Mitigating Active Attacks Towards Client Networks Using the Bitmap Filter*

Chun-Ying Huang, Kuan-Ta Chen, and Chin-Laung Lei
National Taiwan University
Department of Electrical Engineering
No.1, Sec. 4, Roosevelt Road, Taipei, Taiwan 106
{huangant,jethro}@fractal.ee.ntu.edu.tw, lei@cc.ee.ntu.edu.tw

Abstract

With the emergence of active worms, the targets of attacks have been moved from well-known Internet servers to generic Internet hosts, and since the rate at which patches can be applied is always much slower than the spread of a worm, an Internet worm can usually attack or infect millions of hosts in a short time. It is difficult to eliminate Internet attacks globally; thus, protecting client networks from being attacked or infected is a relatively critical issue.

In this paper, we propose a method that protects client networks from being attacked by people who try to scan, attack, or infect hosts in local networks via unpatched vulnerabilities. Based on the symmetry of network traffic in both temporal and spatial domains, a bitmap filter is installed at the entry point of a client network to filter out possible attack traffic. Our evaluation shows that with a small amount of memory (less than 1 megabyte), more than 95% of attack traffic can be filtered out in a small- or medium-scale client network.

1 Introduction

An active attack is behavior that deliberately scans, probes, or intrudes on certain hosts or networks with malicious intent. Due to the popularity of Internet worms, there is always active attack traffic on the Internet. Attacks usually adopt a random IP scanning technique and infect hosts through one or more known vulnerabilities. Thus, compared with traditional Internet attacks, the victims are not limited to well-known Internet servers. Instead, anonymous client hosts have become targets for attackers. As client machines now have wider bandwidth and more computation power, recent studies [6, 13, 21] have shown that an active worm can efficiently spread among millions of hosts in a short period of time. Even if a network does not have vulnerable hosts, a huge number of random scanning packets from infected hosts can occupy precious network resources. Furthermore, vulnerable hosts may be infected by malicious intruders and then form a larger attack group.

A simple way to prevent a network being attacked or infected is to limit illegal traffic entering the network by installing a bandwidth throttling mechanism and/or an intrusion prevention mechanism at the entry point of the network. Since the bottleneck bandwidth usually lies on the link between the client network and the ISP, *these mechanisms must be installed at the ISP side, not the client network side in order to better utilize the bottleneck bandwidth.*

However, there are three major problems with a bandwidth throttling mechanism. First, since attack packets may use spoofed source IP addresses, a throttling mechanism may be not able to effectively identify attack traffic through aggregates. Second, even if an aggregate can be identified, only rate-limiting an aggregate at the edge may completely shutdown all connections depending on the aggregate. Finally, an attacker may not send a large volume of traffic, especially in the early stages of the attack so that the throttling mechanism would not be activated.

Intrusion prevention mechanisms also have drawbacks. In general, such solutions can be classified as either anomaly or misuse mechanisms. An anomaly solution monitors current network behavior and compares it with normal behavior. Any deviation from normal activity is treated as suspicious and possibly indicative of a hitherto unknown attack. However, this approach can generate both false positives and false negatives so that normal behavior may be treated as an attack or vice versa. In contrast, a misuse solution collects the signatures of well-known attacks and checks if any traffic matches patterns in the signature database. Although these solutions are more precise than anomaly mechanisms, they cannot detect unknown attacks.

Given the shortcomings of the above mechanisms, we must seek a better solution to make client networks robust

*This work was supported in part by the Ministry of Economic Affairs under the Grants 94-EC-17-A-02-S1-049, and by the Taiwan Information Security Center (TWISC), National Science Council under the Grants No. NSC 94-3114-P-001-001Y and NSC 94-3114-P-011-001.

against malicious attacks. A good candidate may be to install a stateful packet inspection (SPI) filter at the entry point of a client network to maintain the per-flow state of all outgoing connections. Thus, an intentionally active probe or intrusion that the packet filter has not encountered before would be dropped. However, installing such a mechanism at the ISP side incurs a high computational cost as the required storage space and computation complexity depends linearly on the number of concurrent active connections, which may be in the order of tens of thousands or even millions. Thus, we need a more lightweight and efficient solution.

In this paper, we propose a bitmap filter mechanism that mitigates active attacks, either DoS-like bandwidth attacks or active worm-like intrusions. The effectiveness of the bitmap filter is similar to that of an SPI filter, but it requires much less storage space and computational resources.

The remainder of this paper is organized as follows. In Section 2, we review related works. In Section 3, we discuss the usage model, client network traffic characteristics, and the detailed design of the bitmap filter. In Section 4 and Section 5, respectively, we evaluate the performance of the bitmap filter and discuss issues related to the proposed solution in detail. Finally, in Section 6, we present our conclusions.

2 Related Works

A great deal of research effort has been devoted to develop defenses against malicious attacks. While Internet users cannot tolerate high false positive rates, most implementations [4,7,16] of intrusion detection or prevention systems are misuse (also known as rule-based) mechanisms. The major differences among these implementations are the database lookup engine, the backend database, and the support service. Research in this area focuses on improving the performance of signature lookup, reducing the number of false positives and false negatives, and automatic signature generation [18]. The major problem with rule-based intrusion/prevention systems is that they cannot detect unknown attacks. Since generating and propagating new signatures is always slower than the spread of new types of attack, a network will be at risk until a patch or a signature for a new attack is released to the public.

Distributed denial of service (DDoS) counter-attack mechanisms can also be considered as solutions. However, they may not be suitable for attacks against a client network. An ingress filtering approach [8] is good for filtering outgoing packets with spoofed source addresses, but it may be not suitable for filtering incoming packets. Traceback mechanisms [14, 15], overlay networks [10, 17], and capability-based packet filtering [1,20] solutions are expensive computationally for a client network, since they require global deployment, cross-ISP cooperation, or modification of existing network architecture and usage models.

Bandwidth throttling mechanisms, such as those proposed in [5, 9, 11], adopt quality-of-service (QoS) mechanisms to rate-limit incoming packets. Before doing so, however, these mechanisms have to know that an attack is in progress, identify aggregates, and then apply rate-limiters to the identified aggregates. An aggregate is a common characteristic extracted from packets. For example, all UDP packets with a destination port of 445 is one type of aggregates.

However, there are a number of problems when deploying bandwidth throttling mechanisms in client networks. First, if attack packets sent to different clients in the same network contain random source IP addresses and destination ports, the aggregate is difficult to identify. In addition, if all traffic for a given port is rate-limited, normal traffic sent to the same destination port are also limited. This may result in denial of service for certain applications. Finally, if an attack is launched at a slow rate, the bandwidth throttling mechanism may not be triggered. The study in [5] concludes that *a bandwidth throttling mechanism is suitable for a network that holds the following conditions: 1) The target to be attacked is clear; 2) there are several up-links for the network and attacks only come from some of these up-links; and 3) it would be better if the deployment of rate-limiters is closer to attackers.* Although it may be easy for a server network to meet these conditions, it may be difficult for a client network, since *a client network usually holds only one or two up-links, the attack target is often a group of random hosts, and the rate-limiters can only be installed close to the client network.*

Compared to all the above solutions, we believe that an SPI-based mechanism is a better choice for client networks. However, since an SPI mechanism has to keep all per-flow states, adopting it incurs high cost for an ISP. Take a popular SPI implementation in the Linux open-source operating system [19] as an example. The required storage space grows linearly according to the number of kept flows. Besides, the data structures used to maintain these states are basically link-lists with an indexed hash table that reduces the length of a link-list. Obviously, both the storage and computation complexities are $O(n)$, which is not affordable for a larger ISP containing several client networks.

3 The Bitmap Filter

By definition, a client network should have only client hosts, such as a business enterprise customer, a group of DSL users, a wireless network, or a building on a campus. Usually client hosts only initiate requests, and seldom receive requests from the Internet. The bitmap filter is a lightweight and efficient method designed to mitigate active attacks, including bandwidth-attacks, intrusions, and

worms. In this section, we introduce the usage model, the client traffic characteristics, and the detailed design of our solution.

3.1 The Usage Model

Bitmap filters should be installed in an ISP network. As shown in Figure 1, an ISP usually has edge routers (black nodes) and core routers (white nodes). The bitmap filter can be installed on an edge router directly connected to a client network or a core router, which is an aggregate of two or more client networks. In Figure 1, the nodes with an outlined circle are possible locations to install the bitmap filter. Actually, the bitmap filter can be installed at any location through which traffic from client networks must pass.

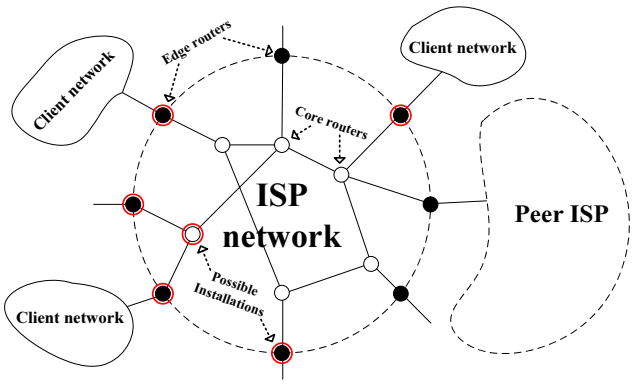

Figure 1. An ISP network with bitmap filters.

3.2 The Client Network Traffic Characteristics

The design of the bitmap filter leverages certain client network traffic characteristics to improve the filter performance. Thus, before constructing our solution, we make an observation about several client networks. A 6-hour TCP and UDP packet trace was collected on a core router between 10AM and 4PM on a weekday. The router aggregates the up-links of six class C client networks on a campus. In the 6-hour trace, 96.25% were TCP packets and 3.75% were UDP packets. The average packet rate was 24.63K per second, the average bandwidth utilization was 138.55Mbps, and the average packet size was 720 bytes. First, we examine the connection lifetime of TCP connections in the trace data. The lifetime of a connection is counted from the appearance of the first TCP-SYN packet to the appearance of a TCP-FIN or TCP-RST packet. The connection lifetime varies widely from a minimum of several milliseconds to a maximum of six hours, as shown in Figure 2-a (data exceeding the 12000^{th} second are removed, since there are no more peaks). However, the lifetime of most connections is short. The statistics show that 90% of connections are under 76 seconds, 95% are under 6 minutes, and less than one percent last for more than 515 seconds. Although the lifetime for each connection varies greatly, an interesting phenomenon is that *the out-in packet delay is always short*. Before introducing out-in packet delay, we define two types of packet. An *outgoing packet* is a packet sent from a client network, while *incoming packet* is a packet received by a client network. A packet always contains its address information in a tuple τ of $\{source\text{-}address, source\text{-}port, destination\text{-}address, destination\text{-}port\}$. Thus, for an outgoing packet with an address tuple of $\tau_{out} = \{saddr, sport, daddr, dport\}$, the address tuple of its corresponding incoming packet should be in an inverse form, that is $\tau_{in} = \{daddr, dport, saddr, sport\}$. Note that for an outgoing packet and its corresponding incoming packet, τ_{in}^{-1}, which is the inverse of the tuple τ_{in}, and τ_{out} should be the same. Based on these definitions, the out-in packet delay is then obtained as follows:

1. On receipt of an outgoing packet with an address tuple $\tau_{out} = \{saddr, sport, daddr, dport\}$ on an edge router at timestamp t, the router checks if the tuple has been recorded previously. If the tuple is new, it is associated with the timestamp t and stored in the edge router's memory. Otherwise, the existing tuple is updated with the timestamp t.

2. On receipt of an incoming packet with an address tuple $\tau_{in} = \{daddr, dport, saddr, sport\}$ at timestamp t, the edge router checks if the inverse tuple τ_{in}^{-1} has been recorded before. If it already exists, the timestamp associated the inverse tuple τ_{in}^{-1} is read as t_0 and the out-in packet delay is computed as $t - t_0$.

3. To avoid the problem of port-reuse, which affects the accuracy of computing the out-in packet delay, an expiry timer T_e deletes existing address tuples when $t - t_0 > T_e$.

The out-in packet delay may be caused by network propagation delay, processing delay, queueing delay, or mechanisms like delayed-ACK [3]. However, they should not be too long. The statistics of out-in packet delay are shown in Figure 2-b. Since we use a large timer, $T_e = 600$ seconds, to handle expired address tuples, in Figure 2-b, the effect of port-reuse can be observed roughly at the peaks. Although the port-reuse timer varies in different implementations, we find that most of them are in multiples of 60 seconds. The statistics also show that most out-in packet delays are very short. In Figure 2-c, 99% of out-in packet delays are under 2.8 seconds. The result also implies that the most Internet traffic is bi-directional.

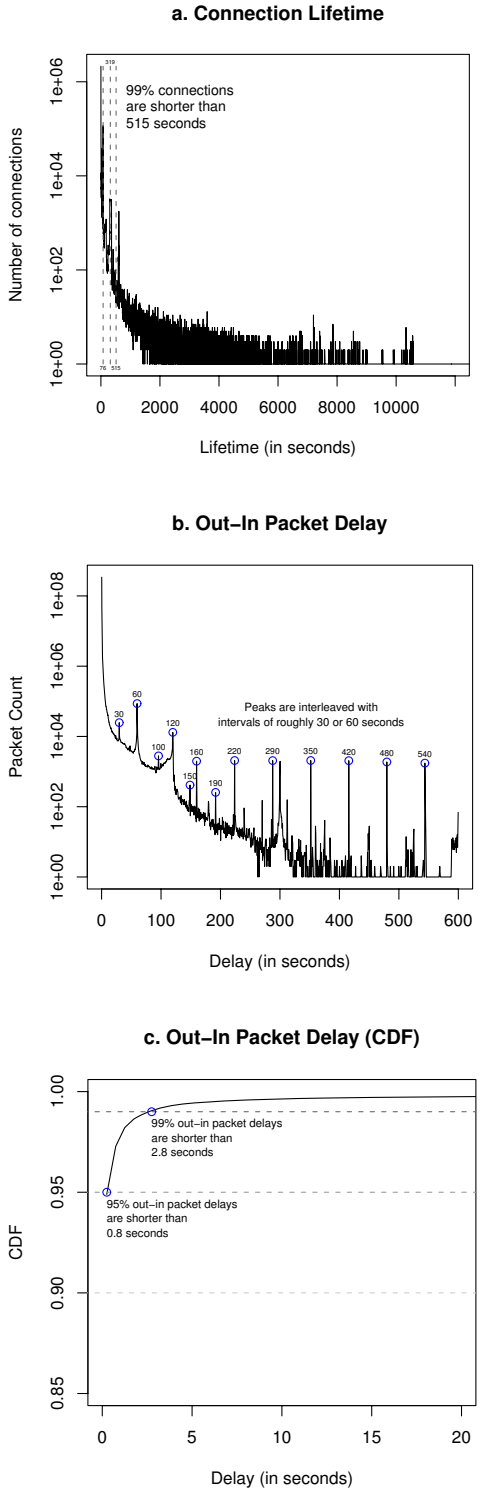

Figure 2. Traffic characteristics extracted from a client network. The sampled traffic is a 6-hour packet trace collected from six class-C client networks on a campus.

3.3 Construct the Bitmap Filter

Based on the observations that 1) the client network traffic is bi-directional, and 2) most out-in packet delays are short, a naïve solution is to filter out unwanted traffic. Suppose that a timer with an initial value of T is associated with the address tuple $\tau_{out} = \{source\text{-}address, source\text{-}port, destination\text{-}address, destination\text{-}port\}$ of each outgoing packet that is new to an edge router. If the tuple τ_{out} is not new to the router, the value of the associated timer is simply reset to T. The timer reduces every time unit Δt. When the timer expires (reaches zero), the associated address tuple is deleted. For each incoming packet, the router extracts the address tuple τ_{in} and checks if the inverse tuple τ_{in}^{-1} exists. If it exists, the packet is bypassed; otherwise, it is dropped.

Like SPI-based mechanisms, the above solution has several drawbacks. For example, the complexity of storage and computation make it infeasible to deploy in an ISP network. Thus, a bitmap filter, which is a composite of k bloom filters [2] of equal size 2^n-bit, denoted as a $\{k \times n\}$-bitmap filter, is used instead. An example of a bitmap is illustrated in Figure 3. Each column in the bitmap represents a bit-vector of a bloom filter. For convenience, in the algorithm, the bit-vector of the i^{th} bloom filter is written as $bit\text{-}vector[i]$.

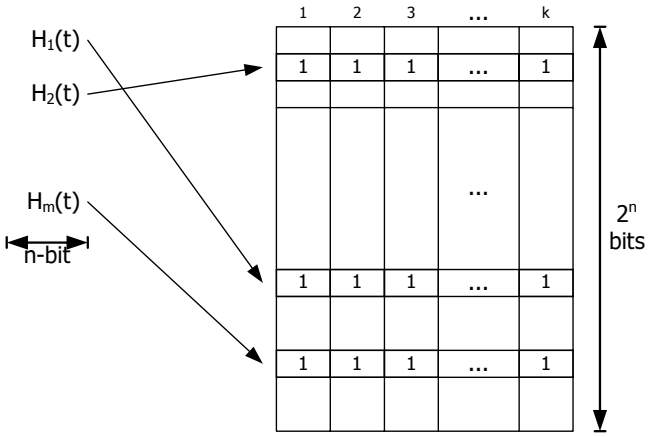

Figure 3. An example of a $\{k \times n\}$-bitmap, the core architecture for a bitmap filter.

First, the $\{k \times n\}$-bitmap is initialized to zero, and an index of the current bit vector idx is set to the first bit-vector. All the bloom filters in the bitmap share the same m hash functions, each of which should only output an n-bit value. An output that exceeds n-bit should be truncated. The bitmap filter comprises two algorithms, the $b.rotate$ algorithm, which clears expired bits from the bitmap, and the $b.filter$ algorithm, marks and looks up bits in the bitmap. The algorithms are detailed in Algorithm 1

and Algorithm 2, respectively. The $b.rotate$ algorithm is quite simple. The algorithm runs every time unit Δt. When it is activated, the index of the current bit vector idx is set to the next bit vector and all bits in the previous bit vector are set to zero. For example, assume there are k bit vectors in a bitmap filter indexed from $\{1, \cdots, k\}$. If the current index is set to the 1^{st} bit vector, then the last bit vector will be the k^{th} bit-vector; however, if the current index is set to the $j^{th}(1 < j \leq k)$ bit vector, then the last bit vector will be the $(j-1)^{th}$ bit-vector.

Algorithm 1 The Timer Handler - $b.rotate()$

Require: An initialized $\{k \times n\}$-bitmap and an index to current bit vector idx.

1: $last = idx$
2: $idx = (idx + 1) \pmod k$
3: set all bits in $bit\text{-}vector[last]$ to zero
4: **return** idx

The bitmap is marked and looked up using the $b.filter$ algorithm, as shown in Algorithm 2. When a packet is received by an edge router, the $b.filter$ algorithm is applied to determine whether the packet should be bypassed or dropped. For an outgoing packet, the $b.filter$ iteratively applies all the m hash functions on the tuple τ_{out} and marks the corresponding bits in all bit vectors to a value of 1. Outgoing packets are always bypassed. On the other hand, when an incoming packet is received, the $b.filter$ iteratively applies all the hash functions on the tuple τ_{in}^{-1} and checks if the corresponding bit in the current bit vector indicated by the index idx is marked or not. If a bit is not marked, then the packet will be dropped.

Algorithm 2 The Filtering Function - $b.filter()$

Require: An initialized $\{k \times n\}$-bitmap, an index of current bit vector idx, and a packet pkt to be inspected.

1: **if** pkt is an output packet **then**
2: **for** $h \in hash\text{-}function\ list$ **do**
3: $j = h(\tau_{out})$
4: mark the j^{th} bit in all bit vectors as 1
5: **end for**
6: **else if** pkt is an input packet **then**
7: **for** $h \in hash\text{-}function\ list$ **do**
8: $j = h(\tau_{in}^{-1})$
9: **if** the j^{th} bit in $bit\text{-}vector[idx]$ is 0 **then**
10: **return** DROP
11: **end if**
12: **end for**
13: **end if**
14: **return** PASS

Note that the bitmap filter does not use all fields in the address tuple τ to compute the hash value. Instead, for an outgoing packet, it only hashes $\{source\text{-}address, source\text{-}port, destination\text{-}address\}$. In contrast, for an incoming packet, only $\{destination\text{-}address, destination\text{-}port, source\text{-}address\}$ are used to compute the hash value. Further details are given in Section 5.1.

In summary, the "mark" action is always performed for all bit vectors, the "look up" and the "clean up" actions are only performed for the current bit vector and the last bit vector, respectively. The combination of these operations achieves the same purpose as the naïve solution described at the beginning of this sub-section, which effectively filters out unwanted traffic sent to a client network.

3.4 Choose Proper Parameters

As stated in Section 3.3, several parameters for the bitmap filter must be decided. They are the k - the number of bit vectors in a bitmap, the n - the size of a bit vector, the Δt - the time unit to clean up a bit vector, and the m - the number of hash functions used in the bitmap filter. The k and n parameters decide how much storage space is required for the bitmap filter; and the k and Δt parameters decide the countdown time of the timer T_e mentioned in Section 3.2. Thus, given a moderate expiry timer T_e and a proper time unit Δt, the value k can be decided by $\lfloor \frac{T_e}{\Delta t} \rfloor$.

Recall the result in Section 3.2. T_e should not be too long, since the port-reuse effect may incur more false negatives. In other words, a packet that should be dropped may be accepted by the filter. However, to prevent overkilling connections with longer delays, T_e should not be too short either. A value below 60 seconds, such as 20 or 30 seconds, would be acceptable. On the other hand, the time unit Δt need not to be too short. Although a shorter Δt improves the timer's granularity, a Δt that is too short may raise the frequency of running bitmap clean-ups too much and thus reduce the overall performance of the system. A value of 4 or 5 seconds would be appropriate.

The n is a flexible parameter. An ISP can decide the value according to the number of concurrently active connections and the memory space that they are willing to devote to the system. Note that a small n will also raise the possibility of false negatives and reduce the effectiveness of packet filtering. To avoid the problem, more hash functions (i.e., m) may be used to reduce false negatives. When deploying such a system, administrators should consider a trade-off between storage space and computation power to decide the value of n and m. We further evaluate the effects of different sets of parameters in the next section.

4 Evaluations

In this section, we evaluate several aspects of the proposed solution by analyses, comparisons, or simulations.

4.1 False Positives and False Negatives

In our solution, the definition of a false positive is the same as that used in generic intrusion detection mechanisms. That is, an instance of normal behavior is detected as an attack. In contrast, a false negative is an attack that is treated as normal behavior. Since the bitmap filter works in flavor of a positive listing, only incoming packets with an out-in packet delay longer than the expiry timer T_e are filtered out. Thus, the number of false positives is very low. As the result in Section 3.2 shows, false positives should be lower than 1% when T_e is greater than 2.8 seconds.

However, we should focus more on false negatives. Assume m hash functions are applied to a single incoming packet and the utilization of the current bit vector is $U = \frac{b}{2^n}$, where b is the number of marked bits in a bit vector. The probability p that a random incoming tuple τ will penetrate the bitmap filter is

$$p = U^m = \left(\frac{b}{2^n}\right)^m. \quad (1)$$

The number of marked bits on the bit vector should be proportional to the number of active connections c inside a time unit of T_e. If we assume that the results of the hash functions seldom collide when the utilization of the bit vector is low, Equation 1 can be rewritten as

$$p \simeq \left(\frac{c \cdot m}{2^n}\right)^m. \quad (2)$$

Given a bit vector size n and the expected max number of active connections c, then to minimize the desired penetration probability p, we differentiate Equation 2 and get

$$p' = \left(\frac{c}{2^n} \cdot m\right)^m \left(1 + \ln\left(\frac{c}{2^n} \cdot m\right)\right). \quad (3)$$

Thus, m that minimizes the penetration probability p can be obtained by solving $1 + \ln(\frac{c}{2^n} \cdot m) = 0$, which is

$$m = \frac{e^{-1} \cdot 2^n}{c}, \quad (4)$$

where e is the base for the natural logarithm. By replacing m in Equation 2 with $\frac{e^{-1} \cdot 2^n}{c}$ when m minimizes the penetration probability p, the ratio of the expected max number of active connections c should satisfy

$$\frac{c}{2^n} \leq -\frac{1}{e \ln p}. \quad (5)$$

For example, if we adopt a bitmap filter of size $n = 20$ (about 1-million bits) with $k = 4$, and $\Delta t = 5$ seconds, and set the desired penetration probability to be roughly 10%, 5%, and 1%, the number of active connections inside a time unit $T_e = 20$ seconds should be less than 167K, 125K, and 83K, respectively. Compared with our trace data, which has only average 15K active connections inside a time unit of 20 seconds, these upper bounds are much higher than the actual traffic. The number of used hash functions m in the setup can be 3, and the memory space required by the bitmap filter is only $(k \times 2^n)/8 = 512K$ bytes.

4.2 Performance

The bitmap filter is efficient because almost all operations can be performed in constant time. The processing time for an outgoing packet is $O(m \times t_h) + O(k \times t_m)$, where m is the number of used hash functions, t_h is the time taken to execute a hash function, k is the number of bit vectors to be marked, and t_m is the processing time to mark a bit. Since hash functions can be implemented as a dedicated hardware chip, the processing time is negligible. Thus, the outgoing packet process can be treated as a constant time operation. Processing incoming packets is simpler than for outgoing packets. The required processing time is $O(m \times t_h) + O(t_c)$ where t_c is the processing time need to check whether a bit on a bit vector is marked or not. Incoming packet processing is also a constant time operation.

The most time consuming operation may be the $b.rotate$ algorithm, which executes every Δt seconds. The algorithm first advances the current index idx to set to the next bit vector, and then resets all bits in the last bit vector to zero. Thus, the operation is proportional to the size of a bit vector, which is $O(n)$. However, since the memory space of a bit vector is fixed and continuous, implementing such an algorithm in software or hardware should be very simple and efficient. We also compare the performance of the bitmap filter and SPI based-implementations. In Table 1, the "hash+link-list" implementation is the method used in the popular open-source Linux operating system. The "AVL-tree" is an implementation that efficiently reduces the time complexity searching flow states. Our solution is listed in the column labelled "bitmap filter".

4.3 Simulation with the Packet Trace

We also perform several simulations to verify the effectiveness of the bitmap filter. A bitmap filter and an SPI-based packet filter are both implemented. The input to both filters is the packet trace used in Section 3.2. First, we compare the packet drop rate of the filters. The SPI filter is set to delete idle connections after 240 seconds, which is the default TIME_WAIT timeout used in the Microsoft windows operating system [12]. The bitmap filter is configured as follows: $n = 20$, $k = 4$, $T_e = 20$, and $\Delta t = 5$. This constructs a 512K-byte bitmap filter that handles the out-in packet latency shorter than 20 seconds. As Figure 4

Table 1. Performance comparison of the bitmap filter and SPI-based filters.

	Hash + link-list (Linux)	AVL-tree	Bitmap filter
Storage space - Complexity.	$O(n)$	$O(n)$	$O(n)^{(a)}$
Storage space - Handle maxima 2.56M concurrent connections.	76.8M bytes $^{(b)}$	76.8M bytes $^{(b)}$	8M bytes $^{(c)}$
Computation complexity - Insert a new state.	$O(1)$	$O(\log n)$	$O(1)$
Computation complexity - Lookup an existing state.	$O(n)$	$O(\log n)$	$O(1)$
Computation complexity - Garbage collection $^{(d)}$.	$O(n)^{(e)}$	$O(n)^{(e)}$	$O(n)^{(f)}$
Hardware acceleration	Possible	Difficult	Easy

(a) Although the complexity of storage space is also $O(n)$, the required memory space of the bitmap filter is much smaller than other implementations when handling the same number of active connections. (b) The size of a flow state is set at 30 bytes, including source address, source port, destination address, destination port, connection state, timestamp, and pointers to maintain the list or tree data structure. (c) The random packet penetration rate is set at about 10%. (d) The purpose of garbage collection is to remove expired flow states. (e) The garbage collector has to traverse all states kept in the memory. (f) The garbage collector only resets values in a fixed-size and continuous memory to zero.

shows, the filters have similar packet drop rates, and the gray-dashed line has a slope of 1.0. The SPI filter has an average drop rate of 1.56% compared to 1.51% for the bitmap filter. This is because that the SPI filter knows the exact time of closed connections and can therefore drop packets precisely than the bitmap filter.

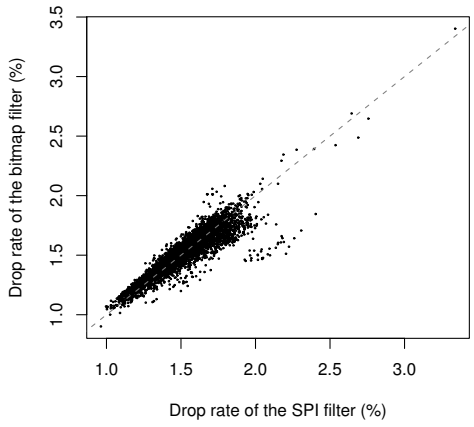

Figure 4. Comparison of the packet drop rates of the SPI and the bitmap filters. The gray-dashed line has a slope of 1.0**.**

We also test the filtering rate of the bitmap filter. In this simulation, an attack generator releases incoming attack packets with address tuples in the form of $\{saddr, sport, daddr, dport\}$, where $saddr$, $sport$, and $dport$ are chosen at random; however, $daddr$ is confined to the address space of the given sub-networks. The random attack packets are generated at the rate of 500K packets per second, which is about 20 times faster than the normal traffic packet rate in the trace data. The attack traffic is mixed with the normal packet trace and then fed to the bitmap filter. To prove the effectiveness of the bitmap filter, each attack packet is verified whether it penetrates the bitmap filter or not. The result of the test is shown in Figure 5. The attack begins at the 10800^{th} second. In Figure 5-a, the black line represents the number of packets that penetrates the bitmap filter. The light-gray area indicates the number of normal packets, while the dark-gray area shows the number of attack packets. We can see that the amount of penetrated traffic is similar to normal traffic as the black line fits the border of the light-gray area. Figure 5-b shows the filter rate of attack packets. In our simulation, almost all attack packets (99.983% on average) are filtered out. The result thus shows that the 512K bytes $\{4 \times 20\}$-bitmap filter with 3 hash functions can effectively filter out attacks for the small- or medium-scale client networks.

5 Discussion

In this section, we discuss several issues related to the bitmap filter, including the bitmap filter's compatibility with existing Internet protocols, possible attacks on the bitmap

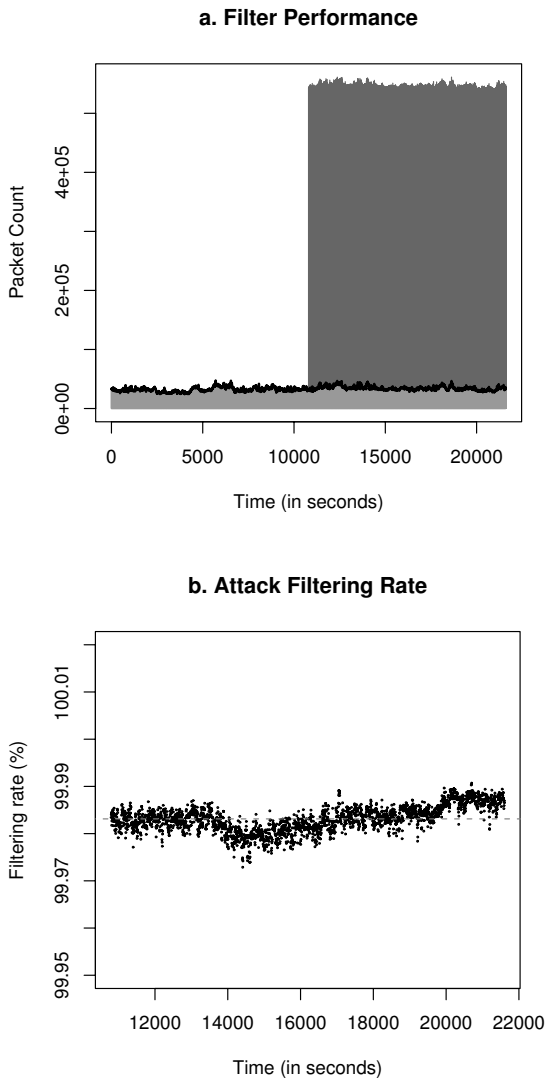

Figure 5. The performance of the bitmap filter for the mixed attack and normal traffic.

filter, and design issues.

5.1 Compatibility

The bitmap filter is completely compatible with all client initiated Internet protocols, including the hyper-text transfer protocol (HTTP), the e-mail transmitting/receiving protocols (SMTP, POP3, IMAP), the file transfer protocol (FTP) using passive mode, the telnet protocol, and the secure shell (SSH) protocol. Note that use of the bitmap filter does not require modification of existing network infrastructure. However, since the filter drops all active requests sent to the client network, some protocols that separate the command and the data channel may be problematic. Protocols like the active mode FTP or peer-to-peer protocols, will fail when another data communication channel that initiates outside the client network is required.

To solve the problem, these applications cat adopt the *hole punching* technique. That is, when a client requires active connections initiated by an outsider, it first sends out a packet to mark related bits in the bitmap filter. The remote peer can then reach the client inside the protected network. Take the active mode FTP as an example, and assume that a client c needs the server s to transfer data to a port p. The client can send a TCP or UDP packet with the address tuple $\{c, p, s, x\}$, where x is a random port number, to the server. Then, the server is allowed to actively connect to the port p of the client before the marked corresponding bits on the bitmap filter expire.

Since a client always establishes a command channel with outsiders before creating more data channels, most protocols that require extra data channels should be able to function properly with the hole punching technique. Network administrators may also consider the solution described in Section 5.3 if the purpose of deploying the bitmap filter is only to mitigate bandwidth attacks against client networks.

5.2 Attack from Insiders

The bitmap filter may fail when an internal user are attacking outsiders. Suppose that a client host is infected with worms. When the client transmits a large volume of random traffic to other networks, the bitmap filter will be filled with the malicious traffic, which would increase the random packet penetration rate. Given an attack rate r, the increased bitmap utilization U will be roughly $\frac{m \cdot r \cdot T_e}{2^n}$. To prevent such attacks, we can use a larger bitmap (by increasing n) or shorten the expiry timer T_e. As the results in Section 3.2 show, it may be safe to reduce the T_e to around 3 or 5 seconds because 99% of out-in packet delay is relatively short.

However, since there are limitations on both n and T_e, the bitmap filter may ultimately be compromised by the out-

going attacks if the number of attackers inside the client network continues to increase. The best way to avoid the problem is to find the attacking hosts and eliminate attackers inside the client network. The design of the bitmap filter is based on the principle of fairness. That is, a network will not be attacked if it does not attack other networks.

5.3 Adaptive Packet Dropping

If the sole purpose of deploying a bitmap filter is to mitigate bandwidth attacks against client networks, the administrator may consider using adaptive packet dropping (APD), instead of dropping all unmatched incoming packets. An APD-enabled bitmap filter uses one or more indicators to decide whether a "should-be-dropped" packet will be actually dropped. Two adaptive packet dropping designs are shown as follows. The first takes bandwidth utilization as the indicator:

1. The edge router monitors the bandwidth utilization U_b ($0 \leq U_b \leq 1$) of given links.
2. The bitmap filter runs as usual: it decides to pass or drop a packet.
3. When a packet is to be dropped, the edge router drops it with a probability of U_b.

The second design takes the ratio of incoming packets versus outgoing packets as the indicator:

1. The edge router monitors the incoming packet count P_{in} and outgoing packet count P_{out} of given links.
2. The bitmap filter runs as usual: it decides to pass or drop a packet.
3. Given two thresholds l, h ($l < h$), and an indicator $r = \frac{P_{in}}{P_{out}}$, when a decision is made to drop a packet, the edge router drops it with a probability of

$$p = \begin{cases} 0 & \text{, if } r < l \\ \frac{r-l}{h-l} & \text{, } l \leq r \leq h \\ 1 & \text{, if } r \geq h \end{cases}$$

Note that when adaptive packet dropping is enabled for incoming packets, the packet marking policy of the bitmap filter must be modified. In a non-APD-enabled bitmap filter, all outgoing TCP and UDP packets are considered in order to mark corresponding bits in the bitmap. However, for an APD-enabled bitmap filter, the outgoing packets are classified as signal packets and data packets. Outgoing UDP and TCP data packets without SYN, FIN, or RST flags, are still used for marking bit vectors. However, outgoing TCP signal packets with SYN+ACK, FIN+ACK, RST, or RST+ACK flags, are not used to mark bit vectors. An exception is that when a signal packet only takes SYN or FIN flag, it is used to mark the corresponding bits in the bitmap filter. This is because an APD-enabled bitmap filter may admit all incoming packets when the dropping probability is low. In such a condition, *the bitmap filter should only keep those connections that are really connected*. For attacks like SYN-scanning or FIN-scanning, which cause the target host to return SYN+ACK, FIN+ACK, or RST, marking the bitmap filter carefully can avoid a rapid increase in the number of false negatives caused by attacks that reduce the efficiency of the bitmap filter.

5.4 Colluding with Attackers

To effectively attack a client network protected by the bitmap filter, attackers have to "guess" the connections with hosts in the network. An attacker may consider installing sniffers on clients inside the network or at peers connected to the client network. Obviously, identifying connections at peers is not efficient, since a client network's connections are diverse. Besides, aggregates are easy to find and rate-limit if the number of monitored connected peers is not very large.

Although it may be possible to identify the connection states of client hosts inside a client network, it may not help a potential attacker. First, in a switching environment, it is hard for a client to capture traffic from other hosts. Thus, the client can only report its own connection states to attackers. Since these connection states also form aggregates to identify sniffers, they can be blocked easily. In addition, if a sniffer can capture traffic from all clients and report connection states to attackers, short connections will be deleted quickly from a bitmap filter with a short expiry timer T_e. In such a situation, the sniffer has to report new states to attackers frequently, which increases the risk of the sniffers and attackers being identified. Thus, colluding with attackers may be not a suitable strategy to attack the bitmap filter.

6 Conclusions

A client host that connects to the Internet will always receive random attack traffic whether it is vulnerable or not. With the rapid development of telecommunication technologies and hand-held mobile devices, new Internet clients also receive massive amounts of random attack traffic. To filter out such traffic, we propose a bitmap filter, which stops most malicious traffic. Our analyses and simulations show that with a small amount of resources and proper configuration, an ISP can efficiently filter out 90% to 99% of attack traffic for client networks.

References

[1] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet denial-of-service with capabilities. *SIGCOMM Computer and Communication Review*, 34(1):39–44, 2004.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM*, 13(7):422–426, 1970.

[3] R. Braden. Requirements for internet hosts – communication layers. *RFC 1122*, Oct. 1989.

[4] CheckPoint Software Technologies Ltd. *Internal Security*.

[5] S. Chen and Q. Song. Perimeter-based defense against high bandwidth DDoS attacks. *Transactions on Parallel and Distributed Systems*, 16(8):526–537, June 2005.

[6] Z. Chen, L. Gao, and K. Kwiat. Modeling the spreak of active worms. In *Proceedings of IEEE 22th Annual Joint Conference of the IEEE Computer and Communications Societies.* IEEE INFOCOM, Mar. 2003.

[7] Cisco Systems. *Cisco IPS 4200 Series Sensors*.

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

[9] J. Ioannidis and S. M. Bellovin. Implementing pushback: Router-based defense against DDoS attacks. In *Proceedings of Network and Distributed System Security Symposium*. ISOC, 2002.

[10] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. *IEEE Journal on Selected Areas in Communications*, 22:176–188, Feb. 2004.

[11] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *SIGCOMM Computer and Communication Review*, 32(3):62–73, 2002.

[12] Microsoft Corporation. *TCP/IP and NBT configuration parameters for Windows XP*.

[13] D. Moore, C. Shannon, and k claffy. Code-Red: a case study on the spread and victims of an internet worm. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 273–284, New York, NY, USA, 2002. ACM Press.

[14] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Network support for IP traceback. *IEEE/ACM Transactions on Networking*, 9(3):226–237, 2001.

[15] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer. Single-packet IP traceback. *IEEE/ACM Transactions on Networking*, 10(6):721–734, 2002.

[16] Sourcefire, Inc. *Snort - the de facto standard for intrusion detection/preveison*.

[17] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. *IEEE/ACM Transactions on Networking*, 12(2):205–218, 2004.

[18] Y. Tang and S. Chen. Defending against internet worms: A signature-based approach. In *Proceedings of IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* IEEE INFOCOM, Mar. 2005.

[19] L. Torvalds. *The Linux open-source operating system - kernel archive.*

[20] A. Yaar, A. Perrig, and D. Song. SIFF: a stateless Internet flow filter to mitigate ddos flooding attacks. In *Proceedings of Symposium on Security and Privacy*, pages 130–143, May 2004.

[21] C. C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 138–147, New York, NY, USA, 2002. ACM Press.