Evaluation of Worm Containment Algorithms and their Effect on Legitimate Traffic *

Mohamed Abdelhafez
George F. Riley

Department of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250
{mofta7,riley}@ece.gatech.edu

Abstract

Internet worm attacks have become increasingly more frequent and have had a major impact on the economy, making the detection and prevention of these attacks a top security concern. Several counter–measures have been proposed and evaluated in recent literature. However, the effect of these proposed defensive mechanisms on legitimate competing traffic has not been analyzed. Clearly, a defensive approach that slows down or stops worm propagation at the expense of completely restricting any legitimate traffic is of little value. Here we perform a comparative analysis of the effectiveness of several of these proposed mechanisms, including a measure of their effect on normal web browsing activities. In addition, we introduce a new defensive approach that can easily be implemented on existing hosts, and which significantly reduces the rate of spread of worms using TCP connections to perform the infiltration. Our approach has no measurable effect on legitimate traffic.

1. Introduction

The first known worm was the Morris worm in 1988 [15]. Since then, the security threats and damaging effects of modern worms have increased dramatically. The Code Red [11] and Nimda [3] worms infected hundreds of thousands computers around the world, and in 2003 the SQL Slammer worm [10] infected more than 90% of the vulnerable hosts (75,000) in less than 10 minutes. It has become apparent that no human intervention can react on a timely enough basis to react to these types of attacks, and therefore automatic detection and prevention mechanisms are a necessity.

In order to design and implement detection mechanisms, it is important to understand the basic behavior and activities of a typical, modern–day worm [20].

There are three stages in the worm life-cycle:

1. Propagation: The worm is transfered to a certain host by exploiting some vulnerability
2. Activation: The worm starts to execute a set of commands to gain higher access to the compromised system
3. Infection: The worm starts looking for other hosts to infect, and replicates itself on those hosts.

There are several different types of worms, classified by how they find their targets:

1. Topological worms: These worms find information about new targets from data stored on an infected host. Many applications contain information about other hosts, and therefore give the worm a good basis for finding other potential victims. The Morris worm was a topological worm.
2. Passive worms: These worms do not actively seek potential victims. Instead they rely on user actions to spread elsewhere. There have been been many passive worms like, Gnuman [2] and the CRClean [7]. Gnuman operates by acting as a gnutella node which replies to all queries with copies of itself. If this copy is run the worm starts on the victim machine and repeats the process. CRClean was intended to remove the Code Red II from the machine. The CRClean worm waits for a Code Red II probe, when it detects an infection attempt it launches a counterattack removing Code Red II and installing itself on the machine. These worms spread without any scanning.

* This work is supported in part by NSF under contract numbers ANI-9977544, ANI-0136969, ANI-0240477, ECS-0225417, and DARPA under contract number N66002-00-1-8934.

Proceedings of the Third IEEE International Workshop on Information Assurance (IWIA'05)
0-7695-2317-X/05 $20.00 © 2005 **IEEE**

3. Scanning worms: These worms search for new targets by probing IP addresses across the Internet. Scanning can be *sequential* where the worm works through an address block using an ordered set of addresses, or *random* where the worm selects addresses in a random fashion.

Our focus in this study is the *Scanning* category of worms, such as Code Red, Nimda, and SQL–Slammer.

There are some optimizations to the random scanning method, such as those discussed in [19] and [17]. These include *Localized scanning* (Code Red II [14]), which chooses a random address from within the same class B or class A address space as the infected host with higher probability than other non–local addresses. Another optimization is *hit-list scanning*, where the attacker collects a list of known vulnerable hosts on the Internet before releasing the worm. The worm chooses victims from this list and assigns the newly infected host a subset of the list to continue the spread. A third optimization is *permutation scanning* where all the worm instances share a common pseudo random permutation of the IP address space. Using this approach, there is less chance that different worm instances will choose the same victim, thus leading to faster infection spread. In this way worms will not spend much time scanning the same host multiple times. In a permutation scan the already infected host responds differently than a potential target as a way of telling the worm that it is already infected. When the worm detects that it scanned an already infected machine it realizes that another worm already scanned this portion of address space so it chooses a new random starting point and proceeds from there, this way coordination is imposed on the worm and needless reinfections are removed.

A combination of hit-list and permutation scanning can create what is termed a *Warhol worm* [17], which is capable of attacking most vulnerable targets in less than 15 minutes.

A number of methods have been proposed to detect, react to, or prevent these worm attacks. Since almost all worms exploit some software coding error or design flaw to infect the hosts, the most effective method would be to eliminate these software errors. However, it is fairly clear that some large fraction of existing or new Internet hosts will always have some exploitable vulnerability, since not all users or system administrators are willing or able to install security patches as they become available. Further, new software releases almost always introduce a new set of exploits.

Another approach is through the use of so–called Intrusion Detection Systems (*IDS*) [1], which are either signature-based or anomaly-based. In signature-based systems, a firewall checks incoming packets against a database of known worm signatures and drops the packet if a match is found. In anomaly-based systems the normal behavior of hosts is monitored, and if a significant deviation in the hosts activities is detected then some defensive action is taken. The signature-based approach is only effective against known worms, and has no effect against a new worm until that worm is analyzed, and its signature extracted and added to the database. Since this is a a time-consuming activity, it is clear that such approaches have little hope in containing new worms. Thus, the anomaly detection method is the method of choice when faced with detection and prevention of unknown worm attacks.

In this paper we will study different algorithms for worm containment, and evaluate their effectiveness. There are several requirements for a successful worm containment algorithm.

1. Quick response: The ability to quickly detect worm activity and stop it before it infects a significant number of others.
2. Low False detections: The algorithm should not consider a healthy host infected (false positive), since such detections typically trigger some sort of filtering or reduction in capacity of that host.
3. Simplicity: The algorithm must be simple to implement and deploy, and not take excessive resources on routers or end–systems.

2. Worm and Network Modeling

For all of our experiments, we are using the detailed models of Internet worms found in the *Georgia Tech Network Simulator* (*GTNetS*) [13]. These models are discussed in [6], and include a number of parameters that specify the behavior of the worm, including:

- *Transport protocol*: The underlying transport protocol used by the worm, which can be either UDP or TCP. UDP worms do not wait for any acknowledgment from the target, while the TCP worm require a three–way handshake (*SYN/SYN–ACK/ACK* before it can send its payload.
- *Infection length*: The size of the exploitation data that the worm needs to send to a host in order to infect it.
- *Infection port*: The transport layer port that exhibits the security vulnerability that is to be exploited.
- *Target vector*: The algorithm used by the worm to determine the IP address of a new victim. This can be either uniform, local preference or sequential scanning.
- *Scan rate*: The rate at which UDP worms send infection packets.
- *Connections*: The number of simultaneous connections attempts used by TCP worms.

For the network topology model we use a ring of *Random Tree* topologies as discussed in [6]. Each random tree is characterized by the *depth* and *fanout* as in typical tree topologies. However, the random tree provides an additional random probability factor that decides whether a child node will be created or not. This leads to more realistic topologies with *holes* in the assigned IP address space. We also vary the bandwidth of the links in these trees.

3. Worm Detection Algorithms

In this section, we give a short overview of worm detection and prevention algorithms, and then discuss in detail the five different proposed algorithms that are compared here. *Moore et al.* [12] have studied the effectiveness of worm containment systems and divided them into 2 types: *Address blacklisting* and *content filtering*. The *Address blacklisting* approach detects the misbehavior of certain network addresses and blocks any connection attempts from them. The *Content filtering* approach identifies common features of worm network connections and then filters all connections that share these features. Of the methods we study in detail, the the Virus Throttle 3.1 and counter malice 3.2 are examples of the first type; Packet Matching 3.3, DAW 3.4 and TCP ACK 3.5 as examples of the second type.

Each of the proposed algorithms is discussed in detail below.

3.1. Virus Throttle

The *Virus Throttle* approach, proposed by Williamson [18] relies on the fact that worm scanning involves communicating with a large number of hosts simultaneously (or nearly so), in order to find a vulnerable host to infect. This behavior is assumed to be atypical of *normal* application activity, which tend to communicate with a limited number of hosts. The goal of the algorithm is to delay connection attempts that appear to be more than what the host would normally make in a certain period of time. The more aggressive the infection action is, the more delay its connection requests would experience.

3.1.1. Implementation Details The *Virus Throttle* approach has the following parameters:

- *WorkingSet*: The set of the IP Addresses of the machines that this host has connected with recently. This list has limited size; our implementation uses 5. Each entry in the working set has a time flag.
- *DelayQueue*: A queue used to store packets that are to be delayed by the algorithm.

The virus throttle approach inspects all outgoing packets from a host, searching for TCP *SYN* packets. When a SYN packet is detected, the following algorithm is run.

- `If this host is in blocked state`
  - `Drop the packet.`
- `else`
  - `Compare destination address with addresses in the working set`
  - `If destination address is in the working set`
    - `Allow the connection immediately`
  - `Else if working set is not full`
    - `Add destination address to the working set.`
    - `Allow the connection to proceed immediately.`
  - `else`
    - `Add the packet to the delay queue.`
    - `If delay queue size is more than 100`
      - `Set the state of this host to blocked state.`

The following method is called once every second.
`Process-Queue()`

- `If working set is full`
  - `Remove oldest member.`
- `If delay queue is not empty`
  - `Pop the SYN packet from its head and any other packets addressed to the same destination.`
  - `Send the packet(s).`
  - `Add the destination address of the packet to the working set.`

3.2. CounterMalice

The *CounterMalice* [16] approach was developed by Silicon Defense and is conceptually similar to the virus throttle approach, except that it is intended to operate on a network device, such as a router, rather than on an end host. Counter malice works by monitoring the packets sent by a given host, and building a composite score of misbehavior based both on the number of unique destinations and the number of those that haven't responded.

3.2.1. Implementation Details We based our implementation of the counter malice approach on the information published in [16]. The published work lacks complete details on the working of the algorithm, but does provide sufficient information to make an approximation of the approach. In our implementation, We have an entry for each host in the subnetwork containing all the parameters mentioned in the virus throttle approach. When a host within a subnetwork sends a SYN packet to a host outside the subnetwork the following method is called.

`Output-packet-received()`

- `Does the source address represent a new entry ?`
  - `Create new entry`
- `Is the connection blocked ?`
  - `Drop packet.`
- `else`
  - `If destination address is in the working set`
    - `Allow the connection immediately.`
  - `Else if working set is not full`
    - `Add destination address to the working set.`
    - `Allow the connection immediately.`
  - `else`
    - `Add the packet to the delay queue.`
    - `If delay queue size is more than 100.`
      - `Set the state of that host to blocked.`

The following method is called once every second.

`Process-Queues()`

- `Loop through the list of host entries`
  - `If working set is full`
    - `Remove oldest member.`
  - `If delay queue is not empty`
    - `Pop the SYN packet from its head and any other packets addressed to the same destination.`
    - `Send the packet(s).`
    - `Add the destination address of the packet to the working set.`

3.3. Packet Matching

Packet Matching algorithm was proposed by Xuan Chen and John Heidemann [5]. This algorithm relies on the fact that a worm usually exploits some particular security vulnerability corresponding to a specific port number. Further, the nature of worms is that an infected host will probe other vulnerable hosts with the same vulnerability. Therefore routers seeing unusually high levels of bi-directional probing traffic with the same destination port number can infer a new worm attack is underway.

3.3.1. Implementation Details The algorithm operates on 2 steps; *port matching* and *address checking*. In the port matching step the algorithm compares the list of destination ports observed for inbound traffic to the list of destination ports observed for outbound traffic. If a match is noted, then the port is flagged as suspicious. In the address checking step, suspicious ports are monitored to detect how many unique IP addresses are being contacted, and an exponentially weighted moving average is computed for the number of unique destination IP addresses seen. When the instantaneous number of unique destinations is much larger than the moving average, the port is flagged infected.

The authors also suggest using collaboration between routers to disseminate suspicious and infected port information. We did not model this extension in the algorithm in our simulations.

Parameters used in the packet matching algorithm are:

- *Outgoing port list*: Ports on remote hosts that local hosts send packets to.
- *Incoming port list*: Ports on the local hosts that are the destination port for received packets.
- *Suspicious Portlist*: Ports that are suspicious or infected.
- β: Average number of unique IPs contacted for a given port
- *N*: Instantaneous number of unique IPs contacted for a given port
- δ: Sensitivity parameter set to 3 in our implementation
- α: Weight for the moving average set to 0.125 in our implementation

When a local host within a subnetwork sends connection request packets to a remote host outside the subnetwork, the following method is called.

`Out-Syn-packet()`

- `If the destination port infected ?`
  - `Drop packet.`

- Else if the destination port suspicious ?
  - Add the destination IP to the list of Unique IPs associated with this port
- Else
  - Add the destination port to the outgoing port list.
  - Forward the packet.
  - If the destination port is in the incoming port list
    - Add the port to the suspicious ports list

When a remote host outside the local subnetwork sends connections request packets to a local host, the following method is called:

`In-Syn-packet()`

- If the destination port infected ?
  - Drop packet.
- Else if the destination port suspicious ?
  - Add the destination IP to the list of Unique IPs associated with this port
- else
  - Add the destination port to the incoming port list.
  - Forward the packet.
  - If the destination port is in the outgoing port list
    - Add the port to the suspicious ports list

The following method is called periodically.

`Check-Infection()`

- Loop through the list of suspicious ports
- If N $> \beta \times \delta$
  - Mark this port as infected
- Else
  - Update the moving average $\beta = \alpha \times \beta + (1 - \alpha) \times N$
  - N = 0

3.4. DAW

The Distributed Anti-Worm architecture (*DAW* [4]), has been proposed as a distributed solution, with ISP's deploying the algorithm on edge routers. This algorithm relies on the fact that the failure rate for a random scanning worm is much higher than that of a normal well–behaved host.

A connection fails if the destination host does not exist (an ICMP Host Unreachable or Network Unreachable packet is sent) or if the destination host does exist, but has no layer 4 protocol accepting connections on the destination port (an ICMP Port Unreachable packet is sent). In addition, a *TCP Reset* packet will be sent if the destination host and port are valid, but the receiving application detects malformed data and closes the connection. In the *DAW* algorithm, the failure rate is measured as the number of ICMP host, network, or port unreachable messages and TCP Resets per unit time. Clearly, this algorithm assumes that there is no filtering of ICMP or TCP reset packets by a firewall or gateway between the source and destination. The algorithm has 2 components, the DAW agent that is deployed on the edge routers and a management station that collects data from multiple agents. In our simulations, we only consider the actions the agents individually, and do not take collaboration between agents into account. The basic principle is that if the connection failure rate of a host exceeds a pre-configured threshold, the *DAW* agent will begin dropping some of the connection requests from that host in order to keep its failure rate under the threshold.

3.4.1. Implementation Details The Parameters used by the *DAW* algorithm are:

- *size*: Size of the token bucket.
- *tokens*: Initialized to the size.
- *c*: Failure counter.
- *f*: Failure rate.
- β: Weight for the moving average for the failure rate set to 0.2 in our implementation.
- *t*: Timestamp.
- λ: Failure rate threshold.

The following method is called every time an indication of a failed connection is received.

`Update-Failure-Rate-Record()`

- `tokens = tokens - 1`
- `c = c + 1`
- `If ( c is a multiple of 10 )`
  - `f' = 10 / (the current system clock - t)`
  - `If (c == 10)`

        * f = f'
    - Else
        * $f = \beta \times f + (1 - \beta) \times f'$
- t = the current system clock

Upon observation of a connection request from a host in the local subnetwork, the following method is called.

Basic-Rate-Limit()

- $\delta =$ the current system clock - time
- $tokens =$ min(tokens + $\delta \times \lambda$, size)
- time = the current system clock
- If (tokens $\geq$ 1)
    - Forward the request
- Else
    - Drop the request

3.5. *TCP–ACK*

In addition to the previously mentioned detection methods, we introduce a new method called *TCP–ACK*. The approach is simple, easy to deploy on a large scale, and takes practically no resources. Our approach requires modifications to the protocol stack for existing hosts connected to the Internet (ie. Windows, Linux, MAC-OSX, etc.), which can easily be accomplished using the security update mechanisms already in place for existing operating systems. With our modified protocol stack, any host receiving a TCP *SYN* packet for non–existent port will unconditionally send a *SYN–ACK* to the originator, indicating that the connection has been accepted. The originator will then begin sending data packets to the same destination port, which are silently dropped.

To see the rationale behind our *TCP–ACK* approach, consider the actions by a normal TCP worm without our approach in place. A TCP worm creates multiple threads (up to some fixed limit) that attempt connection requests to random hosts. Without *TCP–ACK*, a host that has no corresponding layer 4 protocol at the specified port will create an *ICMP port unreachable* message, and the connecting thread receives an indication that the connection has failed. Thus, in only one round–trip–time the worm has determined that the target host and port is not vulnerable, and is free to try another one.

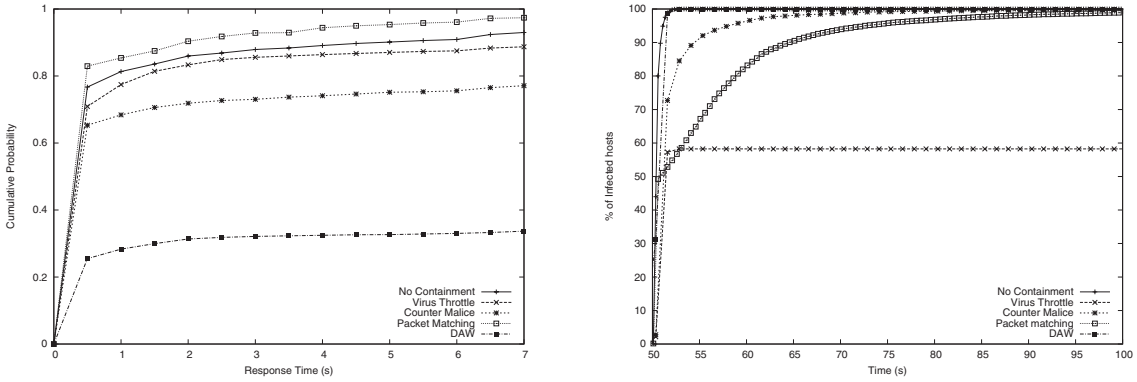
With *TCP–ACK*, the connection request to hosts that are not vulnerable (ie. those with no protocol bound to the destination port) will act as if they are. At that point, the worm will begin sending the payload packets which are silently dropped. From the perspective of the worm, this appears as normal lost packets, with the corresponding timeouts and retransmissions. Instead of one round–trip–time per failed connection, the worm is tied up for several re–transmission timeout periods which is substantially longer, potentially several minutes. The net result is a decrease in the effective probing rate of the worm, and a resulting decrease in the rate of spread. If a large fraction of hosts on the Internet implement the *TCP–ACK* mechanism, it will be nearly impossible for a TCP–style worm to effectively probe for vulnerabilities.

We point out that *LaBrea* has proposed an approach that is conceptually similar to ours. The *LaBrea* [8] method uses un-allocated IP address space to create the same trap. In this method, if the worm sends a SYN packet to an un-allocated address the LaBrea program would reply with a SYN ACK with a window size of zero trapping that thread. However, this approach requires substantial infrastructure enhancements at subnetworks in order to forward the un–mapped IP addresses to some host to create and send the *SYN–ACK*. Further, worms can easily detect the window size of zero and simply ignore any *SYN–ACK* with this signature. Our results also show that for any similar approach to work, the number of addresses with the trap installed must be more than the number of the vulnerable hosts. Which means in the case of *LaBrea* the ratio of unallocated addresses to real hosts in a subnetwork has to be greater than one which can not be used on a wide scale. In contrast, our approach can be easily implemented on a wide scale simply by including it as part of an operating system update.

4. Experimental Results

In this section, we describe the simulation experiments we used to measure the effectiveness of each of the previously discussed detection algorithms and defenses. In addition to measuring their effect on the overall worm spread rate, we also monitored the effect on *normal* web browsing activity.

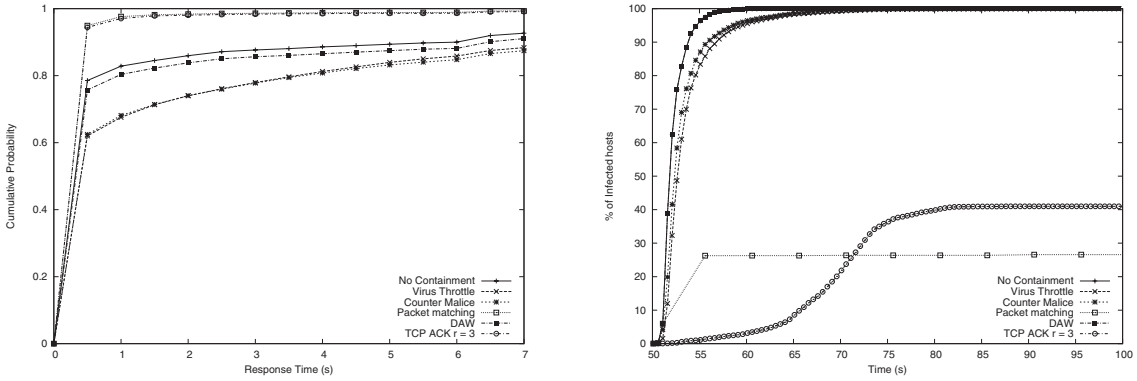
For the network topology model we created a topology consisting of about 9000 nodes using 11 *Random Tree* topologies connected with a ring. There are a mix of parameters for the random trees, as follows. We have 2 trees with fanout 8 and depth 5 allocating 4096 addresses each; 4 trees with fanout 4 and depth 5 allocating 256 addresses each; 4 trees with fanout 8 and depth 4 allocating 512 addresses each; and one tree with fanout 16 and depth 4 allocating 4096 addresses, for a total of 15,360 possible IP addresses. The tree is populated with child probability such that we have only about 60% (on average) of the possible 15,360 leaf nodes, or about an average of 9,000 leaf nodes in each simulation. Since the random scanning of IP addresses is an essential feature of all worms, and since the probability of a "correct guess" is a fundamental parameter in determining the worm's spreading rate, we have reduced the entire Internet IP address space down to the 15,360 pos-



(a) cdf of response times

(b) Worm spread

Figure 1. Effect of algorithms on the network for UDP worm



(a) cdf of response times

(b) Worm spread

Figure 2. Effect of algorithms on the network for TCP worm

sible addresses in our simulation. Thus the probability of guessing a "good" IP address is approximately 60%.

For the worm parameters, we set the infection length to 500, the infection port to 1040, the target vector to a uniform random generator spanning the defined address space. We have set the UDP worm to have a scan rate 100 probes per second, while the TCP worm is set to have 3 simultaneous connections.

We conducted experiments by simulating a worm outbreak on the "ring–of–trees" network described above, and measured the overall rate of spread for the worm. In addition, we monitored the web response time for normal web browsing actions. The web browser model is that defined by Mah [9], and is the default web browser model in *GTNetS*. The worm attack was not started in the simulations until time $t = 50$ seconds, to allow the web browsing traffic to get started and reach steady state.

Figure 1a shows the effect of the different implementations of the discussed algorithms on the web browsing response times when a UDP worm attacks the network. Figure 1b shows their effect on the *worm spread rate*. Figures 2a,b shows the same results but for a TCP worm.

Since some of the defenses look specifically for the *TCP–SYN* packet as an indication, we modified those algorithms to treat a UDP packet as an infection attempt (since we had no "normal" UDP traffic in these scenarios this is a

reasonable approach).

4.1. Virus Throttle

The figures show that the throttle is capable of stopping the UDP worm spread in less than three seconds for this small network. However, in this environment, nearly 60% of vulnerable hosts are infected in that same time period. Moreover, this approach has a significant impact on the normal web browsing activity, increasing considerably the average web response time.

In the case of the TCP worm, the virus throttle approach fails to detect it or to cause any significant reduction in its infection spread. Further, we still notice a considerable reduction in the performance of browsers in the network. This is due to the fact that the throttle is slowing down infected hosts (all hosts in this case) both for the infection packets and normal web browser connections, but the slow down is not significant enough to block the worm spread significantly.

Some of the shortcomings in the practical implementation of this approach are:

1. It is host based, so there is a risk of the worm to actually attack the algorithm and stop it from executing.
2. This approach is not effective against slow spreading worms that are below the threshold of detection
3. Deployment must be complete in order to attain good results, which means that deployment cost is going to be high.
4. Complete blocking of the infected host would result in blocking non worm traffic from that host as well.

4.2. CounterMalice

In our experiments for this approach, we placed the counter malice algorithm on each of the first level routers in the random trees (the children of the root of each tree). Thus, each counter malice process has a variable number of existing and non–existing hosts in the tree below it.

From the performance figures, we can see that the performance is worse than the virus throttle approach both for TCP and UDP worms. This is primarily due to the fact that the counter malice algorithm can not detect infections within a subnet, since the detection is on the gateway to other networks. Further, once the counter malice algorithm begins blocking actions, it has a significant degradation on normal web browsing.

However, this approach does address some of the shortcomings of the throttle approach. Since it is not host based we do not need to deploy it on every host but rather just on the routers. Also it can detect slow spreading worms as it does take into account the number of hosts that do not respond to a connection request, but it has the additional shortcoming of being unable to detect worm spread within a subnetwork. Unfortunately, it still has the same detrimental effect on the normal web traffic as the virus throttle approach.

4.3. Packet Matching

In our experiments for this approach, we placed the packet matching algorithm on each of the first level routers in the random trees (the children of the root of each tree).

This approach is better than the previous two in the fact that it blocks only certain port access rather than all traffic for suspected host. This means that the infected host can still have its normal traffic go through without any delays while the worm traffic is blocked or delayed.

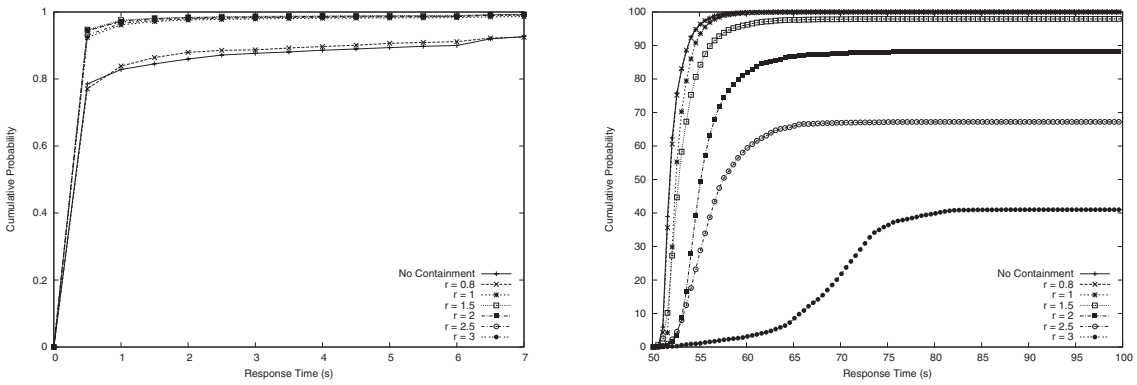
Figure 2a shows that, for the TCP worm, the packet matching algorithm was able to stop the worm infections very quickly and Figure 2 shows that the performance of the web browsing activities improved by 20%, due to the suppression of worm traffic.

Figure 1a shows that, for a UDP worm, the packet matching algorithm was *not* able to stop the worm completely and the infections continued to spread until it reached 100%, This is due to the dynamic threshold used in this approach, which calculates a moving average. This means that if the worm traffic was increasing slowly for a subnetwork such that the average would also increase slowly, the worm would go undetected and would be considered as normal traffic. We would expect a fixed threshold to perform better, but this would require a customized threshold for each port, depending on the application running on that port and the expected level of activity on that port. Another note is that if the infected port had been the same as the web browser port (port 80) the detrimental effects would be extremely severe, since this approach not only blocks the infected hosts, but the entire subnetwork containing the infected host. This approach also has a similar drawback to the counter malice approach in that it cannot detect or affect infections within a subnetwork.

4.4. DAW

In our experiments for this approach, we placed the DAW algorithm on each of the first level routers in the random trees (the children of the root of each tree).

The figures show that the DAW approach, with just the *Basic-Rate-Limit* method applied, has almost no effect on worm spread in either the TCP or UDP worm cases. However, it has significant impact on web response time, resulting in more than 50% of all requests failing to complete in seven seconds or less in the case of the UDP worm, and 20% failing in the TCP worm case. This approach also has



(a) cdf of response times

(b) Worm spread

Figure 3. Effect of *TCP–ACK* on the network for TCP worm

the shortcoming of not being able to detect or react to infections within a subnetwork.

The authors also provided two additional methods to be used for limiting the connection requests, called the *Temporal Rate-Limit Algorithm* and the *Spatial rate-Limit Algorithm*. In the first one they also take into account the number of failed connections during an entire day, as they state that a normal user may generate high failure rate in a short period of time but that should not continue for 24 hours. However, an infected host would have a high failure rate all the time. Thus, they define another parameter Γ that represents the threshold for failed connections in a day.

The second method takes into account the number of failed connections of the network as a whole, using a collaborative methods between the *DAW* processes.

We modeled the first method but instead of a period of a day we defined for a period of one minute and we set Γ to be equal to 30, meaning that we allow 30 failed connections in one minute. The results are not shown here, but did not show any major improvement to the *BasicRateLimit* method presented.

We did not model the second method as we are not taking into account collaborative efforts between agents and the central station.

4.5. *TCP–ACK*

In our new *TCP–ACK* approach, we define r as the ratio between the number of nodes that do not have an application associated with the worm infection port to the number of nodes that have the vulnerable application. We further assume that all systems have the required kernel patch to send the *SYN–ACK* in response to connection requests to non–existent ports.

Figure 2a shows that for the value of $r = 3$, the TCP worm is blocked completely. Further, Figure 2b shows that the performance of the web browsers was improved, since our approach does nothing to packets addressed to legitimate hosts and ports, and our approach caused suppression of the worm traffic.

We also ran experiments on the network by varying the ratio r, The Figure 3 shows that for small values of r (0.8, 1) this method has no noticeable effect but by increasing r the effectiveness of the TCP ACK algorithm increases until it is able to stop the worm completely for $r = 3$ without any negative effect on the normal web traffic.

We point out that in our model the worm does not provide its own timeout period for the hung connections to fail. We expect that experienced worm developers will become aware of this defensive method and will provide some timeout period to terminate the connection. Regardless, the timeout period must be much longer than the single round–trip–time connection failure in present worms, and thus will still reduce the overall rate of spread for TCP–style worms.

5. Conclusion

We have performed a detailed simulation–based study of the effectiveness of several proposed worm detection and defensive methods, and have quantified the effect of these methods on normal web–browsing activities. Further, we introduced a new defensive mechanism we call *TCP–ACK*, which is shown to be effective against worms using TCP connections for payload propagation.

Acknowledgment

We would like to thank David Dagon and Wenke Lee for their insightful comments and feedback.

References

[1] Intrusion detection and prevention: protecting your network from attacks.

[2] W32/gnuman.worm.

[3] E. J. Aronne. The nimda worm: An overview. Technical report, SANS, October 2001.

[4] S. Chen and Y. Tang. Slowing down internet worms. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, March 2004.

[5] X. Chen and J. Heidemann. Detecting early worm propagation through packet matching. Technical Report ISI-TR-2004-585, USC/Information Sciences Institute, February 2004.

[6] M. I. S. George F. Riley and W. Lee. Simulating internet worms. In *Proceedings of The IEEE Computer Societys 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)*, pages 268–274, Volendam, The Netherlands, October 2004.

[7] M. Kern. Re: Codegreen beta release.

[8] T. Liston. Labrea project, 2001.

[9] B. A. Mah. An empirical model of http network traffic. In *Proceedings of IEEE INFOCOMM*, pages 592–600, 1997.

[10] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Magazine of Security and privacy*, 1(4):33–39, July/August 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 Internet Measurement Workshop (IMW)*, Marseille, France, November 2002.

[12] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings IEEE INFOCOM*, San Francisco, CA, USA, March 2003.

[13] G. F. Riley. The Georgia Tech Network Simulator. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 5–12. ACM Press, 2003.

[14] R. Russell and A. Mackie. Code red worm ii analysis report. Technical report, SecurityFocus, August 2001.

[15] D. Seeley. A tour of the worm. In *Proceedings of the winter USENIX Conference*, pages 287–304, San Diego, CA, USA, January 1989.

[16] S. Staniford. Containment of scanning worms in enterprise networks. *Journal of Computer Security*, to appear 2004.

[17] V. P. Stuart Staniford and N. Weaver. How to 0wn the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, USA, August 2002.

[18] J. Twycross and M. M. Williamson. Implementing and testing a virus throttle. In *Proceedings of the 12th USENIX Security Symposium*, pages 285–294, Washington, D.C., USA, August 2003.

[19] N. Weaver. Potential strategies for high speed active worms: A worst case analysis. http://www.cs.berkeley.edu/∼nweaver/ worms.pdf, March 2002.

[20] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy of computer worms. In *Proceedings of the 2003 ACM CCS workshop on Rapid Malcode (WORM'03)*, pages 11–18, Washington, DC, USA, 2003.