# Very Fast Containment of Scanning Worms

Nicholas Weaver
*ICSI*
*nweaver@icsi.berkeley.edu*

Stuart Staniford
*Nevis Networks*
*stuart@nevisnetworks.com*

Vern Paxson
*ICSI & LBNL*
*vern@icir.org*

## Abstract

Computer worms — malicious, self-propagating programs — represent a significant threat to large networks. One possible defense, *containment*, seeks to limit a worm's spread by isolating it in a small subsection of the network. In this work we develop containment algorithms suitable for deployment in high-speed, low-cost network hardware. We show that these techniques can stop a scanning host after fewer than 10 scans with a very low false-positive rate. We also augment this approach by devising mechanisms for *cooperation* that enable multiple containment devices to more effectively detect and respond to an emerging infection. Finally, we discuss ways that a worm can attempt to bypass containment techniques in general, and ours in particular.

## 1 Introduction

Computer worms — malicious, self propagating programs — represent a substantial threat to large networks. Since these threats can propagate more rapidly than human response [24, 12], automated defenses are critical for detecting and responding to infections [13]. One of the key defenses against scanning worms which spread throughout an enterprise is *containment* [28, 23, 21, 7, 14]. Worm containment, also known as virus throttling, works by detecting that a worm is operating in the network and then blocking the infected machines from contacting further hosts. Currently, such containment mechanisms only work against *scanning* worms [27] because they leverage the anomaly of a local host attempting to connect to multiple other hosts as the means of detecting an infectee.

Within an enterprise, containment operates by breaking the network into many small pieces, or *cells*. Within each cell (which might encompass just a single machine), a worm can spread unimpeded. But between cells, containment attempts to limit further infections by blocking outgoing connections from infected cells.

A key problem in containment of scanning worms is efficiently detecting and suppressing the scanning. Since containment *blocks* suspicious machines, it is critical that the false positive rate be very low. Additionally, since a successful infection could potentially subvert any software protections put on the host machine, containment is best effected inside the network rather than on the end-hosts.

We have developed a scan detection and suppression algorithm based on a simplification of the Threshold Random Walk (TRW) scan detector [9]. The simplifications make our algorithm suitable for both hardware and software implementation. We use caches to (imperfectly) track the activity of both addresses and individual connections, and reduce the random walk calculation of TRW to a simple comparison. Our algorithm's approximations generally only cost us a somewhat increased false negative rate; we find that false positives do not increase.

Evaluating the algorithm on traces from a large (6,000 host) enterprise, we find that with a total memory usage of 5 MB we obtain good detection precision while staying within a processing budget of at most 4 memory accesses (to two independent banks) per packet. In addition, our algorithm can detect scanning which occurs at a threshold of one scan per minute, much lower than that used by the throttling scheme in [28], and thus significantly harder for an attacker to evade.

Our trace-based analysis shows that the algorithms are both highly effective and sensitive when monitoring scanning on an Internet access link, able to detect low-rate TCP and UDP scanners which probe our enterprise. One deficiency of our work, however, is that we were unable to obtain internal enterprise traces. These can be very difficult to acquire, but we are currently pursuing doing so. Until we can, the efficacy of our algorithm when deployed internal to an enterprise can only be partly inferred from its robust access-link performance.

We have also investigated how to enhance containment through *cooperation* between containment devices. Worm containment systems have an *epidemic threshold*: if the number of vulnerable machines is few enough relative to a particular containment deployment, then containment will almost completely stop the worm [21]. However, if there

are more vulnerable machines, then the worm will still spread exponentially (though less than in the absence of containment). We show that by adding a simple inter-cell communication scheme, the spread of the worm can be dramatically mitigated in the case where the system is above its epidemic threshold.

Finally, we discuss inadvertent and malicious attacks on worm containment systems: what is necessary for an attacker to create either false negatives (a worm which evades detection) or false positives (triggering a response when a worm did not exist), assessing this for general worm containment, cooperative containment, and our particular proposed system. We specifically designed our system to resist some of these attacks.

## 2 Worm Containment

Worm containment is designed to halt the spread of a worm in an enterprise by detecting infected machines and preventing them from contacting further systems. Current approaches to containment [28, 21, 19] are based on detecting the scanning activity associated with scanning worms, as is our new algorithm.

Scanning worms operate by picking "random" addresses and attempting to infect them. The actual selection technique can vary considerably, from linear scanning of an address space (Blaster [25]), fully random (Code Red [6]), a bias toward local addresses (Code Red II [4] and Nimda [3]), or even more enhanced techniques (Permutation Scanning [24]). While future worms could alter their style of scanning to try to avoid detection, all scanning worms share two common properties: most scanning attempts result in failure, and infected machines will institute many connection attempts.[1] Because containment looks for a class of behavior rather than specific worm signatures, such systems can stop *new* (scanning) worms.

Robust worm defense requires an approach like containment because we know from experience that worms can find (by brute force) small holes in firewalls [4], VPN tunnels from other institutions , infected notebook computers [25], web browser vulnerabilities [3], and email-borne attacks [3] to establish a foothold in a target institution. Many institutions with solid firewalls have still succumbed to worms that entered through such means. Without containment, even a single breach can lead to a complete internal infection.

Along with the epidemic threshold (Section 2.1) and sustained sub-threshold scanning (Section 2.2), a significant issue with containment is the need for complete deployment within an enterprise. Otherwise, any uncontained-but-infected machines will be able to scan through the enterprise and infect other systems. (A single machine, scanning at only 10 IP addresses per second, can scan through an entire $/16$ in under 2 hours.)

Thus, we strongly believe that worm-suppression needs to be built into the network fabric. When a worm compromises a machine, the worm can defeat host software designed to limit the infection; indeed, it is already common practice for viruses and mail-worms to disable antivirus software, so we must assume that future worms will disable worm-suppression software.

Additionally, since containment works best when the cells are small, this strongly suggests that worm containment needs to be integrated into the network's outer switches or similar hardware elements, as proximate to the end hosts as economically feasible. This becomes even more important for cooperative containment (Section 6), as this mechanism is based on some cells becoming compromised as a means of better detecting the spread of a worm and calibrating the response necessary to stop it.

### 2.1 Epidemic Threshold

A worm-suppression device must necessarily allow some scanning before it triggers a response. During this time, the worm may find one or more potential victims. Staniford [21] discusses the importance of this "epidemic threshold" to the worm containment problem. If on average an infected computer can find more than a single victim before a containment device halts the worm instance, the worm will still grow exponentially within the institution (until the average replication rate falls below 1.0).

The epidemic threshold depends on

- the sensitivity of the containment response devices
- the density of vulnerable machines on the network
- the degree to which the worm is able to target its efforts into the correct network, and even into the current cell

Aside from cooperation between devices, the other options to raise the epidemic threshold are to increase the sensitivity of the scan detector/suppressor, reduce the density of vulnerable machines by distributing potential targets in a larger address space, or increase the number of cells in the containment deployment.

One easy way to distribute targets across a larger address space arises if the enterprise's systems use NAT and DHCP. If so, then when systems acquire an address through DHCP, the DHCP server can select a random address from within a private /8 subnet (e.g., 10.0.0.0/8). Thus, an institution with $2^{16}$ workstations could have an internal vulnerability density of $2^{16}/2^{24} = 1/256$, giving plenty of headroom for relatively insensitive worm-suppression techniques to successfully operate.

[1] There are classes of worms—topological, meta-server, flash (during their spreading phase, once the hit-list has been constructed), and contagion [27]—that do *not* exhibit such scanning behavior. Containment for such worms remains an important, open research problem.

Alternatively, we can work to make the worm detection algorithm more accurate. The epidemic threshold is directly proportional to the scan threshold $T$: the faster we can detect and block a scan, the more vulnerabilities there can be on the network without a worm being able to get loose. Thus, we desire highly sensitive scan-detection algorithms for use in worm containment.

## 2.2 Sustained Scanning Threshold

In addition to the epidemic threshold, many (but not all) worm containment techniques also have a *sustained scanning threshold*: if a worm scans slower than this rate, the detector will not trigger. Although there have been systems proposed to detect very stealthy scanning [22], these systems are currently too resource-intensive for use in this application.

Even a fairly low sustained scanning threshold can enable a worm to spread if the attacker engineers the worm to avoid detection. For example, consider the spread of a worm in an enterprise with $256$ ($2^8$) vulnerable machines distributed uniformly in a contiguous $/16$ address space. If the worm picks random addresses from the entire Internet address space, then we expect only $1$ in $2^{24}$ scans to find another victim in the enterprise. Thus, even with a very permissive sustained scanning threshold, the worm will not effectively spread within the enterprise.

But if the worm biases its scanning such that $1/2$ the effort is used to scan the local $/16$, then on average it will locate another target within the enterprise after $2^9$ scans. If the threshold is one scan per second (the default for Williamson's technique [28]), then the initial population's doubling time will be approximately $2^9$ seconds, or once every 8.5 minutes. This doubling time is sufficient for a fast-moving worm, as the entire enterprise will be infected in less than two hours. If the worm concentrates its entire scanning within the enterprise's /16, the doubling time will be about four minutes.

Thus, it is vital to achieve as low a sustained scanning threshold as possible. For our concrete design, we target a threshold of 1 scan per minute. This would change the doubling times for our example above to 8.5 and 4 hours respectively — slow enough that humans can notice the problem developing and take additional action. Achieving such a threshold is a much stricter requirement than that proposed by Williamson, and forces us to develop a different scan-detection algorithm.

# 3 Scan Suppression

The key component for today's containment techniques is *scan suppression*: responding to detected *portscans* by blocking future scanning attempts. Portscans—probe attempts to determine if a service is operating at a target IP address—are used by both human attackers and worms to discover new victims. Portscans have two basic types: *horizontal* scans, which search for an identical service on a large number of machines, and *vertical* scans, which examine an individual machine to discover all running services. (Clearly, an attacker can also combine these and scan many services on many machines. For ease of exposition, though, we will consider the two types separately.)

The goal of scan suppression is often expressed in terms of preventing scans coming from "outside" inbound to the "inside." If "outside" is defined as the external Internet, scan suppression can thwart naive attackers. But it can't prevent infection from external worms because during the early portion of a worm outbreak an inbound-scan detector may only observe a few (perhaps only single) scans from any individual source. Thus, unless the suppression device halts all new activity on the target port (potentially disastrous in terms of collateral damage), it will be unable to decide, based on a single request from a previously unseen source, whether that request is benign or an infection attempt.

For worm *containment*, however, we turn the scan suppressor around: "inside" becomes the enterprise's larger internal network, to be protected from the "outside" local area network. Now any scanning worm will be quickly detected and stopped, because (nearly) *all* of the infectee's traffic will be seen by the detector.

We derived our scan detection algorithm from TRW (Threshold Random Walk) scan detection [9]. In abstract terms, the algorithm operates by using an oracle to determine if a connection will fail or succeed. A successfully completed connection drives a random walk upwards, a failure to connect drives it downwards. By modeling the benign traffic as having a different (higher) probability of success than attack traffic, TRW can then make a decision regarding the likelihood that a particular series of connection attempts from a given host reflect benign or attack activity, based on how far the random walk deviates above or below the origin. By casting the problem in a Bayesian random walk framework, TRW can provide deviation thresholds that correspond to specific false positive and false negative rates, if we can parameterize it with good *a priori* probabilities for the rate of benign and attacker connection successes.

To implement TRW, we obviously can't rely on having a connection oracle handy, but must instead track connection establishment. Furthermore, we must do so using data structures amenable to high-speed hardware implementation, which constrains us considerably. Finally, TRW has one added degree of complexity not mentioned above. It only considers the success or failure of connection attempts to *new* addresses. If a source repeatedly contacts the same host, TRW does its random walk accounting and decision-making only for the first attempt. This approach inevitably requires a very large amount of state to keep track of which pairs of addresses have already tried

to connect, too costly for our goal of a line-rate hardware implementation. As developed in Section 5, our technique uses a number of approximations of TRW's exact bookkeeping, yet still achieves quite good results.

There are two significant alternate scan detection mechanisms proposed for worm containment. The first is the new-destination metric proposed by Williamson [28]. This measures the number of new destinations a host can visit in a given period of time, usually set to 1 per second. The second is dark-address detection, used by both Forescout [7] and Mirage Networks [14]. In these detectors, the device routes or knows some otherwise unoccupied address spaces within the internal network and detects when systems attempt to contact these unused addresses.

# 4 Hardware Implementations

When targeting hardware, memory access speed, memory size, and the number of distinct memory banks become critical design constraints, and, as mentioned above, these requirements drive us to use data structures that sometimes only approximate the network's state rather than exactly tracking it. In this section we discuss these constraints and some of our design choices to accommodate them. The next section then develops a scan detection algorithm based on using these approximations.

Memory access speed is a surprisingly significant constraint. During transmission of a minimum-sized gigabit Ethernet packet, we only have time to access a DRAM at 8 different locations. If we aim to monitor both directions of the link (gigabit Ethernet is full duplex), our budget drops to 4 accesses. The situation is accordingly even worse for 10-gigabit networks: DRAM is no longer an option at all, and we must use much more expensive SRAM. If an implementation wishes to monitor several links in parallel, this further increases the demand on the memory as the number of packets increases.

One partial solution for dealing with the tight DRAM access budget is the use of independent memory banks allowing us to access two distinct tables simultaneously. Each bank, however, adds to the overall cost of the system. Accordingly, we formulated a design goal of no more than 4 memory accesses per packet to 2 separate tables, with each table only requiring two accesses: a read and a write to the same location.

Memory size can also be a limiting factor. For the near future, SRAMs will only be able to hold a few tens of megabytes, compared with the gigabits we can store in DRAMs. Thus, our ideal memory footprint is to stay under 16 MB. This leaves open the option of implementing using only SRAM, and thus potentially running at 10 gigabit speeds.

Additionally, software implementations can also benefit from using the approximations we develop rather than exact algorithms. Since our final algorithm indeed meets our design goals—less than 16 MB of total memory (it is highly effective with just 5 MB) and 2 uncached memory accesses per packet—it could be included as a scan detector within a conventional network IDS such as Bro [16] or Snort [20], replacing or augmenting their current detection facilities.

## 4.1 Approximate Caches

When designing hardware, we often must store information in a fixed volume of memory. Since the information we'd *like* to store may exceed this volume, one approach is to use an *approximate cache*: a cache for which collisions cause imperfections. (From this perspective, a Bloom filter is a type of approximation cache [2].) This is quite different from the more conventional notion of a cache for which, if we find an entry in the cache, we know exactly what it means, but a failed lookup requires accessing a large secondary data-store, or of a hash table, for which we will always find what we put in it earlier, but it may grow beyond bound. Along with keeping the memory bounded, approximate caches allow for very simple lookups, a significant advantage when designing hardware.

However, we then must deal with the fact that collisions in approximate caches can have complicated semantics. Whenever two elements map to the same location in the cache, we must decide how to react. One option is to combine distinct entries into a single element. Another is to discard either the old entry or the new entry. Accordingly, collisions, or *aliasing*, create two additional security complications: false positives or negatives due to the policy when entries are combined or evicted, and the possibility of an attacker manipulating the cache to exploit these aliasing-related false outcomes.

Since the goal of our scan-suppression algorithm is to generate automatic responses, we consider false positives more severe than false negatives, since they will cause an instance of useful traffic to be completely impaired, degrading overall network reliability. A false negative, on the other hand, often only means that it takes us longer to detect a scanner (unless the false negative is systemic). In addition, if we can structure the system such that several positives or negatives must occur before we make a response decision, then the effect will be mitigated if they are not fully correlated.

Thus, we decided to structure our cache-based approximations to avoid creating additional false positives. We can accomplish this by ensuring that, when removing entries or combining information, the resulting combination could only create a false negative, as discussed below.

Attackers can exploit false negatives or positives by either using them to create worms that evade detection, or by triggering responses to impair legitimate traffic. Attacker can do so through two mechanisms: predicting the hashing algorithm, or simply overwhelming the cache.

The first attack, equivalent to the algorithm complexity attacks described by Crosby and Wallach [5], relies on the

attacker using knowledge of the cache's hash function to generate collisions. For Crosby's attack, the result was to increase the length of hash chains, but for an approximation cache, the analogous result is a spate of evicted or combined entries, resulting in excess false positives or negatives. A defense against it is to use a keyed hash function whose output the attacker cannot predict without knowing the key.

The second attack involves flooding the cache in order to hide a true attack by overwhelming the system's ability to track enough network activity. This could be accomplished by generating a massive amount of "normal" activity to cloak malicious behavior. Unlike the first attack, overwhelming the cache may require substantial resources.

While such attacks are a definite concern (see also Section 7), approximate caching is vital for a high-performance hardware implementation. Fortunately, as shown below, we are able to still obtain good detection results even given the approximations.

### 4.2 Efficient Small Block Ciphers

Another component in our design is the use of small (32 bit) block ciphers. An $N$-bit block cipher is equivalent to an $N$-bit keyed permutation: there exists a one-to-one mapping between every input word and every output word, and changing the key changes the permutation.

In general, large caches are either direct-mapped, where any value can only map to one possible location, or $N$-way associative. Looking up an element in a direct-mapped cache requires computing the index for the element and checking if it resides at that index. In an associative cache, there are $N$ possible locations for any particular entry, arranged in a contiguous block (cache line). Each entry in an associative cache includes a tag value. To find an element, we compute the index and then in parallel check all possible locations based on the tag value to determine if the element is present.

Block ciphers give us a way to implement efficiently tagged caches that resist attackers predicting their collision patterns. They work by, rather than using the initial $N$-bit value to generate the cache index and tag values, first permuting the $N$-bit value, after which we separate the resulting $N$-bit value into an index and a tag. If we use $k$ bits for the index, we only need $N - k$ bits for the tag, which can result in substantial memory savings for larger caches. If the block-cipher is well constructed and the key is kept secret from the attacker, this will generate cache indices that attackers cannot predict. This approach is often superior to using a hash function, as although a good hash function will also provide an attacker-unpredictable index, the entire $N$-bit initial value will be needed as a tag.

Ciphers that work well in software are often inefficient in hardware, and vice versa. For our design, we used a simple 32 bit cipher based on the Serpent S-boxes [1], particularly well-suited for FPGA or ASIC implementation as it requires only 8 levels of logic to compute.

## 5 Approximate Scan Suppression

Our scan detection and suppression algorithm approximates the TRW algorithm in a number of ways. First, we track connections and addresses using approximate caches. Second, to save state, rather than only incorporating the success or failure of connection attempts to new addresses, we do so for attempts to new addresses, new ports at old addresses, and old ports at old addresses if the corresponding entry in our state table has timed out. Third, we do not ever make a decision that an address is benign; we track addresses indefinitely as long as we do not have to evict their state from our caches.

We also extend TRW's principles to allow us to detect vertical as well as horizontal TCP scans, and also horizontal UDP scans, while TRW only detects horizontal TCP scans. Finally, we need to implement a "hygiene filter" to thwart some stealthy scanning techniques without causing undue restrictions on normal machines.

Figure 1 gives the overall structure of the data structures. We track connections using a fixed-sized table indexed by hashing the "inside" IP address, the "outside" IP address, and, for TCP, the inside port number. Each record consists of a 6 bit age counter and a bit for each direction (inside to outside and outside to inside), recording whether we have seen a packet in that direction. This table combines entries in the case of aliasing, which means we may consider communication to have been bidirectional when in fact it was unidirectional, turning a failed connection attempt into a success (and, thus, biasing towards false negatives rather than false positives).

We track external ("outside") addresses using an associative approximation cache. To find an entry, we encrypt the external IP address using a 32 bit block cipher as discussed in Section 4.2, separating the resulting 32 bit number into an index and a tag, and using the index to find the group (or line) of entries. In our design, we use a 4-way associative cache, and thus each line can contain up to four entries, with each entry consisting of the tag and a counter. The counter tracks the difference between misses and hits (i.e., successful and unsuccessful connection attempts), forming the basis of our detection algorithm.

Whenever the device receives a packet, it looks up the corresponding connection in the connection table and the corresponding external address in the address table. Per Figure 2, the status of these two tables, and the direction of the packet, determines the action to take, as follows:

For a non-blocked external address (one we have not already decided to suppress), if a corresponding connection has already been established in the packet's direction, we reduce the connection table's age to 0 and forward the packet. Otherwise, if the packet is from the outside and

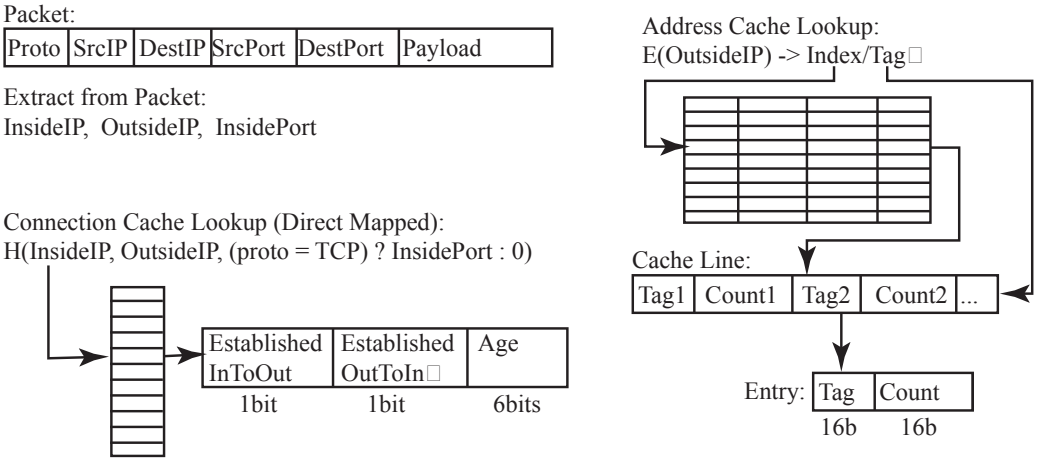


Figure 1: The structure of the connection cache and the address cache. The connection cache tracks whether a connection has been established in either direction. The age value is reset to 0 every time we see a packet for that connection. Every minute, a background process increases the age of all entries in the connection cache, removing any idle entry more than $D_{conn}$ minutes old. The address cache keeps track of all detected addresses, and records in "count" the difference between the number of failed and successful connections. Every $D_{miss}$ seconds, each positive count in the address cache is reduced by one.

Condition:
SrcIP = InsideIP

```
If(!EstablishedInToOut)
   if(EstablishedOutToIn)
      # Was previously
      # recorded as a miss
      # but is now a hit
      Count <- Count - 2
   EstablishedInToOut <- True
Age <- 0
Forward packet
```

Condition:
SrcIP = OutsideIP &
Count < Threshhold

```
If(!EstablishedOutToIn)
   if(EstablishedInToOut)
      # Record as a hit
      Count <- Count - 1
      EstablishedOutToIn <- True
   else if(hygiene_drop)
      Drop packet
   else
      # A possible miss
      Count <- Count + 1
      EstablishedOutToIn <- True
if(!DroppedPacket)
   Age <- 0
   Forward packet
```

Condition:
SrcIP = OutsideIP &
Count >= Threshhold

```
# Address is being blocked
if(EstablishedInToOut)
   if(isSYN | isUDP)
      # No matter what, drop
      Drop packet
   else if(!EstablishedOutToIn){
      # Record as a hit
      Count <- Count - 1
      EstablishedOutToIn <- True
   # Internally requested or old
   # connection, so allow
   Age <- 0
   Forward packet
else
   Drop packet
```

Figure 2: The high level structure of the detection and response algorithm. We count every successful connection (in either direction) as a "hit", with all failed or possibly-failed connections as "misses". If the difference between the number of hits and misses is greater than a threshold, we block further communication attempts from that address.

we have seen a corresponding connection request from the inside, we forward the packet and decrement the address's count in the address table by 1, as we now credit the outside address with a successful connection. Otherwise, we forward the packet but increment the external address's count by 1, as now that address has one more outstanding, so-far-unacknowledged connection request.

Likewise, for packets from internal addresses, if there is a connection establishment from the other direction, the count is reduced, in this case by 2, since we are changing our bookkeeping of it from a failure to a success (we previously incremented the failure-success count by 1 because we initially treat a connection attempt as a failure).

Thus, the count gives us an on-going estimate of the difference between the number of misses (failed connections) and the number of successful connections. Given the assumption that legitimate traffic succeeds in its connection attempts with a probability greater than 50%, while scanning traffic succeeds with a probability less than 50%, by monitoring this difference we can determine when it is highly probable that a machine is scanning.

## 5.1 Blocking and Special Cases

If an address's count exceeds a predefined threshold $T$, the device blocks it. When we receive subsequent packet from that address, our action depends on the packet's type and whether it matches an existing, successfully-established connection, which we can tell from the connection status bits stored in the connection table. If the packet does not match an existing connection, we drop it. If it does, then we still drop it if it is a UDP packet or a TCP initial SYN. Otherwise, we allow it through. By blocking in this manner, we prevent the blocked machine from establishing subsequent TCP or UDP sessions, while still allowing it to *accept* TCP connection requests and continue with existing connections. Doing so lessens the collateral damage caused by false positives.

We treat TCP RST, RST+ACK, SYN+ACK, FIN, and FIN+ACK packets specially. If they do not correspond to a connection established in the other direction, the hygiene filter simply drops these packets, as they could reflect stealthy scanning attempts, backscatter from spoofed-source flooding attacks, or the closing of very-long-idle connections. Since they might be scans, we need to drop them to limit an attacker's information. But since they might instead be benign activity, we don't use them to trigger blocks.

Likewise, if a connection has been established in the other direction, but not in the current direction, then we forward TCP RST, RST+ACK, FIN, and FIN+ACK packets, but do not change the external address's counter, to avoid counting failed connections as successful. (A FIN+ACK could reflect a successful connection *if* we have seen the connection already established in the current direction, but the actions here are those we take if we have not seen this.)

## 5.2 Errors and Aliasing

Because connection table combines entries when aliasing occurs, it can create a false negative at a rate that depends on the fullness of the table. If the table is 20% full, then we will fail to detect roughly 20% of individual scanning attempts. Likewise, 20% of the successful connection attempts will not serve to reduce an address's failure/success count either, because the evidence of the successful connection establishment aliases with a connection table entry that already indicates a successful establishment.

To prevent the connection table from being overwhelmed by old entries, we remove any connection idle for more than an amount of time $D_{conn}$, which to make our design concrete we set to $D_{conn} = 10$ minutes. We can't reclaim table space by just looking for termination (FIN exchanges) because aliasing may mean we need to still keep the table entry after one of the aliased connections terminates, and because UDP protocols don't have a clear "terminate connection" message.

While the connection table combines entries, the address table, since it is responsible for blocking connections and contains tagged data, needs to evict entries rather than combining information. Yet evicting important data can cause false negatives, requiring a balancing act in the eviction policy. We observe that standard cache replacement policies such as least recently used (LRU), round robin, and random, can evict addresses of high interest. Instead, when we need to evict an entry, we want to select the entry with the most negative value for the (miss–hit) count, as this constitutes the entry least likely to reflect a scanner; although we thus tend to evict highly active addresses from the table, they represent highly active *normal* machines.

In principle, this policy could occasionally create a transient false positive, if subsequent connections from the targeted address occur in a very short term burst, with several connection attempts made before the first requests can be acknowledged. We did not, however, observe this phenomenon in our testing.

## 5.3 Parameters and Tuning

There are several key parameters to tune with our system, including the response threshold $T$ (miss–hit difference that we take to mean a scan detection), minimum and maximum counts, and decay rates for the connection cache and for the counts. We also need to size the caches.

For $T$, our observations below in Section 5.5 indicate that for the traces we assessed a threshold of 5 suffices for blocking inbound scanning, while a threshold of 10 is a suitable starting point for worm containment.

The second parameters, $C_{min}$ and $C_{max}$, are the minimum and maximum values the count is allowed to achieve. $C_{min}$ is needed to prevent a previously good address that is subsequently infected from being allowed too many connections before it is blocked, while $C_{max}$ limits how long

it takes before a highly-offending blocked machine is allowed to communicate again. For testing purposes, we set $C_{min}$ to $-20$, and $C_{max}$ to $\infty$ as we were interested in the maximum count which each address could reach in practice.

The third parameter, $D_{miss}$, is the decay rate for misses. Every $D_{miss}$ seconds, all addresses with positive counts have their count reduced by one. Doing so allows a low rate of benign misses to be forgiven, without seriously enabling sub-threshold scanning. We set $D_{miss}$ equal to 60 seconds, or one minute, meeting our sub-threshold scanning goal of 1 scan per minute. In the future, we wish to experiment with a much lower decay rate for misses.

We use a related decay rate, $D_{conn}$, to remove idle connections, since we can't rely on a "connection-closed" message to determine when to remove entries. As mentioned earlier, we set $D_{conn}$ to 10 minutes.

The final parameters specify the size and associativity of the caches. A software implementation can tune these parameters, but a hardware system will need to fix these based on available resources. For evaluation purposes, we assumed a 1 million entry connection cache (which would require 1 MB), and a 1 million entry, 4-way associative address cache (4 MB). Both cache sizes worked well with our traces, although increasing the connection cache to 4 MB would provide increased sensitivity by diminishing aliasing.

## 5.4 Policy Options

Several policy options and variations arise when using our system operationally: the threshold of response, whether to disallow all communication from blocked addresses, whether to treat all ports as the same or to allow some level of benign scanning on less-important ports, and whether to detect horizontal and vertical, or just horizontal, TCP scans.

The desired initial response threshold $T$ may vary from site to site. Since all machines above a threshold of 6 in our traces represent some sort of scanner (some benign, most malicious, per Section 5.5), this indicates a threshold of 10 on outbound connections would be conservative for deployment within our environment, while a threshold of 5 appears sufficient for incoming connections.

A second policy decision is whether to block all communication from a blocked machine, or to only limit new connections it initiates. The first option offers a greater degree of protection, while the second is less disruptive for false positives.

A third decision is how to configure $C_{min}$ and $C_{max}$, the floor and ceiling on the counter value. We discussed the tradeoffs for these in the previous section.

A fourth policy option would be to treat some ports differently than others. Some applications, such as Gnutella [17], use scanning to find other servers. Likewise, at some sites particular tools may probe numerous machines to discover network topology. One way to give different ports different weights would be to changing the counter from an integer to a fixed-point value. For example, we could assign SNMP a cost of .25 rather than 1, to allow a greater degree of unidirectional SNMP attempts before triggering an alarm. We can also weight misses and hits differently, to alter the proportion of traffic we expect to be successful for benign vs. malicious sources.

Finally, changing the system to only detect horizontal TCP scans requires changing the inputs to the connection cache's hash function. By excluding the internal port number from the hash function, we will include all internal ports in the same bucket. Although this prevents the algorithm from detecting vertical scans, it also eliminates an evasion technique discussed in Section 7.6.

## 5.5 Evaluation

We used hour-long traces of packet header collected at the access link at the Lawrence Berkeley National Laboratory. This gigabit/sec link connects the Laboratory's 6,000 hosts to the Internet. The link sustains an average of about 50–100 Mbps and 8–15K packets/sec over the course of a day, which includes roughly 20M externally-initiated connection attempts (most reflecting ambient scanning from worms and other automated malware) and roughly 2M internally-initiated connections. The main trace we analyzed was 72 minutes long, beginning at 1:56PM on a Friday afternoon. It totaled 44M packets and included traffic from 48,052 external addresses (and all 131K internal addresses, due to some energetic scans covering the entire internal address space). We captured the trace using `tcpdump`, which reported 2,200 packets dropped by the measurement process.

We do not have access to the ideal traces for assessing our system, which would be all internal and external traffic for a major enterprise. However, the access-link traces at least give us a chance to evaluate the detection algorithm's behavior over high-diverse, high-volume traffic.

We processed the traces using a custom Java application so we could include a significant degree of instrumentation, including cache-miss behavior, recording evicted machines, maintaining maximum and minimum counts, and other options not necessary for a production system. Additionally, since we developed the experimental framework for off-line analysis, high performance was not a requirement. Our goal was to extract the necessary information to determine how our conceptual hardware design will perform in terms of false positives and negatives and quickness of response.

For our algorithm, we just recorded the maximum count rather than simulating a specific blocking threshold, so we can explore the tradeoffs different thresholds would yield. We emulated a 1 million entry connection cache, and a 1 million entry, 4-way associative address cache. The connection cache reached 20% full during the primary trace.

| Anonymized IP | Maximum Count | Cause |
|---|---|---|
| 221.147.96.4 | 16 | Benign DNS Scanner? Dynamic DNS host error? |
| 147.95.58.73 | 12 | AFS-Related Control Traffic? |
| 147.95.35.149 | 12 | NetBIOS "Scanning" and activity |
| 147.95.238.71 | 8 | AFS-Related Control Traffic? |
| 144.240.17.50 | 6 | Benign SNMP (UDP) "Scanning" |
| 144.240.96.234 | 6 | NetBIOS "scanning" of a few hosts |

Table 1: All outbound connections over a threshold of 5 flagged by our algorithm

The eviction rate in the address cache was very low, with no evictions when tested with the Internet as "outside," and only 2 evictions when the enterprise was "outside." Thus, the 5 MB of storage for the two tables was quite adequate.

We first ran our algorithm with the enterprise as outside, to determine which of its hosts would be blocked by worm containment and why. We manually checked all alerts that would be generated for a threshold of 5, shown in Table 1. Of these, all represented benign scanning or unidirectional control traffic. The greatest offender, at a count of 16, appears to be a misconfigured client which resulted in benign DNS scanning. The other sources appears to generate AFS-related control traffic on UDP ports 7000-7003; scanning from a component of Microsoft NetBIOS file sharing; and benign SNMP (UDP-based) scanning, apparently for remotely monitoring printer queues.

With the Internet as "outside," over 470 external addresses reached a threshold of 5 or higher. While this seems incredibly high, it in fact represents the endemic scanning which occurs continually on the Internet [9]. We manually examined the top 5 offenders, whose counts ranged from 26,000 to 49,000, and verified that these were all blatant scanners. Of these, one was scanning for the FTP control port (21/tcp), two were apparently scanning for a newly discovered vulnerability in Dameware Remote Administrator (6129/tcp), and two were apparently scanning for a Windows RPC vulnerability (135/tcp; probably from hosts infected with Blaster [25]).

Additionally, we examined the offenders with the lowest counts above the threshold. 10 addresses had a maximum count between 20 and 32. Of these, 8 were scans on a NetBIOS UDP port 137, targeted at a short (20–40 address) sequential range, with a single packet sent to each machine. Of the remaining two offenders, one probed randomly selected machines in a /16 for a response on TCP port 80 using 3 SYN packets per attempt, while the other probed randomly selected machines on port 445/tcp with 2 SYN packets per attempt. All of these offenders represented true scanners: none is a false positive.

We observed 19 addresses with a count between 5 and 19, where we would particularly expect to see false positives showing up. Of these, 15 were NetBIOS UDP scanners. Of the remaining 4, one was scanning 1484/udp, one was scanning 80/tcp, and one was scanning 445/tcp. The final entry was scanning both 138/udp and generating successful communications on 139/tcp and port 80/tcp. The final entry, which reached a maximum count of 6, represents a NetBIOS-related false positive.

Finally, we also examined ten randomly selected external addresses flagged by our algorithm. Eight were UDP scanners targeting port 137, while two were TCP scanners targeting port 445. All represent true positives.

During this test, the connection cache size of 1 million entries reached about 20% full. Thus, each new scan attempt has a 20% chance of not being recorded because it aliases with an already-established connection. If the connection cache was increased to 4 million entries (4 MB instead of 1 MB), the false negative rate would drop to slightly over 5%.

We conducted a second test to determine the effects of setting the parameters for maximum sensitivity. We increased the connection cache to 4 million entries, reducing the number of false negatives due to aliasing. We also tightened the $C_{min}$ threshold to -5, which increases the sensitivity to possible misbehavior of previously "good" machines, and increased $D_{miss}$ to infinity, meaning that we never decayed misses. Setting the threshold of response to 5 would then trigger an alert for an otherwise idle machine once it made a series of 5 failed connections; while a series of 10 failed connections would trigger an alert regardless of an address's past behavior.

We manually examined all outbound alerts (i.e., alerts generated when considering the enterprise "outside") that would have triggered when using this threshold, looking for additional false positives. Table 2 summarizes these additional alerts.

We would expect that, by increasing the sensitivity in this manner, we would observe some non-scanning false positives. Of the additional alerts, only one new alert was generated because of the changed $C_{min}$. This machine sent out unidirectional UDP to 15 destinations in a row, which was countered by normal behavior when $C_{min}$ was set to -20 instead of -5. The rest of the alerts were triggered because of the reduced decay of misses. In all these cases, the traffic consisted of unidirectional communication to multiple machines. The TCP-based activity (NNTP, daytime,

| Anonymized IP | Maximum Count | Cause |
|---|---|---|
| 147.95.61.87 | 11 | NNTP, sustained low rate of failures |
| 147.95.35.154 | 11 | High port UDP, 10 scans in a row |
| 221.147.96.220 | 9 | TCP port 13 (“daytime”), detected due to reduced sub-threshold |
| 144.240.96.234 | 9 | NetBIOS and failed HTTP, detected due to reduced sub-threshold |
| 144.240.28.138 | 7 | High port UDP, due to reduced sub-threshold |
| 147.95.3.27 | 6 | TCP Port 25, due to reduced sub-threshold |
| 147.95.36.165 | 5 | High port UDP, due to reduced sub-threshold |
| 144.240.43.227 | 5 | High port UDP, due to reduced sub-threshold |

Table 2: Additional alerts on the outbound traffic generated when the sensitivity was increased.

and SMTP) showed definite failed connections, but these may be benign failures.

In summary, even with the aggressive thresholds, there are few false positives, and they appear to reflect quite peculiar traffic.

### 5.6 Williamson Implementation

For comparison purposes, we also included in our trace analysis program an implementation of Williamson's technique [28], which we evaluated against the site's outbound traffic in order to assess its performance in terms of worm containment. Williamson's algorithm uses a small cache of previously-allowed destinations. For all SYNs and any UDP packets, if we find the destination in the allowed-destination cache, we forward it regularly. If not, but if the source has not sent to a new destination (i.e., we haven't added anything to its allowed-destination cache) during the past second, then we put an entry in the cache to note that we are allowing communication between the source and the given destination, and again forward the packet.

Otherwise, we add the packet to a delay queue. We process this queue at the rate of one destination per second. Each second, for each source we determine the next destination it attempted to send to but so far has not due to our delay queue. We then forward the source's packets for that destination residing in the delay queue and add the destination to the allowed-destination cache. The effect of this mechanism is to limit sources to contacting a single new destination each second. One metric of interest with this algorithm then is the maximum size the delay queue reaches.

A possible negative consequence of the Williamson algorithm is that the cache of previously established destinations introduces false positives rather than false negatives. Due to its limited size, previously established destinations may be evicted prematurely. For testing purposes, we selected cache sizes of 8 previously-allowed destinations per source (3 greater than the cache size used in [28]). We manually examined all internal sources where the delay queue reached 15 seconds or larger, enough to produce a significant disturbance for a user (Table 3).

In practice, we observed that the Williamson algorithm has a very low false positive rate, with only a few minor exceptions. First, the DNS servers in the trace greatly overflow the delay queue due to their high fanout when resolving recursive queries, and thus would need to be special-cased. Likewise, a major SMTP server also triggered a response due to its high connection fanout, and would also require white-listing. However, of potential note is that three HTTP clients reached a threshold greater than 15, which would produce a user-noticeable delay but not trigger a permanent alarm, based on Williamson's threshold of blocking machines when their delay queue reaches a depth of 100 [26].

## 6 Cooperation

Staniford analyzed the efficacy of worm containment in an enterprise context, finding that such systems exhibit a phase structure with an epidemic threshold [21]. For sufficiently low vulnerability densities and/or $T$ thresholds, the system can almost completely contain a worm. However, if these parameters are too large, a worm can escape and infect a sizeable fraction of the vulnerable hosts despite the presence of the containment system. The epidemic threshold occurs when on average a worm instance is able to infect exactly one child before being contained. Less than this, and the worm will peter out. More, and the worm will spread exponentially. Thus we desire to set the response threshold $T$ as low as possible, but if we set it too low, we may incur unacceptably many false positives. This tends to place a limit on the maximum vulnerability density that a worm containment system can handle.

In this section, we present a preliminary analysis of performance improvements that come from incorporating communication between cells. The improvement arises by using a second form of a-worm-is-spreading detector: the alerts generated by other containment devices. The idea is that every containment device knows how many blocks the other containment devices currently have in effect. Each devise uses this information to dynamically adjust its response threshold: as more systems are being

| Anonymized IP | Delay Queue Size | Cause |
|---|---|---|
| 144.240.84.131 | 11,395 | DNS Server |
| 147.95.15.21 | 8,772 | DNS Server |
| 144.240.84.130 | 3,416 | DNS Server |
| 147.95.3.37 | 23 | SMTP Server |
| 144.240.25.76 | 19 | Bursty DNS Client |
| 147.95.52.12 | 18 | Active HTTP Client |
| 147.95.208.255 | 17 | Active HTTP Client |
| 147.95.208.18 | 15 | Active HTTP Client |

Table 3: All outbound connections with a delay queue of size 15 or greater for Williamson's algorithm

blocked throughout the enterprise, the individual containment devices become more sensitive. This positive feedback allows the system to adaptively respond to a spreading worm.

The rules for doing so are relatively simple. All cells communicate, and when one cell blocks an address, it communicates this status to the other cells. Consequently, at any given time each cell can compute that $X$ other blocks are in place, and thereby reduces $T$ by $(1-\theta)^X$, where $\theta$ is a parameter that controls how aggressively to reduce the threshold as a worm spreads. For our algorithm, the cell also needs to increase $C_{min}$ by a similar amount, to limit the scanning allowed by a previously normal machine.

In our simulations, very small values of $\theta$ make a significant difference in performance. This is good, since reducing the threshold also tends to increase false positive rates.[2]

However, we can have the threshold return to its normal (initial) value using an exponentially weighted time delay to ensure that this effect is short lived.

A related policy question is whether this function should allow a complete shutdown of the network (no new connections tolerated), or should have a minimum threshold below which the containment devices simply will not go, potentially allowing a worm to still operate at a slower spreading rate, depending on its epidemic threshold. The basic tradeoff is ensuring a degree of continued operation, vs. a stronger assure that we will limit possible damage from the worm.

[2]Large values of $\theta$ risk introducing catastrophic failure modes in which some initial false positive drives thresholds low enough to create more false positives, which drive thresholds still lower. This could lead to a complete blockage of traffic due to a runaway positive feedback loop. This is unlikely with the small values of $\theta$ in this study, and moreover could be addressed by introducing a separate threshold for communication that was *not* adaptively modified. The two thresholds would begin at the same value, but the blocking threshold would lower as the worm spread, while the communication threshold — i.e., the degree of scanning required before a device *tells* other devices that it has blocked the corresponding address — would stay fixed. This would sharply limit the positive feedback of more false positives triggering ever more changes to the threshold.

## 6.1 Testing Cooperation

To evaluate the effects of cooperation, we started with the simulation program used in the previous evaluation of containment [21]. We modified the simulator so that each response would reduce the threshold by $\theta$. We then reran some of the simulations examined in [21] to assess the effect on the epidemic threshold for various values of $\theta$.

The particular set of parameters we experimented with involved an enterprise network of size $2^{17}$ addresses. We assumed a worm that had a $50\%$ probability of scanning inside the network, with the rest falling outside the enterprise. We also assumed an initial threshold of $T = 10$, that the network was divided into $512$ cells of $256$ addresses each, and that the worm had no special preference to scan within its cell. We considered a uniform vulnerability density. These choices correspond to Figure 2 in [21], and, as shown there, the epidemic threshold is then at a vulnerability density of $v = 0.2$ (that is, it occurs when $20\%$ of the addresses are vulnerable to the worm).

We varied the vulnerability density across this epidemic threshold for different values of $\theta$, and studied the resulting average infection density (the proportion of vulnerable machines which actually got infected). This is shown in Figure 3, where each point represents the average of 5,000 simulated worm runs. The top curve shows the behavior when communication does not modify the threshold (i.e., $\theta = 0$), and successively lower curves have $\theta = 0.00003$, $\theta = 0.0001$, and $\theta = 0.0003$. It is to be emphasized that these are tiny values of $\theta$ (less than $3/100$ of $1\%$). One would not expect there to be any significant problem of increased false positives with such small changes; but that they are larger than zero suffices to introduce significant positive feedback in the presence of a propagating worm (i.e., the overall rate of blocked scans within the network rises over time).

The basic structure of the results is clear. Changing $\theta$ does not significantly change the epidemic threshold, but we can greatly reduce the infection density that the worm can achieve above the epidemic threshold. It makes sense that the epidemic threshold is not changed, since below the epidemic threshold, the worm cannot gain much traction

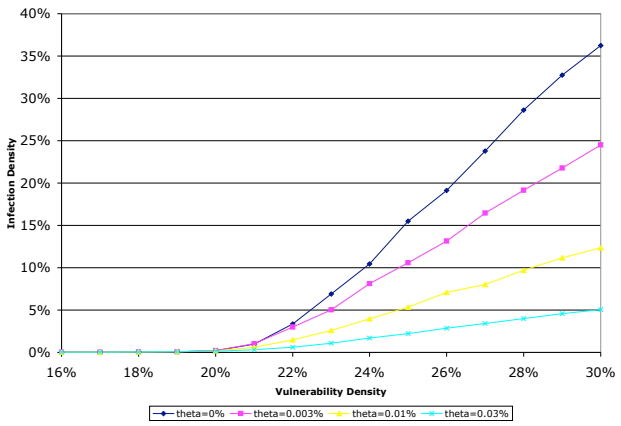


Figure 3: Plot of worm infection density against vulnerability density $v$ for varying values of the threshold modification value $\theta$. See the text for more details.

and so the algorithm that modifies $T$ has no chance to engage and alter the situation. However, above the epidemic threshold, adaptively changing $T$ can greatly reduce the infection density a worm can achieve. Clearly, inter-cell communication mechanisms hold great promise at improving the performance of containment systems.[3]

We must however discuss a simplification we made in our simulation. We effectively assumed that communication amongst cells occurs instantaneously compared to the worm propagation. Clearly, this an idealization. A careless design of the communication mechanism could result in speeds that cause the threshold modification to always substantially lag behind the propagation of the worm, greatly limiting its usefulness. (See [15] for a discussion of the competing dynamics of a response to a worm and the worm itself).

For example, it can be shown that a design in which we send a separate packet to each cell that needs notification allows worm instances to scan (on average) a number of addresses equal to half the number of cells before any threshold modification occurs (assuming that the worm can scan at the same speed as the communication mechanism can send notifications). This isn't very satisfactory.

One simple approach to achieve very fast inter-cell communication is to use broadcast across the entire network. However, this is likely to pose practical risks to network performance in the case where there are significant numbers of false positives.

A potentially better approach is for the containment devices to cache recently contacted addresses. Then when a source IP crosses the threshold for scan detection, the cells it recently communicated with can be contacted first (in order). These cells will be the ones most in need of the information. In most cases, this will result in threshold modification occurring before the threshold is reached on any cells that got infected as a result (rather than the message arriving too late and the old unmodified threshold being used).

[3]Particularly in parts of the parameter space where the epidemic threshold vulnerability density is much lower than 20% — e.g., if the worm has the ability to differentially target its own cell.

## 7 Attacking Worm Containment

Security devices do not exist in a vacuum, but represent both targets and obstacles for possible attackers. By creating a false positive, an attacker can trigger responses which wouldn't otherwise occur. Since worm containment *must* restrict network traffic, false positives create an attractive DOS target. Likewise, false negatives allow a worm or attacker to slip by the defenses.

General containment can incur inadvertent false positives both from detection artifacts and from "benign" scanning. Additionally, attackers can generate false positives if they can forge packets, or attempt to evade containment if they detect it in operation. When we also use cooperation, an attacker who controls machines in several cells can cause significant network disruption through cooperative collapse: using the network of compromised machines to trigger an institution-wide response by driving down the thresholds used by the containment devices through the institute (if $\theta$ is large enough to allow this). Our scan detection algorithm also has an algorithm-specific, two-sided evasion, though we can counter these evasions with some policy changes, which we discuss below. Although we endeavor in this section to examine the full range of possible attacks, undoubtedly there are more attacks we haven't considered.

### 7.1 Inadvertent False Positives

There are two classes of inadvertent false positives: false positives resulting from artifacts of the detection routines, and false positives arising from "benign" scanning. The first are potentially the more severe, as these can severely limit the use of containment devices, while the second is often amenable to white-listing and other policy-based techniques.

In our primary testing trace, we observed only one instance of an artifact-induced false positive, due to unidirectional AFS control traffic. Thus, this does not appear to be a significant problem for our algorithm. Our implementation of Williamson's mechanism showed artifact-induced false positives involving 3 HTTP clients that would have only created a minor disruption. Also, Williamson's algorithm is specifically not designed to apply to traffic generated by servers, requiring these machines to be white-listed.

Alerting on benign scanning is less severe. Indeed, such scans should trigger all good scan-detection devices. More generally, "benign" is fundamentally a policy distinction: is this particular instance of scanning a legitimate activity, or something to prohibit?

We have observed benign scanning behavior from Windows File Sharing (NetBIOS) and applications such as Gnutella which work through a list of previously-connected peers to find access into a peer-to-peer overlay. We note that if these protocols were modified to use a rendezvous point or a meta-server then we could eliminate their scanning behavior. The other alternative is to whitelist these services. By whitelisting, their scanning behavior won't trigger a response, but the containment devices can no longer halt a worm targeting these services.

## 7.2 Detecting Worm Containment

If a worm is propagating within an enterprise that has a containment system operating, then the worm could slow to a sub-threshold scanning rate to avoid being suppressed. But in the absence of a containment system, the worm should instead scan quickly. Thus, attackers will want to devise ways for a worm to detect the presence of a containment system.

Assuming that the worm instance knows the address of the host that infected it, and was told by it of a few other active copies of the worm in the enterprise, then the worm instance can attempt to establish a normal communication channel with the other copies. If each instance sets up these channels, together they can form a large distributed network, allowing the worm to learn of all other active instances.

Having established the network, the worm instance then begins sending out probes at a low rate, using its worm peers as a testing ground: if it can't establish communication with already-infected hosts, then it is likely the enterprise has a containment system operating. This information can be discovered even when the block halts all direct communication: the infection can send a message into the worm's overlay network, informing the destination worm that it will attempt to probe it. If the ensuing direct probe is blocked, the *receiving* copy now knows that the sender is blocked, as it was informed about the experimental attempt.

This information can then be spread via the still-functional connections among the worm peers in order to inform future infections in the enterprise. Likewise, if the containment system's blocks are only transient, the worm can learn this fact, and its instances can remain silent, waiting for blocks to lift, before resuming sub-threshold scanning.

Thus we must assume that a sophisticated worm can determine that a network employs containment, and probably deduce both the algorithm and parameters used in the deployment.

## 7.3 Malicious False Negatives

Malicious false negatives occur when a worm is able to scan in spite of active scan-containment. The easiest evasion is for the worm to simply not scan, but propagate via a different means: topological, meta-server, passive, and target-list (hit-list) worms all use non-scanning techniques [27]. Containing such worms is outside the scope of our work. We note, however, that scanning worms represent the largest class of worms seen to date and, more generally, a broad class of attack. Thus, eliminating scanning worms from a network clearly has a great deal of utility even if it does not address the entire problem space.

In addition, scanning worms that operate below the sustained-scanning threshold can avoid detection. Doing so requires more sophisticated scanning strategies, as the worms must bias their "random" target selection to effectively exploit the internal network in order to take advantage of the low rate of allowed scanning. The best countermeasure for this evasion technique is simply a far more sensitive threshold. We argue that a threshold of 1 scan per second (as in Williamson [28]), although effective for stopping current worms, is too permissive when a worm is attempting to evade containment. Thus we targeted a threshold of 1 scan per minute in our work.

Additionally, if scanning of some particular ports has been white-listed (such as Gnutella, discussed above), a worm could use that port to scan for liveness—i.e., whether a particular address has a host running on it, even though the host rejects the attempted connection—and then use followup scans to determine if the machine is actually vulnerable to the target service. While imperfect—failed connection attempts will still occur—the worm can at least drive the failure rate lower because the attempts will fail less often.

Another substantial evasion technique can occur if a corrupted system can obtain multiple network addresses. If a machine can gain $k$ distinct addresses, then it can issue $k$ times as many scans before being detected and blocked. This has the effect of reducing the epidemic threshold by a factor of $k$, a huge enhancement on a worm's ability to evade containment.

## 7.4 Malicious False Positives

If attackers can forge packets, they can frame other hosts in the same cell as scanners. We can engineer a local area network to resist such attacks by using the MAC address and switch features that prevent spoofing and changing of MAC addresses. This is not an option, though, for purported scans inbound to the enterprise coming from the external Internet. While the attacker can use this attack to deny service to external addresses, preventing them from initiating new connections to the enterprise, at least they can't block new connections initiated by internal hosts.

There is an external mechanism which could cause this

internal DOS: a malicious web page or HTML-formatted email message could direct an internal client to attempt a slew of requests to nonexistent servers. Since this represents an attacker gaining a limited degree of control over the target machine (i.e., making it execute actions on the attacker's behalf), we look to block the attack using other types of techniques, such as imposing HTTP proxies and mail filtering to detect and block the malicious content.

## 7.5 Attacking Cooperation

Although cooperation helps defenders, an attacker can still attempt to outrace containment if the initial threshold is highly permissive. However, this is unlikely to occur simply because the amount of communication is very low, so it is limited by network latency rather than bandwidth. Additionally, broadcast packets could allow quick, efficient communication between all of the devices. Nevertheless, this suggests that the communication path should be optimized.

The attacker could also attempt to flood the containment coordination channels before beginning its spread. Thus, containment-devices should have reserved communication bandwidth, such as a dedicated LAN or prioritized VLAN channels, to prevent an attacker from disrupting the inter-cell communication.

Of greater concern is *cooperative collapse*. If the rate of false positives is high enough, the containment devices respond by lowering their thresholds, which can generate a cascade of false positives, which further reduces the threshold. Thus, it is possible that a few initial false positives, combined with a highly-sensitive response function, could trigger a maximal network-wide response, with major collateral damage.

An attacker that controls enough of the cells could attempt to trigger or amplify this effect by generating scanning in those cells. From the viewpoint of the worm containment, this appears to reflect a rapidly spreading worm, forcing a system-wide response. Thus, although cooperation appears highly desirable due to the degree to which it allows us to begin the system with a high tolerance setting (minimizing false positives), we need to develop models of containment cooperation that enable us to understand any potential exposure an enterprise has to the risk of maliciously induced cooperative collapse.

## 7.6 Attacking Our Algorithm

Our approximation algorithm adds two other risks: attackers exploiting the approximation caches' hash and permutation functions, and vulnerability to a two-sided evasion technique. We discussed attacking the hash functions earlier, which we address by using a block-cipher based hash. In the event of a delayed response due to a false negative, the attacker will have difficulty determining which possible entry resulted in a collision.

Another evasion is for the attacker to embed their scanning within a large storm of spoofed packets which cause thrashing in the address cache and which pollute the connection cache with a large number of half-open connections. Given the level of resources required to construct such an attack (hundreds of thousands or millions of forged packets), however, the attacker could probably spread just as well simply using a slow, distributed scan. Determining the tradeoffs between cache size and where it becomes more profitable to perform distributed scanning is an area for future work.

A more severe false negative is a two-sided evasion: two machines, one on each side of the containment device, generate normal traffic establishing connections on a multitude of ports. A worm could use this evasion technique to balance out the worm's scanning, making up for each failed scanning attempt by creating another successful connection between the two cooperating machines. Since our algorithm treats connections to distinct TCP ports as distinct attempts, two machines can generate enough successes to mask any amount of TCP scanning.

There is a counter-countermeasure available, however. Rather than attempting to detect both vertical and horizontal TCP scanning, we can modify the algorithm to detect only horizontal scans by excluding port information from the connection-cache tuple. This change prevents the algorithm from detecting vertical scans, but greatly limits the evasion potential, as now any pair of attacker-controlled machines can only create a single success.

More generally, however, for an Internet-wide worm infection, the huge number of external infections could allow the worm to generate a large amount of successful traffic even when we restrict the detector to only look for horizontal scans. We can counter this technique, though, by splitting the detector's per-address count into one count associated with scanning within the internal network and a second count to detect scanning on the Internet. By keeping these counts separate, an attacker could use this evasion technique to allow Internet scanning, but they could not exploit it to scan the internal network. Since our goal is to protect enterprise and not the Internet in the large, this is acceptable.

A final option is to use two containment implementations, operating simultaneously, one targeting scans across the Internet and the other only horizontal scans within the enterprise. This requires twice the resources, although any hardware can be parallelized, and allows detection of both general scanning and scanning behavior designed to evade containment.

# 8 Related Work

In addition to the TRW algorithm used as a starting point for our work [9], a number of other algorithms to detect scanning have appeared in the literature.

Both the Network Security Monitor [8] and Snort [20] attempt to detect scanning by monitoring for systems which exceed a count of unique destination addresses contacted during a given interval. Both systems can exhibit false positives due to active, normal behavior, and may also have a significant scanning sub-threshold which an attacker can exploit.

Bro [16] records failed connections on ports of interest and triggers after a user-configurable number of failures. Robinson *et al.* [18] used a similar method.

Leckie *et al* [11] use a probabilistic model based on attempting to learn the probabilistic structure of normal network behavior. The model assumes that access to addresses made by scanners follows a uniform distribution rather than the non-homogeneous distribution learned for normal traffic, and attempts to classify possible scanning sources based on the degree to which one distribution or the other better fits their activity.

Finally, Staniford *et al*'s work on SPICE [22] detects very stealthy scans by correlating anomalous events. Although effective, it requires too much computation to use it for line-rate detection on high-speed networks.

In addition to Williamson's [28, 26] and Staniford's [21, 23] work on worm containment, Jung et al [10] have developed a similar containment technique based on TRW. Rather than using an online algorithm which assumes that all connections fail until proven successful, it uses the slightly delayed (until response seen or timeout) TRW combined with a mechanism to limit new connections similar to Williamson's algorithm.

Zou *et al.* [30] model some requirements for dynamic-quarantine defenses. They also demonstrate that, with a fixed threshold of detection and response, there are epidemic thresholds. Additionally, Moore *et al.* have studied abstract requirements for containment of worms on the Internet [13], and Nojiri *et al* have studied the competing spread of a worm and a not-specifically-modeled response to it [15].

There have been two other systems attempting to commercialize scan containment: Mirage networks [14] and Forescout [7]. Rather than directly detecting scanning, these systems intercept communication to unallocated (dark) addresses and respond by blocking the infected systems.

## 9 Future Work

We have plans for future work in several areas: implementing the system in hardware and deploying it; integrating the algorithm into a software-based IDS; attempting to improve the algorithm further by reducing the sub-threshold scanning available to an attacker; exploring optimal communication strategies; and developing techniques to obtain a complete enterprise-trace for further testing.

The hardware implementation will target the ML300 demonstration platform by Xilinx [29]. This board contains 4 gigabit Ethernet connections, a small FPGA, and a single bank of DDR-DRAM. The DRAM bank is sufficiently large to meet our design goals, while the DRAM's internal banking should enable the address and connection tables to be both implemented in the single memory.

We will integrate our software implementation into the Bro IDS, with the necessary hooks to pass IP blocking information to routers (which Bro already does for its current, less effective scan-detection algorithm). Doing so will require selecting a different 32-bit block cipher, as our current cipher is very inefficient in software. For both hardware and software, we aim to operationally deploy these systems.

Finally, we are investigating ways to capture a full-enterprise trace: record every packet in an large enterprise network of many thousands of users. We believe this is necessary to test worm detection and suppression devices using realistic traffic, while reflecting the diversity of use which occurs in real, large intranets. Currently, we are unaware of any such traces of contemporary network traffic.

## 10 Conclusions

We have demonstrated a highly sensitive approximate scan-detection and suppression algorithm suitable for worm containment. It offers substantially higher sensitivity over previously published algorithms for worm containment, while easily operating within an 8 MB memory footprint and requiring only 2 uncached memory accesses per packet. This algorithm is suitable for both hardware and software implementations.

The scan detector used by our system can limit worm infectees to sustained scanning rates of 1 per minute or less. We can configure it to be highly sensitive, detecting scanning from an idle machine after fewer than 10 attempts in short succession, and from an otherwise normal machine in less than 30 attempts.

We developed how to augment the containment system with using *cooperation* between the containment devices that monitor different cells. By introducing communication between these devices, they can dynamically adjust their thresholds to the level of infection. We showed that introducing a very modest degree of bias that grows with the number of infected cells makes a dramatic difference in the efficacy of containment above the epidemic threshold. Thus, the combination of containment coupled with cooperation holds great promise for protecting enterprise networks against worms that spread by address-scanning.

## 11 Acknowledgments

Funding has been provided in part by NSF under grant ITR/ANI-0205519 and by NSF/DHS under grant NRT-0335290.

## References


[1] R. Anderson, E. Biham, and L. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard.

[2] B. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *CACM*, July 1970.

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

[4] CERT. Code Red II: Another Worm Exploiting Buffer Overflow in IIS Indexing Service DLL, http://www.cert.org/incident_notes/in-2001-09.html.

[5] S. Crosby and D. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium*. USENIX, August 2003.

[6] eEye Digital Security. .ida "Code Red" Worm, http://www.eeye.com/html/Research/Advisories/AL20010717.html.

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

[8] L. T. Heberlein, G. Dias, K. Levitt, B. Mukerjee, J. Wood, and D. Wolber. A Network Security Monitor. In *Proceedings of the IEEE Symopisum on Research in Security and Privacy*, 1990.

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

[10] J. Jung, S. Schechter, and A. Berger. Fast Detection of Scanning Worm Infections, in submission.

[11] C. Leckie and R. Kotagiri. A Probabilistic Approach to Detecting Network Scans. In *Proceedings of the Eighth IEEE Network Operations and Management Symposium (NOMS 2002)*, 2002.

[12] 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.

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

[14] M. Networks. http://www.miragenetworks.com/.

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

[16] V. Paxson. Bro: a System for Detecting Network iItruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[17] G. Project. Gnutella, A Protocol for Revolution, http://rfc-gnutella.sourceforge.net/.

[18] S. Robertson, E. V. Siegel, M. Miller, and S. J. Stolfo. Surveillance Detection in High Bandwidth Environments. In *Proc. DARPA DISCEX III Conference*, 2003.

[19] Silicon Defense. Countermalice Worm Containment, http://www.silicondefense.com/products/countermalice/.

[20] Snort.org. Snort, the Open Source Network Intrusion Detection System, http://www.snort.org/.

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

[22] S. Staniford, J. Hoagland, and J. McAlerney. Practical Automated Detection of Stealthy Portscans. *Journal of Computer Security*, 10:105–136, 2002.

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

[24] 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.

[25] Symantec. W32.blaster.worm, http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm.html.

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

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

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

[29] Xilinx Inc. Xilinx ML300 Development Platform, http://www.xilinx.com/products/boards/ml300/.

[30] C. C. Zou, W. Gong, and D. Towsley. Worm Propagation Modeling and Analysis under Dynamic Quarantine Defense. In *The First ACM Workshop on Rapid Malcode (WORM)*, 2003.