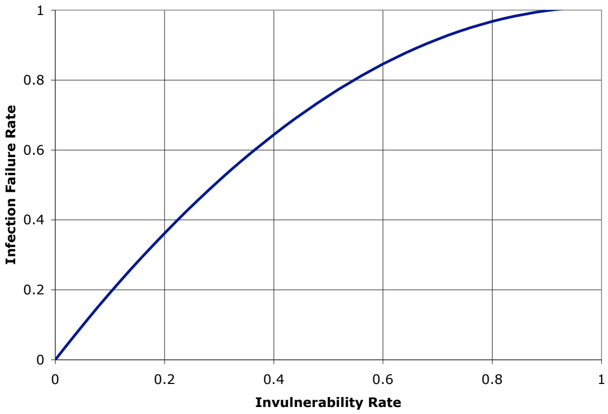

**Figure 7: Proportion of uninfected addresses vs proportion of invulnerable addresses for the $9620 \times 107$ tree.**

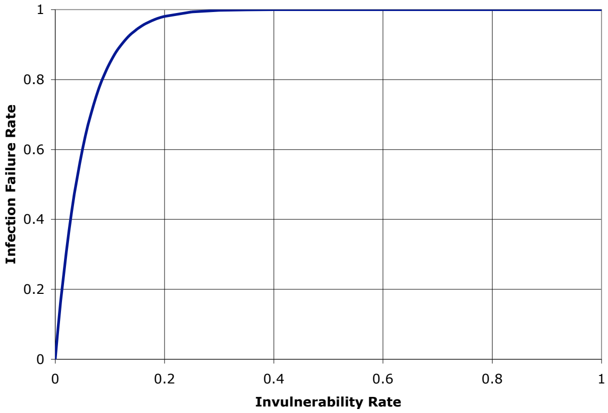

**Figure 8: Proportion of uninfected addresses vs proportion of invulnerable addresses for a $20$ level binary tree.**

## 3.2 Resilient Flash Worms

The worm writer has two tools to make flash worms more resilient. One is to add acknowledgments to the worm so that nodes will realize if their children do not get infected and will take corrective action. The second is to add redundancy into the infection tree (which will become a DAG as a result), so that if one path fails, others will be available and the tree as a whole will not be pruned.

For the case of shallow fast flash worms, the mild fragility to false positives that they suffer from can be solved very readily with acknowledgments. The only issue in need of addressing is the possibility that some intermediate nodes might be invulnerable (if a leaf node is invulnerable, there is nothing the worm can do but accept the fact that the list of truly vulnerable hosts was smaller than realized). Acknowledgments can be achieved by simply adding the initial node to the end of the list of addresses in each intermediate node worm. This will cause a copy of the worm to be sent back when each intermediate worm is done. The total downstream bandwidth required for this will be less than the upstream bandwidth needed to send out the worms originally, since the acknowledgments will not have address lists. The code on the initial node will need an extra thread to receive and collate these worm copies, but since there will be limited overlap between the time sending ou! t worms and the time receiving them, this should not cause a problem. When an acknowledgment is not received after a short time, the initial node can substitute a different address for the original node. Note that the worms at the intermediate nodes do not need to have any special code for creating acknowledgments (it's just another address to send the worm to), or handling acknowledgments (the leaf nodes do not acknowledge).

For deeper infection trees, the issues are more complex. Acknowledgments here require that the intermediate worm nodes be multi-threaded (in order to garner acknowledgments efficiently) or slow down greatly waiting for an acknowledgment after each infection attempt.

It may be simpler, and will certainly be faster, for the worm to always double infect everything. This is simple and robust. For example, in the binary tree, one way to do this is that, at each level, a node first infects its own two descendants, and then sends worm copies to the two descendants of its sibling, just in case the sibling turned out to be invulnerable. The sibling does the same in reverse. This is illustrated in Figure 9. The effect of this is to make it less likely that a portion of the tree will fail—now both siblings must be invulnerable to prevent the tree below them being infected.

To analyze the effect of this, let us again think about a node at level $i$. For it to get infected, a path down the infection tree must make it through every level. The probability that it itself will fail to be vulnerable is just $\sigma$. There are two paths down to it at level $i-1$. Both will be blocked only with probability $\sigma^2$. It turns out that at all higher levels, there are also only two paths down through that level. The reader may be surprised at this, but staring at Figure 9 for a while, it should become apparent that while a node has two parents, with four links up to the next level, because the two parents are siblings, and siblings get infected only from the same two nodes, the four links collapse onto only two grandparents, and so on indefinitely.

Thus the chance of a node failing to be infected is $\sigma^2$ at all these levels.[3] So,

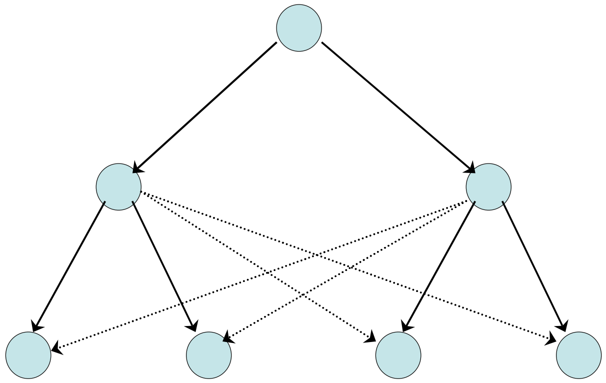

**Figure 9: Scheme for doubling up worm delivery for resilience. Each node infects its own children, and then its sibling's children, in case the sibling did not get infected.**

$$1 - \tau_i = (1 - \sigma^2)^{i-1}(1 - \sigma) \tag{6}$$

Averaging over all levels as before,

$$\tau = \frac{1}{2^{i_{max}+1} - 1} \sum_{i=1}^{i_{max}} 2^i \left[1 - (1 - \sigma^2)^{i-1}(1 - \sigma)\right] \tag{7}$$

The resilience plot for this doubled up flash worm is shown in Figure 10. As can be seen, the performance is much better than the single tree design examined in Figure 8. In that case, just a few percent defects in the target list caused a near complete loss of infection rate. With doubling up, there is still significant loss, but it is not catastrophic. At 20% defects in this list, the worm saturates half of the total it should. Thus doubling up has made an intolerable situation for the worm writer tolerable (if not excellent - the list will still need to be fairly clean).

## 3.3 $K$ independent $K$-way trees

Another approach to resilience is to note that, in a $K$-way tree, the non-leaf nodes use a fraction $1/K$ of the total nodes. This is since

$$\frac{1}{K} + \frac{1}{K^2} + \frac{1}{K^3} + \ldots = \frac{1}{K-1} \tag{8}$$

Therefore there are always $K$ different sets of internal nodes which are completely independent (in the sense that nodes used in an internal node in one tree only ever appear as leaves in the other $K-1$ trees). Thus we can start the flash worm using these multiple independent trees simultaneously, and this increases resilience, since there is no possibility of a failure in one tree also causing a failure in another tree. There is some price in code complexity, since now the worm on a leaf node must listen for possible new worms and extract and infect their address list (the broad variation in Internet

[3] This does not appear to be necessarily true in $K$-way trees where $K > 2$. There, more complex schemes for doubling up are possible in which siblings mix-and-match how they reinforce each other. In some cases, this appears to allow more than two paths through a level to some nodes. We have not yet fully explored the situation in these broader trees.

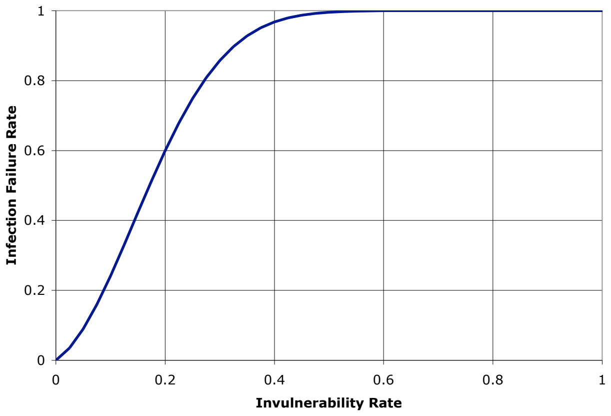

**Figure 10: Proportion of uninfected addresses vs proportion of invulnerable addresses for a twenty level binary tree with doubling up as depicted in Figure 9. This should be contrasted with Figure 8 which showed the resilience of the twenty level binary tree without doubling up.**

latencies means each tree is likely to infect at least some of its leaf nodes before other trees have finished infecting all of their internal nodes).

The failure rate in these trees can be computed as follows. There are $K$ paths to a given node. The node will be uninfected if either (with probability $\sigma$) it's invulnerable itself, or it's not invulnerable (multiply by $1-\sigma$) but no path reaches it through all vulnerable nodes. The probabilities of the $K$ paths being blocked can be combined multiplicatively since they are independent. $K-1$ of the paths are of length $i_{max}-1$, since the node is a leaf with respect to those trees. The probability of blockage in those cases is $1-(1-\sigma)^{i_{max}-1}$. The last case is shorter since the node is internal at level $i$ in that tree, so we only get $1-(1-\sigma)^{i-1}$, and we then have to average everything over $i$, weighted by the size of the level. Putting all this together:

$$\tau = \sigma + (1-\sigma)(1-(1-\sigma)^{i_{max}-1})^{K-1} \times S \qquad (9)$$

where

$$S = \frac{1}{K^{i_{max}}-1} \sum_{i=1}^{i_{max}-1} K^i \left[1-(1-\sigma)^{i-1}\right] \qquad (10)$$

We compared the following designs:

- two 20-layer 2-way trees (1048575 hosts),
- three 14-layer 3-way trees (2391484 hosts),
- four 11-layer 4-way trees (1398101 hosts),
- five 10-layer 5-way trees (2441406 hosts)
- ten 7-layer 10-way trees (1111111 hosts)
- twenty 6-layer 20-way trees (3368421)

Figure 11 summarizes the resilience of these designs. Obviously, the more independent trees, the less fragile the design is. What is surprising is that two independent trees doesn't work as well as the doubling up scheme covered in the previous section, and one has to get to four independent trees to be as good or better at all invulnerability rates. We believe this is because with two independent trees, all it takes for a host to not be infected is for there to be at least one invulnerable node on both paths to that host (the unique path in each of the independent trees). With doubling up, there are two paths to a host at any level, but since they cross over at every level, there are effectively $2^i$ paths to a node at level $i$, but they are not all independent. To prevent infection of that level $i$ node, it requires that there be two invulnerable machines *at the same level* in the tree above the node. That is less likely and so the doubling up is! more resilient than independent trees.

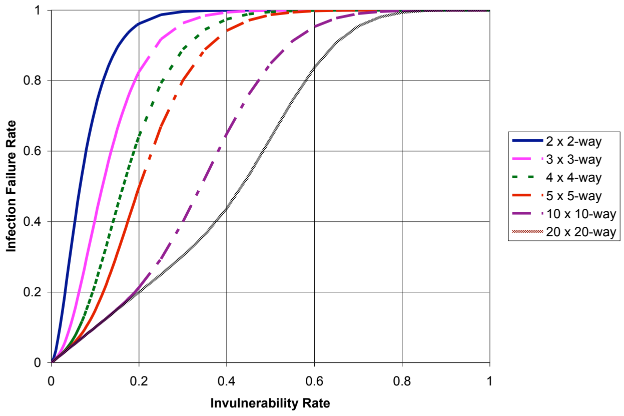

**Figure 11: Proportion of uninfected addresses vs proportion of invulnerable addresses for $K$ independent $K$-way trees for $K = 2, 3, 4, 5, 10, 20$.**

A multitree approach may not increase the bandwidth significantly except for single packet UDP worms. A multitree worm could have the first successful infection transfer over the entire worm body, while subsequent infections only transfer the hitlist. Thus the additional bandwidth cost for a $K$ tree worm is $K$ times more transfers of the hitlist, but no additional transfers of the worm body. For a TCP worm, the SYN-ACK to a new infection attempt could carry information about whether the worm was needed, or only the hit-list. However, for single packet UDP worms, the hitlist will be in flight before anything is known about the need for it, so there, $K$-way trees increase total bandwidth demand by a factor of $K$.

Likewise, the multitree approach does not slow the worm. Rather, it offers a faster infection tree than a conventional binary flash worm, as it is the first appearance in the infection trees where a system will be compromised, offering some of the benefit of a shallower worm.

However, it will make the worm somewhat more vulnerable to detection, and to this we now turn.

# 4. AVOIDING CONTAINMENT DEFENSES

There are several mechanisms which a flash worm author could employ to avoid worm defenses: slow the worm to avoid detection, reduce the degree $K$ at each node in the tree to make the traffic anomaly signal less clear, or add redundancy into the flash worm's propagation network to route around incompletely-deployed worm defenses.

Slowing the worm would work well to avoid existing scanning containment techniques. The worm could avoid the Williamson new-destination metric [25] by limiting each instance to scanning one new destination per second. It likely can avoid TRW-based detectors [6] without even needing to slow down, because TRW relies upon scanners making mostly failed connections, while a flash worm's connections will often be successful.

Even without rate-limiting, a flash worm using a binary tree structure (or, more generally, low $K$) will be undetected by either scan-containment algorithm, because the number of victims contacted by each copy of the worm is lower than the threshold required for the algorithms to make a decision.

Similarly, a flash worm naturally avoids dark-address/dark-port detectors [4, 9], as the worm author already knows which target systems are probably live and which are nonexistent. This is again because the scanning-related anomalies (contacting otherwise-unused addresses) do not appear in a flash worm.

However, there are two detectors likely to detect a flash worm: honeyfarms [19] and EarlyBird-style network detectors [15, 8]. During the hitlist creation, it is difficult for the attacker to distinguish between the honeypots and live systems, so the flash worm's targeting list will likely include some of the honeypots. Similarly, the EarlyBird-style detector (searching for statistical anomalies in the frequency of common content patterns) should detect a flash worm unless the attacker can suitably randomize the exploit or route around the detection locations. Slowing a flash worm's propagation might allow it to evade an EarlyBird detector, but would not help in evading honeypot detectors. Thus, even a very slow flash worm, with each copy contacting a new victim hourly, would probably not be stealthy when considered by humans.

Additionally, for a very high-fanout flash worm, either a Williamson new destination metric [25, 20] or a similar superspreader detector [21] could detect and block the point of initial infection. This would only work if two conditions hold: the flash worm has very high fanout (optimized for speed), and the defense is present at initial point of infection. If a high-fanout worm uses multiple initial points of infection, *all* the initial infection points would need to be protected by containment devices for such devices to stop the spread.

Yet detection is only part of the problem: the worm must also be blocked if a defense is to be effective. Worms move too fast to wait for human response, and an automated response must outrun the worm [13]. This poses a problem for EarlyBird detectors, which must wait until they have enough evidence to draw a conclusion, by which time the worm is well on its way. A honeypot may succeed in detecting the worm early, but what can it tell the rest of the network that will be effective everywhere within a second? Thus, it is likely that detection of flash worms will need to be done broadly in the network, and suspicious but not conclusive patterns of traffic deliberately delayed until a conclusion can be reached.

It appears that the optimum solution for the attacker—considering the plausible near-term worm defenses—is for a flash worm author to simply *ignore* the defenses and concentrate on making the worm as fast and reliable as possible, rather than slowing the worm to avoid detection. Any system behind a fully working defense can simply be considered as resistant, which the worm author counters by using the resiliency mechanisms outlined in the previous sections, combined with optimizing for minimum infection time.

Thus, for the defender, the current best hope is to keep the list of vulnerable addresses out of the hands of the attacker.

# 5. RELATED WORK

Flash worms were first discussed in earlier work by the present authors [18] in conjunction with a discussion of Warhol worms (optimized scanning worms). In that reference, we just introduced the concept of flash worms, and pointed out that they would be very fast, since the saturation time was the product of the logarithm of the size of the vulnerable population times the latency for each generation to infect the next. Since the latter could reasonably be expected to be in the seconds at worst, we argued that a 7 generation 10-way design could comfortably saturate a million vulnerable addresses in less than 30 seconds. We mentioned that the flash design would be fragile and that the fragility could be addressed by doubling up (or $n$-replication), but performed no quantitative analysis on the resilience under any of these schemes.

A different class of very fast worms was noted by Hindocha and Chien [5]. They observed that Instant Messenger (IM) networks can support very rapid topological [23] worms due to a combination of sizable buddy lists and short latency times for sending of messages. They perform back-of-the-envelope calculations for various notional parameter sets describing the topology of the IM network and the generational latency, and come up with values ranging from six seconds to 157 seconds to saturate $500,000$ machines. Thus these modes are slower than a flash worm, but still respectably fast, assuming Hindocha and Chien's approximations of IM topology are sufficiently accurate.

The fastest worm seen in the wild so far was Slammer [10]. That was a random scanning worm, but saturated over $90\%$ of vulnerable machines in under 10 minutes, and appears to have mainly been limited by bandwidth. The early exponential spread had an $8.5$s time constant.

# 6. CONCLUSIONS

In this paper, we performed detailed analysis of how long a flash worm might take to spread on the contemporary Internet. These analyses use simulations based on actual data about Internet latencies and observed packet delivery rates by worms. Flash worms can complete their spread extremly quickly — with most infections occuring in much less than a second for single packet UDP worms and only a few seconds for small TCP worms. Anyone designing worm defenses needs to bear these time factors in mind.

Further, we analyzed the resiliency of flash worms to errors in their target lists and to automated worm containment defenses. Shallow trees are fairly resilient to list errors, but more vulnerable to containment defenses. Deep trees are very hard to contain, but need additional resiliency mechanisms to tolerate an imperfect list. Given those mechanisms, flash worms using deep trees can tolerate modest proportions of list errors or containment defenses.

# 7. ACKNOWLEDGMENTS

This research was supported in part by Nevis Networks, NSF Trusted Computing Grant CCR-0311690, Cisco Systems University Research Program, DARPA FTN Contract N66001-010108933, NSF grant IDR/ANI-0205519, and NSF/DHS under grant NRT-0335290. Gary Grim first suggested to us the idea of a stealthy Internet worm with a precomputed spread tree.

# 8. REFERENCES

[1] CAIDA. Skitter Datasets. http://www.caida.org/tools/measurement/skitter/.

[2] Z. Chen, L. Gao, and K. Kwiat. Modeling the Spread of Active Worms. In *IEEE INFOCOM*, 2003.

[3] C. Dovrolis, R. Prasad, N. Brownlee, and k. claffy. Bandwidth Estimation: Metrics, Measurement Techniques, and Tools. *IEEE Network*, 2004.

[4] Forescout. Wormscout, http://www.forescout.com/wormscout.html.

[5] N. Hindocha and E. Chien. Malicious Threats and Vulnerabilities in Instant Messaging. Technical report, Symantec, 2003.

[6] J. Jung, V. Paxson, A. W. Berger, and H. B. Nan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *2004 IEEE Symposium on Security and Privacy, to appear*, 2004.

[7] J. Jung and S. Schechter. Fast Detection of Scanning Worms Using Reverse Sequential Hypothesis Testing and Credit-Based Connection Rate Limiting. *Submitted to Usenix Security 2004*, 2004.

[8] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the 14th USENIX Security Symposium*. USENIX, August 2004.

[9] Mirage Networks. http://www.miragenetworks.com/.

[10] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Magazine of Security and Privacy*, pages 33–39, July/August 2003 2003.

[11] D. Moore, C. Shannon, and J. Brown. Code-Red: a Case Study on the Spread and Victims of an Internet Worm. In *Proceedings of the Second Internet Measurement Workshop*, pages 273–284, November 2002.

[12] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code, 2003.

[13] D. Nojiri, J. Rowe, and K. Levitt. Cooperative Response Strategies for Large Scale Attack Mitigation. In *Proc. DARPA DISCEX III Conference*, 2003.

[14] C. Shannon and D. Moore. The Spread of the Witty Worm. *To appear in IEEE Security and Privacy*, 2004.

[15] S. Sing, C. Estan, G. Varghese, and S. Savage. The EarlyBird System for Realtime Detection of Unknown Worms: UCSD Tech Report CS2003-0761.

[16] S. Staniford. Containment of Scanning Worms in Enterprise Networks. *Journal of Computer Security, to appear*, 2004.

[17] S. Staniford and C. Kahn. Worm Containment in the Internal Network. Technical report, Silicon Defense, 2003.

[18] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*. USENIX, August 2002.

[19] The Honeynet Project. http://www.honeynet.org/.

[20] J. Twycross and M. M. Williamson. Implementing and Testing a Virus Throttle. In *Proceedings of the 12th USENIX Security Symposium*. USENIX, August 2003.

[21] S. Venkataraman, D. Song, P. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders.

[22] A. Wagner, T. Dubendorfer, B. Plattner, and R. Hiestand. Experiences with Worm Propagation Simulations. In *Proceedings of the 2003 ACM workshop on Rapid Malcode*, pages 34–41, October 2003.

[23] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A Taxonomy of Computer Worms. In *The First ACM Workshop on Rapid Malcode (WORM)*, 2003.

[24] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. *Submitted to Usenix Security 2004*, 2004.

[25] M. M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Mobile Malicious Code. In *ACSAC*, 2002.

[26] Y. Zhang, N. Duffield, V. Paxson, and S. Shenker. On the Constancy of Internet Path Properties. In *Proc. ACM SIGCOMM Internet Measurement Workshop*, November 2001.