High-Speed Router Filter for Blocking TCP Flooding under DDoS Attack

Yoohwan Kim[1], Ju-Yeon Jo[1], H. Jonathan Chao[2] and Frank Merat[1]

[1]Electrical Engineering and Computer Science Department
Case Western Reserve University, Cleveland, OH 44106

[2]Electrical and Computer Engineering Department
Polytechnic University, Brooklyn, NY 11201

ABSTRACT

Protection from Distributed Denial-of-Service attacks has been of a great interest recently and substantial progress has been made for preventing some attack types. However the bandwidth exhaustion attack remains difficult to prevent because the firewalls or servers cannot prevent it locally and a network-wide collaboration is necessary. The routers in use today are not capable of blocking the attack traffic selectively because they cannot distinguish the malicious packets from the legitimate packets. The popular methods based on IDS are difficult to use in high-speed routers due to heavy processing and unnecessary packet drops by false positives. In this paper we present a hardware solution that can reliably block most of the malicious TCP traffic while passing all the legitimate TCP traffic. This filtering is transparent to the hosts or routers and a filtering device can be easily attached to router ports. Our filter is composed of two modules, the connection verifier module that monitors the TCP connection establishment process and the filter array module that selectively passes the legitimate packets. By allocating bandwidths separately for TCP from other protocol types, the TCP portion of the bandwidth can be protected during the attack when such a filter is used. Since many public servers only communicate via TCP, protecting TCP portion is enough for providing protection to those servers. The filter can process packets at 10 Gbps, so it is suitable for most high-speed routers. To prove the concept, we have built a simulation model using OPNET simulation package, where the attack agents create massive attack traffic with spoofed source addresses and ports to a victim. The filters successfully blocked 99.9% of the attack traffic while legitimate traffic showed nearly identical performance as in the non-attacked condition.

Keywords: *Network Security, Denial-of-Service Attack, TCP Flooding, Router Filter*

1 Introduction

Distributed Denial-of-Service (DDoS) attack has drawn a great attention for research after the attack incidents for well-known web sites.[10] In DDoS attack, a large amount of packets are generated and delivered to a victim machine in order to cause a network congestion or to make the victim machine waste resources on such irrelevant packets, preventing the legitimate packets from being delivered and/or processed in time. DDoS attacks are very easy to launch and occurring quite frequently[19], but their detection and protection are still challenging. While the computing resource exhaustion attacks such as TCP SYN flooding[3] may be blocked on the server side[1][4][9], the bandwidth exhaustion attack must be blocked at the routers before the attack traffic congests the link to the victim servers. A commonly used approach is limiting the traffic amount destined to the victim server, but with lack of an effective method for distinguishing the legitimate traffic from the malicious traffic due to source address spoofing, it inevitably affects all the traffic destined to the victim machine, possibly resulting in a denial-of-service condition. Many rate-limiting algorithms have been proposed[12][16][25][26], but isolating and blocking only the malicious packets has not been proposed so far. A scheme called pushback[8][14][22], where the rate limiting is propagated to the routers toward the source of the attack, is effective in isolating and limiting the attack traffic, but it requires all the routers along the attack paths to be upgraded, which may not be immediately available in near future. Other mitigation schemes have been proposed[2][6][11][17][20], but they rarely achieve complete blocking of attack traffic in general conditions. Meanwhile, the DDoS attack is likely to remain as a critical threat for the public servers.

In this research, we are interested in isolating and passing only the legitimate traffic at the edge router at the Point-of-Presence (POP). Passing only the legitimate traffic is more advantageous than blocking known attack traffic because it can block all unknown attack traffic in advance without constantly adding new rules for attack traffic detection. This significantly reduces the maintenance cost. Blocking the attack traffic at the POP location also has some advantages. First, the service providers can offer a protection from DDoS attack immediately without requiring an upgrade of the core routers or coordination with other service providers. They can place the filtering devices only where necessary, i.e., where it is requested and financially supported by the customers. Secondly, blocking the attack at the POP is sufficient in blocking DDoS attack in most cases. Most of the DDoS attack is rather concentrating the traffic to a victim server than increasing the overall traffic amount[18]. For a high-speed edge router in the POP, the link from the backbone usually remains underutilized even during the flooding attack where the link between the victim server and its immediate access router becomes congested. In other words, the legitimate packets are usually not affected by the presence of attack traffic up to the POP location, and dropping the attack packets at the POP location can sufficiently reduce the congestion of the final link to the victim server.

Isolating and passing only the legitimate traffic while blocking all the attack traffic is very difficult to achieve in general, but such a comprehensive protection may not be necessary for most potential victim servers. Considering that many public servers provide services through TCP, protecting the TCP portion of the bandwidth may be sufficient in guaranteeing their services. Indeed, the majority of the DDoS attack is performed using TCP as summarized in Table 1[19], and a large portion of them is targeted to bandwidth exhaustion as shown in Table 2, which is obtained from the response protocol statistics[19].

Protocols	Ratio
TCP	90 – 94 %
UDP	2.4 – 5 %
ICMP	2.1 – 2.6 %
Other	2.06 – 2.92 %

Table 1: Distribution of DDoS attack types

	Trace 1		Trace 2		Trace 3	
	Attacks	Packets	Attacks	Packets	Attacks	Packets
TCP SYN flooding	9.1 %	1.8 %	7.1 %	2.0 %	7.3 %	1.5 %
TCP Packet flooding	52.15 %	29.51 %	53.9 %	21.2 %	57.72 %	30 %

Table 2: Relative distribution between TCP SYN flooding and other TCP packet flooding

In this paper we present an algorithm and its implementation that can block the majority of the TCP flooding without sacrificing the legitimate TCP traffic. It is designed for high-speed router ports operating at 10 Gbps speed. The rest of the paper is organized as follows. In section 2, the basic idea of TCP filtering is described. In section 3, the detailed implementation of the filter is presented. In section 4, the design parameters for high-speed operation are discussed. In section 5, the simulation model and the results are presented. Then we conclude the paper after some discussions.

2 Overview of TCP Filtering Process

In TCP, all the packets must belong to some flow. (We use the terms *segment* and *packet* interchangeably in this paper) Except the initial connection request, i.e., TCP SYN packet, all the packets are sent in response to the previous packet. So there is no need to allow a packet to go through the router if it is not SYN or a legitimate response packet. Our filter achieves that purpose. The filter is composed of three modules, the *packet screener*, *connection verifier* and the *filter array*, which work independently performing distinct functions. The screener determines whether a packet should be processed at the filter, connection verifier detects legitimate flows, and the packet filter decides whether a packet should be dropped.

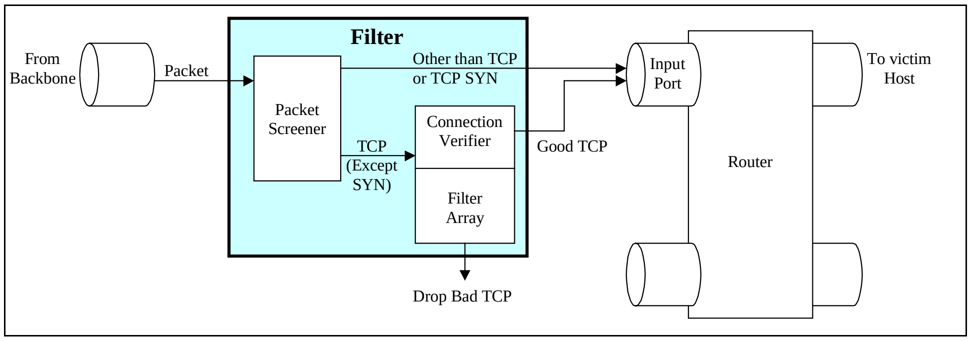


Figure 1: Inbound traffic

2.1 Screening Packets

The first work in the filter is screening the packets to determine whether to apply the filtering process or not. Except TCP SYN packets, all TCP packets are directed to the filter. All non-TCP packets and TCP SYN packets are bypassed. In case of SYN flooding attack, it is assumed that the attack can be better protected at the firewall or the server itself, and our filter is targeted for blocking a bandwidth exhaustion attack. However, the filter may be augmented with SYN flooding blocking methods, such as TCP intercept [5] or SYN cookie. The filter can be inserted either to a server side port or to a network side port. For simplicity, we show only backbone link side port in Figure 1.

For the traffic toward the network, all non-TCP packets are bypassed and TCP packets are monitored for establishing and maintaining connections.

2.2 Verifying Legitimate Flows

TCP has an explicit connection establishment and termination process. Each TCP segment has sequence number to recover from the missing segments or to rearrange the reordered segments. The connection is established through the process called 3-way handshaking as following[23]

1. Client sends a **SYN** segment with client's ISN (Initial Sequence Number).
2. Server responds with a **SYN-ACK** segment, which is Server's SYN segment with its own ISN and an acknowledgement to Client's SYN with Client's ISN+1
3. Client sends **ACK** with server's ISN+1

The connection verifier monitors the 3-way handshaking packets and grants only the flows that pass the 3 stages successfully. Figure 2 illustrates the monitoring process of connection request from the clients. The reply SYN-ACK from the server in step 2 is monitored and its ISN is recorded. If the ACK with the correct ISN comes back in step 3, the connection verifier gives grants the flow. If the source address is spoofed, either no reply or a reset packet will be received from clients. So only the legitimate requests get granted. For the connection requests initiated by the server, similar process can be applied starting from the SYN packet from the server side.

2.3 Filtering Malicious Packets

When the filter array module receives a flow ID from the connection verifier, it records the flow ID. Upon receiving a packet, it checks if the flow ID matches any of the recorded flow IDs. If there is a match, the filter array passes the packet. Otherwise it drops the packet.

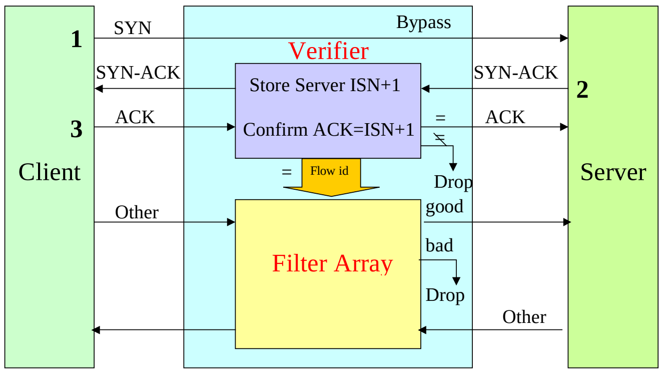


Figure 2: Verifying a legitimate flow

3 Implementation

In this section, we present the implementation for connection verifier and filter array modules. For both modules, a flow ID is used to confirm a legitimate flow. A flow is the traffic from an application in one machine to another application on another machine. The 4-tuple of {source IP, destination IP, source port, destination port} in the packet header normally designates a flow ID in IPv4. Keeping the flow information is normally an expensive process, requiring large amount of memory and processing time. Rather than keeping the full 96-bit flow ID, we use hashing to reduce the flow ID into smaller size. Hashing the flow ID produces the index to the hash table, enabling a lookup in $O(1)$ time.

When multiple flow IDs are assigned to the same hash value, it is called *collision*. Collision is handled entirely differently in two modules. In connection verifier, the ISN for each request must be kept to confirm the replied ACK, so the ISNs with collision must be stored in extra space. On the other hand, the collision in filter array is simply ignored because multiple good flows can be verified from the same hash value without any adverse effect. Rather, the collision enhances the filtering efficiency by reducing the number of hash values.

3.1 Connection Verifier

The structure of connection verifier is shown in Figure 3. The verifier has two memory regions, the primary memory and the collision memory. The hash table in primary memory is accessed by the index generated from the hash function. The ACK waiting indicator (AWI) bit indicates whether the particular hash value is expecting for a reply ACK. The collision indicator (CI) bit indicates whether more than one SYN-ACK is mapped into this particular hash value. The ISN field keeps the server ISN+1, and to save the storage it keeps only the last 16 bits of ISN+1. The timer keeps the timeout value for waiting for ACK.

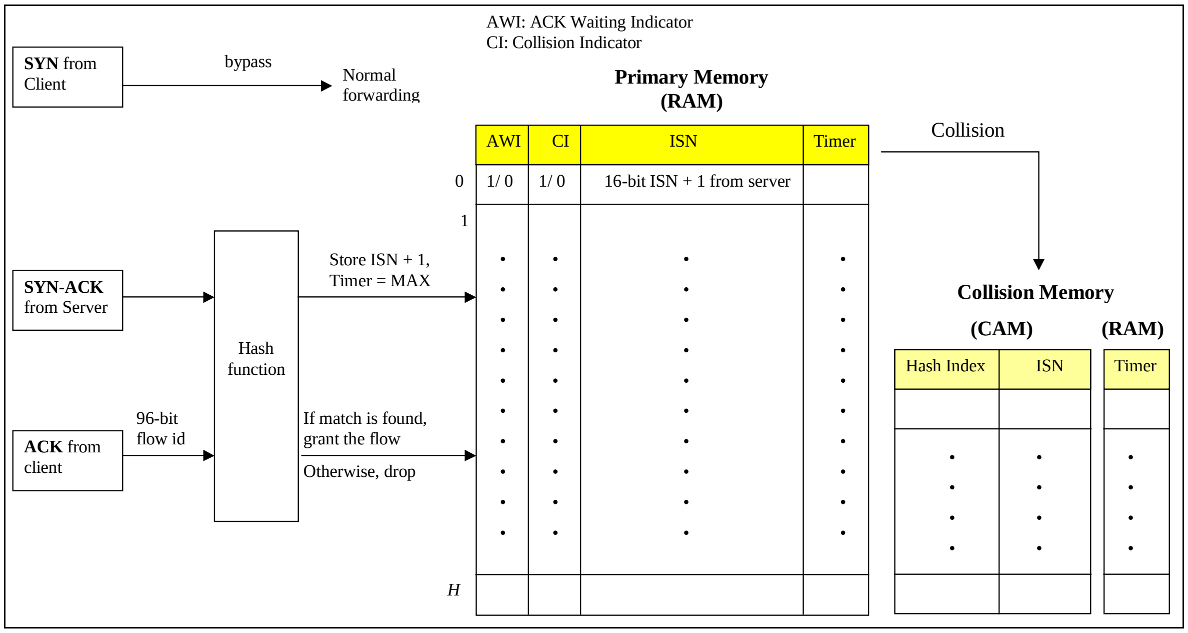


Figure 3: Connection verifier architecture

The collision memory contains CAM (content addressable memory) shown in Figure 4 that can answer the existence of a particular value in *O*(1). The entry in CAM also has individual timer. On its timeout, the associated entry gets deleted.

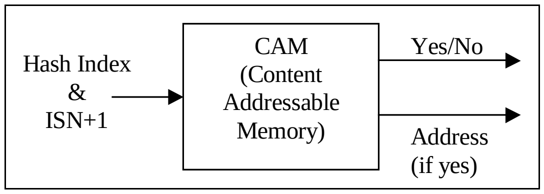


Figure 4: Content addressable memory

For the SYN-ACK packets from the server side, the following actions are taken.

1. 96-bit flow ID is hashed into hash table index
2. If AWI is 0, set AWI to 1, record the last 16 bits of ISN+1 in primary memory and set the timer to max value. (e.g., 3 seconds)
3. If AWI is 1, set CI to 1, record the last 16 bits of ISN+1 in collision memory along with the hash value. Set the timer for both primary and collision memory to max value. (e.g., 3 seconds). If the collision memory is full, do not write it.

For the ACK packets from the client side, the following actions are taken.

1. 96-bit flow ID is hashed into hash table index. (with source and destination fields switched)
2. If AWI is 0, drop the packet
3. If AWI is 1, check CI.
    a. If CI is 0, compare ISN+1 in primary memory with ACK number.
        i. If ISN+1 = ACK, send the flow ID to the filter array module, and clear the entry
        ii. If not, drop the packet
    b. If CI is 1, look in both primary and collision memory to find a match
        i. If a match is found in either memory, give the flow ID to Filter Array module. If the match is found in collision memory, and clear the entry in collision memory.
        ii. If a match is not found, drop the packet

In parallel with packet processing, the timers both in primary memory and collision memory are constantly updated with following actions.

1. Decrement the timers by 1 in every second
2. If a timer becomes 0, clear the associated table entry.

If an ACK packet does not arrive within the timeout period, the corresponding ISN is cleared. In case of collision, the timer in the primary memory always keeps the timeout from the last collided entry, so no ISN gets deleted before the timeout.

3.2 Filter Array

Filter array has one hash table composed of 3 fields, the Pass/Drop indicator (PDI), timer and timeout type as shown in Figure 5. When the filter array module receives a flow ID from the connection verifier module, it applies its own hash function to the flow ID and sets the corresponding PDI to 1 making it *enabled*. The timer indicates the maximum inter-packet arrival time, and it is set to maximum whenever a packet is sent or received. After the timeout, the flow is considered terminated and the PDI is *disabled* by making it 0. The timeout type indicates different timeout values. The timeout type is determined from the destination port number in the flow ID that is associated with a protocol type. For example, telnet or ftp control traffic can have longer timeout values than HTTP traffic. If a new flow ID is mapped into the same hash value where the PDI is already 1, the timeout type with larger value is assigned.

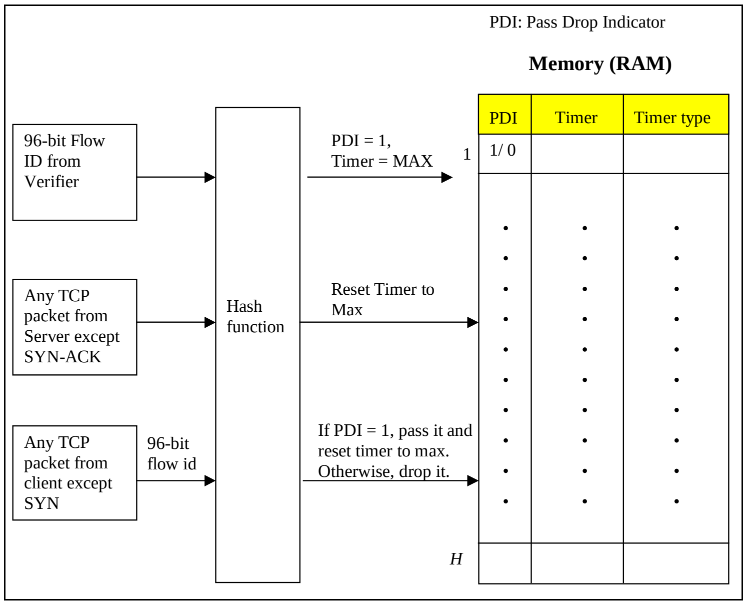

PDI: Pass Drop Indicator
Memory (RAM)
PDI
Timer
Timer type
1/ 0
1
H
96-bit Flow ID from Verifier
Any TCP packet from Server except SYN-ACK
Any TCP packet from client except SYN
96-bit flow id
Hash function
PDI = 1, Timer = MAX
Reset Timer to Max
If PDI = 1, pass it and reset timer to max. Otherwise, drop it.

Figure 5: Filter array architecture

Upon receiving a packet from the client side, the following actions are taken.

1. 96-bit flow ID is hashed into hash table index (source and destination fields are switched)
2. If PDI is 0, drop the packet.
3. If PDI is 1, pass the packet, and set the timer to max value as indicated by the timeout type

Upon receiving a packet from the server side, the following actions are taken.

1. 96-bit flow ID is hashed into hash table index
2. If PDI is 1, set the timer to max value as indicated by the timeout type

In parallel with packet processing, the timer is updated as following.

1. Read the timer and timeout type in every second for each entry
2. Decrement the timer by the unit time as indicated by the timeout type
3. If the timer became 0, disable the PDI and set the timer to 0

ACK packets are processed in both verifier and filter array because we don't know whether the ACK is a part of 3-way handshaking or normal data packet. In either case, a legitimate packet is always forwarded properly.

3.3 Multiple Hash Functions for Filter Array

Attack packets commonly use randomly generated spoofed source address and port numbers. If their hash values match the hash table index with active PDIs, they will pass the filter array. This probability can be greatly reduced by utilizing multiple hash functions [7] because a flow ID must satisfy all the PDIs in the multiple hash tables in order to pass. For multiple hashing, the same structure is duplicated by the desired number of hash functions. For each hash table, a different hash function must be used to get different hash index from the same flow ID. Figure 6 illustrates the architecture for dual hashing. Each hash table entry has own timer and it expires independently.

Given the same hash table size, the chance of accidental pass becomes smaller when the hash table is divided into smaller ones. Let *H* be the aggregate hash table size and *s* be the number of active PDIs. Then the probability of accidental pass under single hashing is s/*H*. When the hash table is divided into *N*, the size of each hash table is *H/N*, and the number of active PDIs in each table is about *s*. (we ignore collision for now). Then the probability of accidental match under *N* tables is $\left(\frac{s}{H/N}\right)^N = \left(\frac{Ns}{H}\right)^N$. When $s < \frac{H}{N^{\frac{N}{N-1}}}$,

$\left(\frac{Ns}{H}\right)^N < \left(\frac{s}{H}\right)$ holds. For example, when the rate of active PDIs is less than 25% of the hash table size, 2-hashing has smaller chance of accidental pass than single hashing. When collision is considered, the *s* becomes smaller as *N* grows, so the accidental pass rate becomes even smaller, which is beneficial for blocking attack packets.

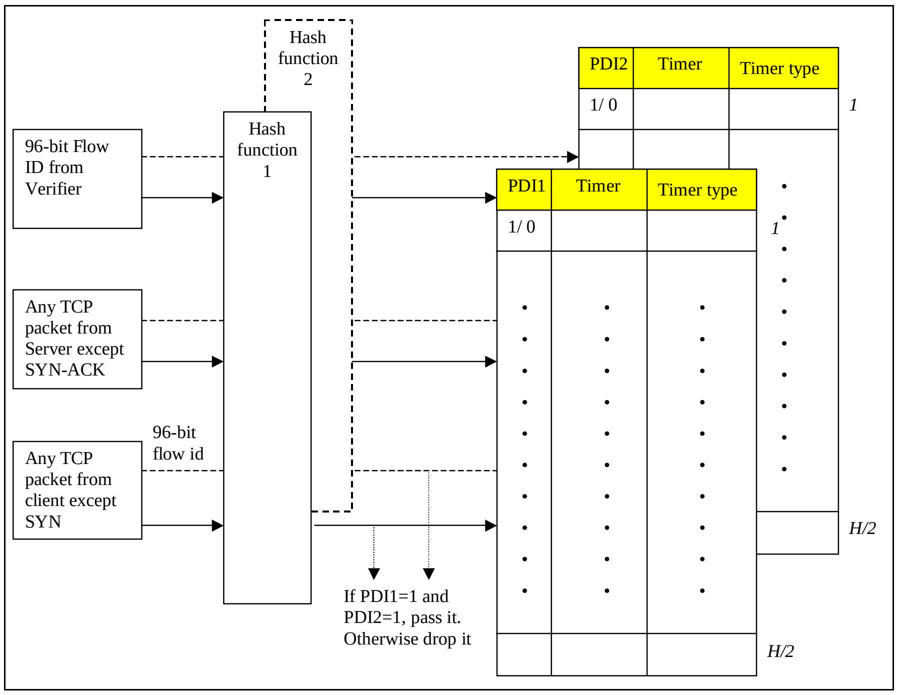

Hash function 2
Hash function 1
96-bit Flow ID from Verifier
Any TCP packet from Server except SYN-ACK
Any TCP packet from client except SYN
96-bit flow id
PDI2
Timer
Timer type
1/ 0
1
PDI1
Timer
Timer type
1/ 0
1
H/2
H/2
If PDI1=1 and PDI2=1, pass it. Otherwise drop it

Figure 6: Multiple hashing with 2 hash functions

4 Design Considerations

We design the filter for high-speed routers with 10 Gbps port speed. The following aspects are considered to ensure proper packet processing at such speed.

4.1 Hash Table Size

For connection verifier and filter array, different aspects should be considered for deciding the hash table size. For connection verifier, the maximum number of connection requests, especially during SYN flooding attack, should be considered. For the filter array, both the anticipated number of legitimate flows and the targeted attack traffic blocking rate should be considered. The size of hash tables for connection verifier and filter array are independently determined, and different sizes can be assigned.

4.1.1 Connection Verifier

If the desired AWI occupancy rate is *r*, the number of anticipated SYN-ACK packets per second is *s*, and the average reply time of ACK is *a*, the hash table size *H* is determined by,

$$H = \frac{s \times a}{r}$$

For example, in our empirical study [15], the number of flows in OC-12 link (with 450 Mbps utilized) is about 300,000 flows for 90 seconds, which translates into 3,333 flows/sec. In OC-192 speed, it would be 73,326 flows/sec. If most of the ACK packets from clients come back within 1 second, less than 73,326 AWI will be active. With a targeted AWI occupancy rate of 1%, a hash table size 2^{23} (8M) will be sufficient. The targeted occupancy rate of AWI should be kept low to make the collision rate low in order to reduce the collision memory requirement.

Under a normal traffic condition, a relatively small hash table may be enough because the reply ACKs from clients come back quickly and not many AWI bits are active. But during SYN flooding attack, more number of AWIs become active, so larger memory is needed. SYN flooding attack increases the active AWI ratio in two aspects. First, it increases the amount of SYN-ACK packets. Second, there is no reply for most of attack SYN-ACK packets, so they occupy the memory until timeout. Here is an extreme example of a SYN flooding. Let’s assume that a group of servers are connected to Internet via OC-3 link (155 Mbps) and they are under SYN flooding attack to an open TCP port. Theoretically, they could reply with SYN-ACK packets filling up the whole uplink bandwidth of 155 Mbps, generating about 480,000 SYN-ACK packets/second. (In reality, the reply rate is much lower, between 500 and 14,000 packets/second[19].) Some of them may be responses for legitimate SYN packets, but let’s ignore it. Not all of the SYN-ACK packets in reply to attack may reach the filter because the destination may not be reachable for some packets due to spoofed addresses. If 50% of the SYN-ACK packets survive through the router, i.e., 240,000 packets/second, the number of ISNs that should be kept in the filter until timeout of 3 seconds will be 720,000.

Above example shows that a SYN flooding attack over OC-3 can easily fill up the hash table as in OC-192 in an extreme case. If the hash table and the collision memory are not big enough, the ISNs from legitimate SYN-ACK packets may not get recorded, and their ACKs get denied. Although over-engineering might be an obvious answer, just placing the filter at a point where traffic is sufficiently aggregated can avoid such a problem. For example, the filter placed at OC-192 link will not be affected greatly by a SYN flooding attack to a server connected at OC-3 link.

4.1.2 Filter Array

SYN flooding attack is not a concern for filter array because only the flow IDs that passed the verification process are given to Filter Array. The size of the hash table in filter array is determined by the desired attack blocking rate. Assuming the flows IDs of the attack packets are randomly distributed, the chance of passing the filter accidentally for an attack packet is proportional to the number of enabled PDIs.

For single hashing, given the desired blocking rate *(1- r)*, and the anticipated number of enabled PDIs *f*, the size of hash table *H* can be determined by

$$H = \frac{f}{r}$$

When *r* is sufficiently small, the collision effect can be ignored. For *N*-hashing, the hash table size is,

$$H = \frac{fN}{r^{\frac{1}{N}}}, \qquad \text{because } r = \left(\frac{fN}{H}\right)^N$$

As mentioned in 4.1.1, the expected flow numbers per second in 10Gbps link is 73,326. Assuming each PDI is enabled for 9 seconds (6 second flow life + 3 second timeout), maximum of 659,934 PDIs are enabled at a moment. For desired blocking rate of 99%, the hash table size is about 66M (=659,934/0.01). Multiple hashing decreases the memory requirements. With two hash functions, 13.2M-entry hash table is required and with four hashing, 8.3M-entry is required.

Unlike in Verifier, collision is not a problem in Filter Array because the legitimate flows can be funneled into the same hash value without interfering each other. Since the collision reduces the number of enabled PDIs slightly, it rather enhances the blocking efficiency of bad traffic. So above formulae give us a conservative number for the hash table size.

4.2 Memory Requirements

The hash table size and the number of bits per hash table entry determine the memory size. For the connection verifier, each entry in primary memory takes 20 bits in our architecture with 1-bit AWI , 1-bit CI 16-bit ISN and 2-bit timer. For hash table size of 8M, 20 Mbytes of memory is required. For the filter array, each entry takes 7 bits with 1-bit PDI, 4-bit timer and 2-bit timeout type. For hash table size of 8M, 7 Mbytes of memory is required. So the combined memory size will be 27 Mbytes for RAM. The size of collision memory depends on the amount of anticipated collision occurrences, which is again dependent on the hash table size and number of anticipated active AWIs at a moment.

4.3 Processing Speed

In our target bandwidth of 10 Gbps, the smallest packet of 40 bytes can arrive with the interval of 32 *ns* (nanoseconds). With the proposed architecture, all the required procedures can be accomplished within the 32 *ns* limit under conservative assumptions on the memory and processing units. We assume that the typical memory access time including CAM is under 10 *ns*, and CPU operation and hashing can be done within 5 *ns*. The estimated processing times are summarized in Table 3. The longest required time is 30 *ns*, which is within the allowed 32 *ns*. Therefore all the packets can be processed without buffering.

Writing SYN-ACK		**Verifying ACK**		**Timer update**	
Steps	Time consumption (*ns*)	Steps	Time consumption (*ns*)	Steps	Time consumption (*ns*)
Hash, ISN = ISN+1	5	Hash	5	Read	10
Read AWI & CI	10	Read AWI, CI and ISN, CAM	10	Decrement	60 (16 timers)
Write ISN and CI	10	Compare ISN	5	Write AWI, CI, ISN	40 (assuming all 4 expired)
		Write AWI, ISN. Send flow id to Filter array	10	Write timer back	10
Total	25	Total	30		120

Table 3: Processing time in connection verifier

Timer update depends on the size of hash table. Assuming 8 M for hash table size, 2-bit timer and 32-bit wide memory access, a total of 500,000 operations per second or 2 microseconds per operation is allowed. The worst-case requirement of 120 *ns* can be easily accommodated within the allowed time. Similarly, all the processing in the filter array can be done within 32 *ns*.

5 Deployment

Since this filter requires monitoring traffic in both ways, it can't be placed where the packets do not follow a unique path or where the forward and reverse paths may be different. Currently the Internet routing protocols do not guarantee the same forward and reverse path, so the filter must be located at a unique choking point. For that purpose, POP is considered a best location because many customers are uniquely connected to a POP and it aggregates many downstream links. Figure 7 shows an example of a filter deployment at a typical POP[13]. In this example, all the customers are connected as single-homed and the traffic from the POP is merged to a single link to the network, so the filter is placed to the backbone-side port of the edge router. If the POP is connected to multiple backbone links, the filter should be placed on the server side ports of the edge router.

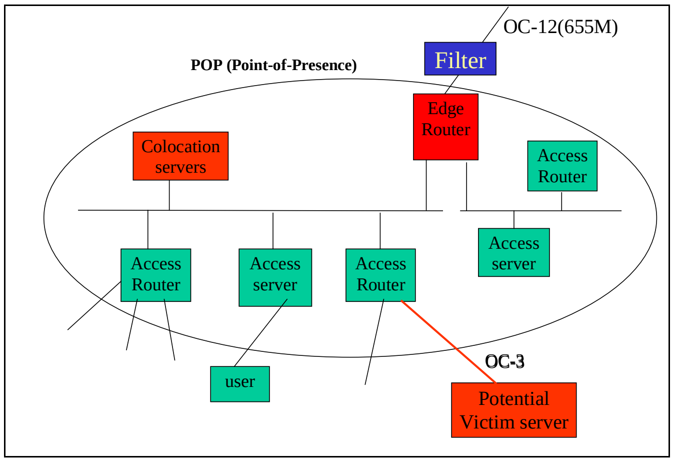


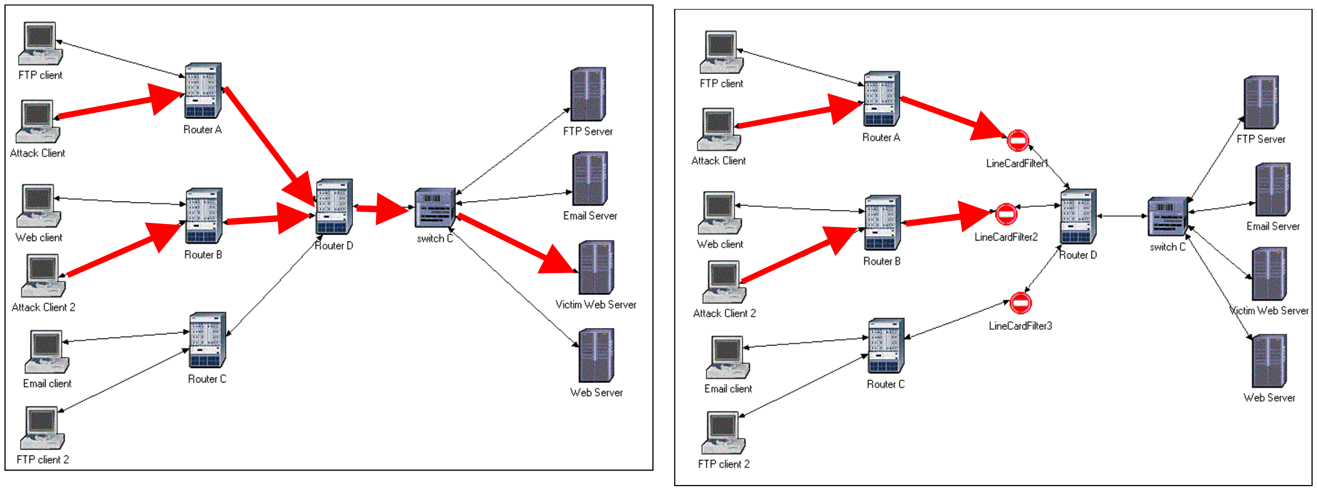
Figure 7: Deployment for single-homed environment

In case some of the customers are multi-homed, the forward and reverse paths may be different for them and the filter may drop all the packets destined for them. The first solution is per-destination screening. If the destination of a packet is for a multi-homed customer, the filter can simply bypass such a packet. But this method may increase the implementation and operation complexity, and cannot provide the protection from a DDoS attack for multi-homed customers. The second solution is configuring the multi-homed customers' network to transmit duplicate SYN-ACK packets toward the filter, preferably with a small TTL number so that it can be dropped after passing the filter. Then the filter can be properly configured and will not drop legitimate packets. However, the timeout value may have to be increased to compensate the increased inter-packet arrival time.

6 Simulation

6.1 *The Model*

To prove the concept, simulation was performed using OPNET simulation package. In Figure 8, there are 6 clients, out of which 2 are attack agents. There are 4 servers providing service in TCP. One of the servers is the target victim server. During the attack period of 25 seconds, the attack agents flood TCP packets with spoofed source addresses and port numbers. All the links are 10 Mbps Ethernet. The duration of simulation is 60 seconds. The total number of normal packets is about 12,000, and the number of attack packets is 100,000. During the 60-second simulation time, about 3,100 flows are observed. We use hash table size of 65,536 (2^{16}) and CRC-32 for hashing algorithm.



a) Attack scenario b) Attack scenario with filter

Figure 8: Simulation setup

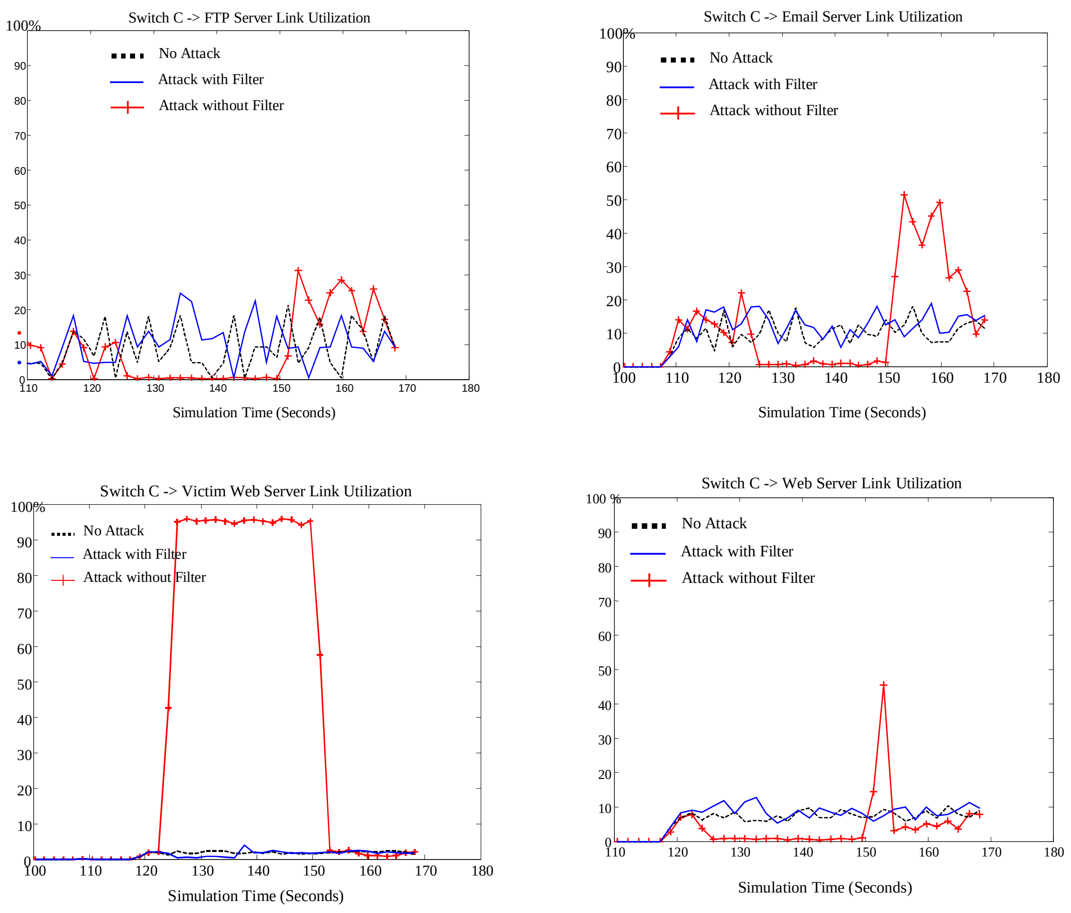
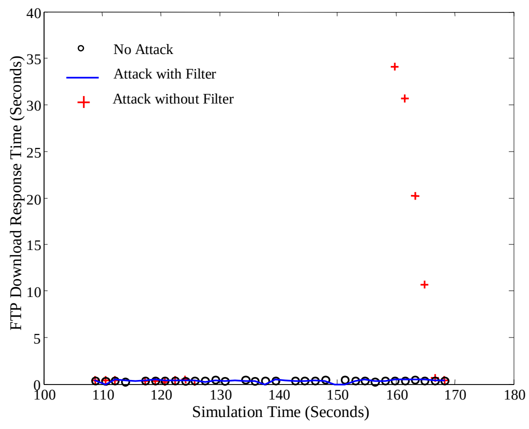


Figure 9: Link utilization

Figure 9 shows the bandwidths of the server links for 3 cases, non-attack condition, attack condition, and attack condition with the filters. In the attack condition, the bandwidth for the victim server is exhausted as well as the link between the router *D* and switch *C* is also exhausted during the attack period (125 to 150 seconds). During the attack, the delivered traffic to other servers is significantly reduced. After filter is installed, the traffic to all the servers becomes normal.

Figure 10 shows the TCP response times. During the attack period, the response time is extreme long. But after the filters are installed, the response time for legitimate flows becomes normal.



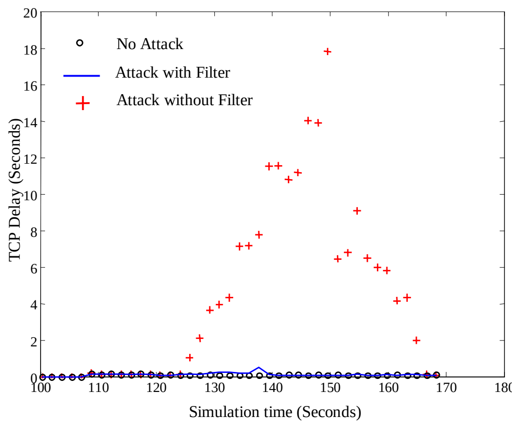
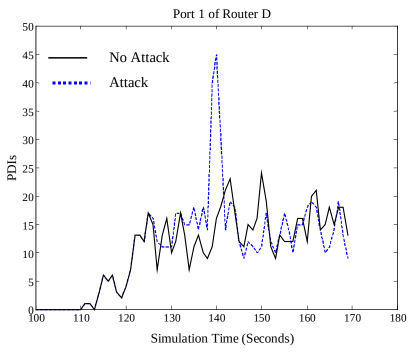
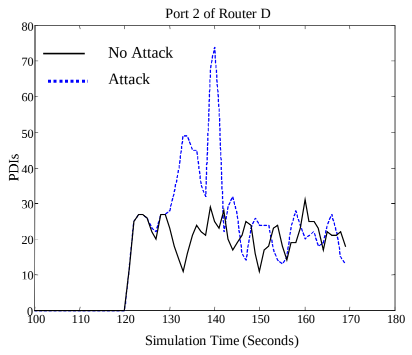


Figure 10: Response time for legitimate traffic

6.2 Attack Blocking Efficiency

Figure 11 shows how many PDIs are active among 65,536 PDIs in filter array. It is about 20 on the average, which suggests blocking rate of (1 - 20/65,536) or 99.9695%. Indeed the measured blocking rate in Table 4 reflects the calculated value.





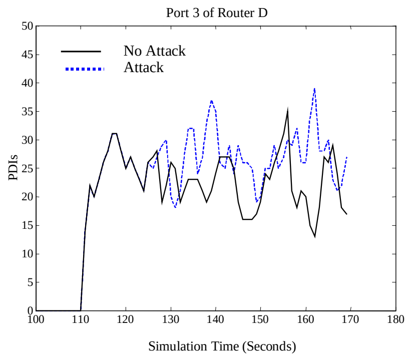


Figure 11: Number of active PDIs

Number of attack packets	Number of dropped packets with 1 hash function	Number of dropped packets with 2 hash functions
50,000	49,980 (99.96%)	50,000 (100%)
100,000	99,961 (99.96%)	100,000 (100%)
150,000	149,943 (99.96%)	150,000 (100%)
200,000	199,929 (99.96%)	200,000 (100%)

Table 4: Number of dropped attack packets

7 Discussions

7.1 Related Work

The concept of rejecting the TCP packets that are not part of established connection is used in Shoreline Firewall[21], which is free software. Some firewalls, such as Riverhead firewall, can be placed at ISPs' router site, blocking undesirable packets before they congest the link to the customer. The concept of Nozzle[24] is close to our architecture in the sense that it detects legitimate flows and blocks other traffic. In comparison with them, our filter is a hardware-based solution suitable for high-speed processing.

7.2 Possible Enhancements

In the filter array, the PDI is disabled only when the timer expires in current design. To make the enabled time for PDI shorter, we can utilize the TCP connection termination procedure. To avoid breaking an active flow when multiple flows are hashed into the same hash index, we can use reference counter. When the reference counter becomes 0, the PDI for that hash value can be set to 0.

The memory utilization in connection verifier is kept very low in order to avoid collision. But if high-speed processing is not strictly required, e.g. by using packet buffer or for slower speed links, the unused memory may be utilized, reducing the memory requirement.

8 Conclusions

The DDoS attack is a serious threat in Internet. Current solutions are either ineffective or inapplicable immediately. We propose an algorithm and implementation that can be readily deployed in edge routers. It protects the TCP bandwidth for potential victim servers, which is sufficient for most servers in Internet. Our architecture allows fast hardware processing at 10 Gbps speed, and reliably blocks only the malicious TCP packets. We have conducted a simulation and the results confirm the high efficiency of the proposed filter architecture.

REFERENCES

[1] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo, “DOS-resistant Authentication with Client Puzzles,” Lecture notes in Computer Science Series, April 2000, Springer.

[2] Jose Carlos Brustoloni, "Protecting Electronic Commerce From Distributed Denial-of-Service Attacks," In *Proceedings of ACM WWW*, May 2002.

[3] CERT Advisory, “TCP SYN Flooding and IP Spoofing Attacks,” www.cert.org/advisories/CA-1996-21.html, Nov. 2000.

[4] Y. W. Chen, “Study on the Prevention of SYN Flooding by Using Traffic Policing,” In *Proceedings of Network Operations and Management Symposium*, 2000.

[5] Cisco IOS Security Configuration Guide, “Configuring TCP Intercept (Preventing Denial-of-Service Attacks),” pp. SC-213 ~ SC-218, http://www.cisco.com.

[6] Cisco IOS Security Configuration Guide, “Configuring Unicast Reverse Path Forwarding,” pp. SC-429 ~ SC-446, http://www.cisco.com.

[7] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder, “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol,” *IEEE Transactions on Networking*, Vol. 8, No. 3. pp. 281-293, June 2000.

[8] Sally Floyd, Steven M. Bellovin, John Ioannidis, Kireeti Kompella, Ratul Manajan, and Vern Paxson, “Pushback Messages for Controlling Aggregates in the Network,” *IETF Internet Draft*, *http://www.ietf.org/internet-drafts/draft-floyd-pushback-messages-00.txt*, July 2001.

[9] Chun-Kan Fung and M.C. Lee, “A Denial-of-Service Resistant Public-key Authentication and Key Establishment Protocol,” In *Proceedings of IEEE International Performance, Computing, and Communications*, 2002.

[10] Lee Garber, “Denial-of-Service Attacks Rip the Internet”, *IEEE Computer*, April, 2000, pp. 12-17

[11] A. Garg and A. L. Narasimha Reddy, "Mitigation of DOS attacks through QOS regulation", in *Proceedings of IWQOS Workshop*, May 2002.

[12] Yih Huang and J. Mark Pullen, “Countering Denial-of-Service Attacks Using Congestion Triggered Packet Sampling and Filtering,” In *Proceedings of Computer Communications and Networks*, 2001.

[13] Geoff Huston, *ISP Survival Guide, Strategies for Running a Competitive ISP*, Wiley Computer Publishing, New York, NY 1999.

[14] John Ioannidis and Steven M. Bellovin, “Implementing Pushback: Router-Based Defense Against DDoS Attacks,” In *Proceedings of Network and Distributed System Security Symposium*, Feb. 2002.

[15] Ju-Yeon Jo, Yoohwan Kim, H. Jonathan Chao, and Frank Merat, “Internet Traffic Load Balancing using Dynamic Hashing with Flow Volume,” In *Proceedings of SPIE ITCom 2002*, July 2002.

[16] Dai Kashiwa, Eric Y. Chen and Hitoshi Fuji, “Active Shaping: A Countermeasure against DDoS Attacks,” In *Proceedings of European Conference on Universal Multiservice Networks*, 2002.

[17] David Mankins, Rajesh Krishnan, Ceilyn Boyd, John Zao, Michael Frentz, “Mitigating Distributed Denial of Service Attacks with Dynamic Resource Pricing”, In *proceedings of 17th Annual conference on Computer Security Applications*, 2001.

[18] Jack May, Jim Peterson, John Bauman, “Attack Detection in Large Networks”, In *Proceedings of DARPA Information Survivability Conference and Expo II*, 2001.

[19] David Moore, Geoffrey M. Voelker and Stefan Savage, “Inferring Internet Denial-of-Service Activity,” In *Proceedings of 10th USENIX Security Symposium*, Aug. 2001.

[20] Kihong Park and Heejo Lee, "On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets," In *Proceedings of ACM SIGCOMM*, Aug. 2001.

[21] Shoreline Firewall, “*Shorewall 1.3*”, http://www.shorewall.net

[22] Dan Sterne, et. al., "Active Network Based DDoS Defense," In *Proceedings of the DARPA Active Networks Conference and Exposition*, 2002.

[23] W. Richard Steven, *TCP/IP Illustrated*, *The Protocols*, Vol. 1. Addison-Wesley Publishing, Reading, MA 1994.

[24] Elizabeth Strother, "Denial of Service Protection, The Nozzle," In *Proceedings of Computer Security Applications*, pp. 32-41, 2000.

[25] Yong Xiong, Steve Liu, and Peter Sun, "On the Defense of the Distributed Denial of Service Attacks: An On-Off Feedback Control Approach," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, Vol. 31, No. 4. pp. 282-293, July 2001.

[26] David K. Y. Yau, John C. S. Lui, and Feng Liang, “Defending Against Distributed Denial-of-Service Attacks with Max-min Fair Server-centric Router Throttles,” In *Proceedings of IEEE International Workshop on Quality of Service*, pp. 35-42, 2002.