COUNTERING DISTRIBUTED DENIAL OF SERVICE ATTACKS

by

Haining Wang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2003

Doctoral Committee:

Professor Kang G. Shin, Chair
Associate Professor Peter M. Chen
Professor Atul Prakash
Professor Demosthenis Teneketzis

To my parents

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Kang G. Shin, for his guidance, support, and enormous patience through the past five years. Many times, he helped me step out of frustration and keep moving forward. I have benefited tremendously from his vision, technical insights, and, most importantly, strong pursuit of excellence. I would like to thank Dr. Chia Shen for giving me the opportunity to spend one summer with her group at MERL. I would also like to thank Professors Peter M. Chen, Atul Prakash and Demosthenis Teneketzis for serving on my advisory committee.

I would like to express my sincere thanks to my collaborators during the past few years: Abhijit Bose, Mohamad El-Gendy, Cheng Jin, Danlu Zhang, Taejoon Park and Min-Gyu Cho. My gratitude also goes to all my friends and colleagues in the Real-Time Computing Laboratory. Special thanks to my roommates Daji Qiao, Zonghua Gu and Jian Wu for their brotherhood, and to my officemates or DiffServ group members: Hai Huang, Hani Jamjoom, Songkuk Kim, Sung-Whan Moon, Babu Pillai, John Reumann, Wei Sun and Chang-Hao Tsai for their friendship and interesting technical and random-topic discussions. Finally, I would like to thank my family for their never-ending love and support.

This research was supported in part by the US Office of Naval Research under Grant N00014-99-1-0465.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

CHAPTER I

INTRODUCTION

1.1 Motivation

Over the past two decades, the Internet has grown from a small network connecting dozens of researchers to a gigantic global network connecting people all over the world. The tremendous success of the Internet lies in its scalability and robustness — outcomes of its so called "end-to-end design principle". The essence of end-to-end design principle is to keep the functionality inside the core of the network as simple as possible, while deploying complex functionalities at the network end-points. Since the Internet was designed with adding connectivity, not security, in mind, the Internet infrastructure has exposed its weakness to recent DoS (Denial of Service) attacks [22, 75].

The TCP/IP protocol suite is the basic component underlying the Internet. The IP (Internet Protocol) is the network layer of the Internet. The basic functionality of IP is to route and forward a packet to its destination. Within the OSI (Open System Interconnection) model, IP is at layer three and provides a best-effort unreliable delivery service. TCP (Transmission Control Protocol) runs on top of IP, supporting a number of "ports" on each end-host running IP. TCP provides reliable data delivery, and performs congestion and flow control between two communicating ports. However, the TCP/IP protocols lack the basic mechanisms for security: authentication or encryption. IP routing is stateless and destination-based. IP routers have minimal or no mechanism to authenticate the true origin of a packet, and IP packets are forwarded to the next hop without source address authentication.

The Internet is an open system, so there is no strict access control upon network resources. An end-host with a valid IP address and connection to the Internet, can inject arbitrary IP packets into the Internet. Due to the security weakness mentioned above, the Internet is susceptible to DoS attacks [22, 75], in which an attacker exploits IP spoofing [76] and simply floods forged packets to the victim site. Instead of subverting Internet services, DoS attacks limit and block legitimate users' access by exhausting victim servers' resources [23], or saturating stub networks' access links to the Internet [45] with a flood of spoofed traffic. Flooding packets is the most common DoS attack, due to the readily available tools and its simple nature. The difficulties in countering DoS attacks are: (1) no matter how secure a victim server may be, it could be bombarded by DoS attacks simply because it is connected to the Internet; and (2) its susceptibility to DoS attacks relies on the state of security in the rest of the Internet. As one of the most difficult problems in network security, DoS attacks have become a serious threat to the availability of Internet services [22, 44, 75].

The recent occurrences of Distributed DoS (DDoS) attacks make it even more difficult for victims to block and trace back the attacking sources. In typical DDoS attacks, instead of flooding spoofed traffic from a single source, attackers compromise a number of machines (slaves) and orchestrate them to simultaneously flood a victim site. The utilization of numerous slaves both strengthens the force of the attack and complicates defense against it: the dilution of locality in the flooding traffic confuses the victim not to filter out bogus packets and trace back the flooding sources. The vulnerability of the Internet to DDoS attacks has been witnessed by the recent attacks on popular web sites like Yahoo, eBay and E*Trade, and their resultant disruption of services [44]. These attacks against high-profile web sites highlight how devastating DDoS attacks are, and how defenseless the Internet is.

1.2 Description of DDoS Attacks

A DDoS attack system can usually be described as a hierarchical model, in which an attacker controls a master (handler) that, in turn, dictates the hordes of slaves (agents) to flood the victim with attacking traffic. The communication between the attacker and the master, as well as between the master and the slaves is called the *control traffic*, while the communication between the slaves and the victim is called the *flooding traffic*. To hide the attacker from its detection, the control traffic is encrypted and its channel is covert and password-protected. The slaves are recruited by employing automatic scanning and propagation techniques, which search for security holes and inject the attacking instructions into the compromised hosts. The hierarchy of the DDoS attack system is shown in Figure 1.1.

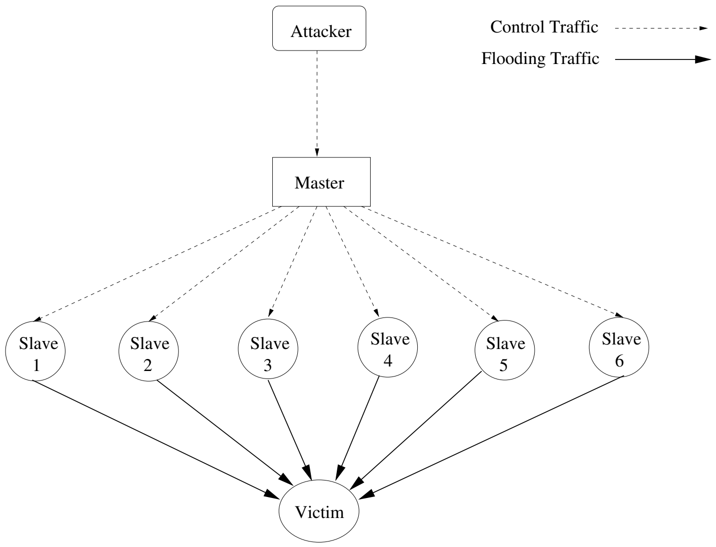


Figure 1.1: Structure of a DDoS attack

The mechanism of DDoS attacks works as follows: the master sends control packets to the previously-compromised slaves, instructing them to target a given victim. The slaves then generate and send high-volume streams of bogus traffic to the victim, but with fake or randomized source addresses, so that the victim cannot locate the attackers. DDoS attacks are difficult to defend against, as they do not target specific vulnerabilities of a computer system, but rather, the very fact that it is connected to the network. With the appearance of Trinoo, which only implements UDP flooding attacks, many tools have been developed to create DDoS attacks. These readily available attacking tools, such as Tribe Flood Network (TFN), TFN2K, Trinity, Plague, Shaft and Stacheldraht, generate various flooding attacks [33].

While the conventional flooding attack is a system resource depletion attack, the recent Distributed Reflection DoS (DRDoS) attacks [45, 86] virtually “disconnect” a victim server from the Internet by hogging the link bandwidth between the victim and its ISP with an excessive number of response packets (a.k.a. bandwidth depletion attack). The DRDoS attacks masquerade the source IP address of each spoofed request with the victim’s IP address, and spray the spoofed requests to a large number of Internet servers, which are exploited as reflectors. The reflectors will then send combined replies to the victim, redirecting and amplifying the flooding traffic.

1.3 Overview of Existing Approaches

To thwart DDoS attacks, researchers have taken two distinct approaches: *router-based* and *victim-based*. Router-based approaches install defense mechanisms at IP routers to detect, block and trace back attacking traffic. In contrast, victim-based approaches enhance the resilience of Internet servers against attacks by fine-grained resource management or resource savings for each request.

1.3.1 Router-based Defense Mechanisms

The router-based approach performs either on-line filtering of DDoS traffic inside routers or off-line analysis of flooding traffic. On-line filtering mechanisms rely on IP router enhancements [40, 46, 60, 63, 66, 72, 84, 120] to detect abnormal traffic patterns and block DDoS traffic. Off-line IP traceback [14, 21, 31, 95, 99, 100, 105] attempts to establish procedures to track down flooding sources *after* occurrences of DDoS attacks. While it does help pinpoint locations of flooding sources, off-line IP traceback does not help sustain service availability during an attack.

On-line Dectection and Filtering

As on-line detection mechanisms, MULTOPS [46] and D-WARD [72] are the two typical examples in this field. MULTOPS [46] is a tree of nodes that keep packet-rate statistics for subnets at different aggregation levels. Based on the observation of a significant disproportional difference between the traffic flowing into and out of the victim, routers use MULTOPS to detect ongoing bandwidth attacks. D-WARD [72] is deployed at source-end networks, and monitors two-way traffic between the network and the rest of the Internet. Based on the comparison with normal traffic models, D-WARD detects and throttles the ongoing flooding attacks. Its normal traffic models are simply based on flow rates. However, due to the diversity of user behaviors and the emergence of new network applications, it is difficult to obtain a robust and general model for describing the normal traffic flow rates.

Among various filtering schemes [40, 63, 84] executing on IP routers, the most straightforward scheme is ingress filtering [40]. It blocks spoofed packets at edge routers, where address ownership is relatively unambiguous, and traffic load is low. Ingress filtering [40] configures the internal router interface to block spoofed packets whose source IP addresses do not belong to the stub network. This limits the ability to flood spoofed packets from that stub network, since the attacker would only be able to generate bogus traffic with internal addresses. However, the success of ingress filtering hinges on its wide-deployment in IP routers. Most ISPs are reluctant to implement this service due to administrative overhead and lack of immediate benefit to their customers.

Given the reachability constraints imposed by the routing and network topology, the route-based distributed packet filtering (DPF) [84] exploits routing information to determine if a packet arriving at the router is valid with respect to its inscribed source/destination addresses. The experimental results reported in [84] show that a significant fraction of spoofed packets are filtered out, and the spoofed packets that escaped the filtering can be localized to five candidate sites which are easy to trace back.

To validate that an IP packet carries the true source address, SAVE [63], a Source Address Validity Enforcement protocol, builds a table of incoming source IP addresses at each router that associates each of its incoming interfaces with a set of valid incoming network addresses. SAVE runs on each IP router and checks if each IP packet arrives at the expected interface. By matching incoming IP addresses with their expected receiving interfaces, the set of IP source addresses that any attacker can spoof are greatly reduced.

Yau *et. al* proposed a *router throttle* mechanism [120], which is installed at the routers that are close to the victim. These routers proactively regulate the incoming packets to a moderate level, thus reducing the amount of the flooding traffic towards the victim. The key idea of *pushback* is close to that of router throttle, and it identifies and controls high bandwidth aggregates in network [55, 66]. The router could then ask adjacent upstream routers to limit the amount of traffic from the identified aggregate. This upstream rate-limiting is called *pushback* and can be propagated recursively to routers further upstream.

IP Traceback

Since the source addresses of flooding packets are faked, it is challenging to find the origin of a flooding source. To identify the compromised end-hosts that directly generate flooding packets and the network path that these packets subsequently follow, various traceback techniques [14, 20, 31, 95, 99, 100, 105] have been proposed. A typical IP traceback mechanism has routers mark IP address information on packets en-route to the victim, which collects this information and uses it to reconstruct the path to the flooding source.

The ICMP traceback mechanism [14] proposed a new type of ICMP message: traceback messages. For each packet to be forwarded, with a low probability, an *itrace* router generates an ICMP message to the destination address of the packet, including the IP address of the router as well as the IP addresses of the previous and next-hop routers. During a flooding attack, the victim can utilize these ICMP traceback messages to reconstruct the path to the flooding source. However, the overhead of ICMP traceback is relatively high, and there is a tension between fast path reconstruction and low network overhead incurred by ICMP traceback messages.

To reduce the tracing overhead, PPM (Probabilistic Packet Marking) [95] was proposed to embed traceback information within an IP packet. Routers probabilistically mark packets with partial path information into one rarely used IP header field — 16-bit IP identification field. Compressed edge fragment sampling is used to embed the 72-bit edge information into the 16-bit field. Fragments are reassembled at the victim to reveal all the IP addresses of the upstream routers towards the flooding source. Compared with ICMP traceback, this scheme can trace back lower-volume flows with less overhead. However, with the increase of flooding sources, the victim runs into computational difficulties caused by fragment reconstruction. This problem was addressed by Song and Perrig [100]. Their scheme encodes routing information more efficiently and accurately with the use of network topology maps. Moreover, this scheme authenticates the routers' markings so that even a compromised router cannot forge or tamper with other legal markings.

The controlled flooding technique developed by Burch and Cheswick [21] infers the attacking path by observing the impact of selectively exhausting some network links upon the victim. Since router buffers are shared, packets across the loaded link are more likely to be dropped. If the attacking packets traverse a bandwidth-exhausted link, the victim will receive less DoS traffic. By monitoring the changes in the rate of attacking traffic, it is possible for the victim to know the links that the DoS traffic are traversing. However, this scheme does not work well if there are many flooding sources in a DDoS attack. Also, it requires the victim to have a good Internet topology map and a list of "willing" hosts, in order to exhaust the related links' bandwidths. Finally, this approach is only effective at tracing on-going attacks.

Using noisy polynomial reconstruction, Dean *et. al* [31] proposed an algebraic approach to performing IP traceback. The marks in packets are the values of linear polynomial with the router's identity as its leading coefficient. The victim runs polynomial interpolation with noise on the received packets to trace out the attacking path. Snoeren *et. al* presented a hash-based IP traceback [99], which can track the origin of a single packet delivered by the network in an efficient and scalable way. The Source Path Isolation Engine (SPIE) records sets of hashes of packets traversing a given router. A victim can then find out the path of a given packet by querying routers for the set of hashes corresponding to the packet. Stone [105] built an IP overlay network for tracking DoS floods, which consists of IP tunnels connecting all edge routers. The topology of this overlay network is deliberately simple, and suspicious flows can be dynamically rerouted across the tracking overlay network for analysis. Then, the origin of the floods can be revealed.

Based on IP traceback marking, Path Identifier (Pi) [119] embeds a path fingerprint in each packet so that a victim can identify all packets traversing the same path across the Internet, even for those with spoofed IP addresses. Instead of probabilistic marking, Pi's marking is deterministic. By checking the marking on each packet, the victim can filter out all attacking packets that match the path signatures of already-known attacking packets. Pi is effective even if only a half of the routers in the Internet participate in packet marking.

1.3.2 Victim-based Defense Mechanisms

Compared to the router-based approaches, the victim-based approaches have the advantage of being immediately deployable. Also, a potential victim has a much stronger incentive to deploy defense mechanisms than network service providers. Victim-based approaches protect Internet servers either by using sophisticated resource management schemes or by significantly reducing each request's resource consumption to withstand the flooding traffic.

Advanced resource-management schemes provide more accurate resource accounting, and fine-grained service isolation and differentiation [11, 17, 90, 101], for example, to shield interactive video traffic from bulk data transfers. However, without a mechanism to detect and discard spoofed traffic, spoofed packets will share the same resource principals and code paths as legitimate requests. While a resource manager can confine the scope of damage to the service under attack, it may not be able to sustain the availability of the service being attacked.

To defend TCP-based network services against SYN flooding attacks, a software agent, called *synkill* [96], has been developed for mitigating the impact of malicious SYN requests. Working in a LAN environment, *synkill* continuously monitors TCP's three-way handshake messages. If a SYN packet is not acknowledged in a certain amount of time, *synkill* will inject a matching RST packet to free the resources occupied by the illegitimate half-open connection. Moreover, based on network observation and administrator-provided input, *synkill* classifies IP source addresses, with a high probability, to have been spoofed or not.

As firewalls have been installed at almost all sites, several SYN flooding protection systems are available at these firewalls, such as SynDefender [65] and Syn proxying [4]. The firewall before the protected server plays a key role in protection mechanisms, which act on behalf of the server before a connection is actually established. It intercepts the TCP traffic between clients and the server, and maintains state for each TCP connection. The drawback of these approaches is the longer delay incurred to each TCP connection setup.

Syn cache [62] and Syn cookies [15] belong to the server-based mechanism. Syn cache also maintains states for each SYN request, but its state structure is much smaller. Syn cache employs a global hash table to keep the incomplete queued connections, instead of the per-socket linear list. The listen queue is split among hash buckets. In the Syn cookies mechanism, when the server receives a SYN packet, it responds with a SYN/ACK packet and the ACK sequence number calculated from the source address, source port, source sequence, destination address, destination port and a secret seed. Then, the server releases all states. If an ACK comes from the client, the server can recalculate the ACK sequence number to determine if it is a response to the previous SYN/ACK. If it is, the server can directly enter the "established" state and open the connection. So, the server removes the need to watch for half-open connections.

Syn cookies have been implemented as a standard component of Linux kernel. However, Syn cookies themselves lack the detection functionality. Syn cookies are activated when the server's backlog queue is full. Unfortunately, backlog queue length increases due often to legitimate flash crowds, instead of SYN flooding attacks. Without a proper detection mechanism, keeping Syn cookies active under legitimate flash crowds cannot mitigate the workload stress; on the contrary, the computational overhead incurred by Syn cookies only aggravates the shortage of CPU cycles of the server.

1.4 Primary Research Contributions

The backbone of this research is the development of a series of methodologies and mechanisms that work together to counter DDoS attacks. These mechanisms include transport-aware IP router architecture that provides fine-grained service differentiation and resource isolation, a *SYN-dog* for detecting SYN flooding attacks that is the most common DDoS attacks [75], a hop-count filter for weeding out spoofed packets at victim servers, and an IP *Easy-pass* for network-edge resource access control. The contributions of this dissertation are described below.

1.4.1 Transport-aware IP Routers: a Built-in Protection Mechanism

The lack of service differentiation and resource isolation by the current IP routers exposes their vulnerability to DDoS attacks [44]. Based on the concept of layer-4 service differentiation and resource isolation, where the transport-layer information is inferred from the IP headers and used for packet classification and resource management, we present a transport-aware IP (*t*IP) router architecture that provides fine-grained service differentiation and resource isolation among different classes of traffic aggregates. The *t*IP router architecture consists of a fine-grained Quality-of-Service (QoS) classifier and an adaptive weight-based resource manager. A two-stage packet-classification mechanism is devised to decouple the fine-grained QoS lookup from the usual routing lookup at core routers. The fine-grained service differentiation and resource isolation provided inside the *t*IP router is a powerful built-in protection mechanism to counter DDoS attacks, reducing the vulnerability of Internet to DDoS attacks. Moreover, the *t*IP architecture is stateless and compatible with the Differentiated Service (DiffServ) infrastructure. Thanks to its scalable QoS support for TCP control segments, the *t*IP router also supports bi-directional differentiated services for TCP sessions.

1.4.2 SYN-dog: Sniffing SYN Flooding Attacks

We present a simple and robust mechanism, called *SYN-dog*, to sniff SYN flooding attacks. The core of SYN-dog is based on the distinct protocol behavior of TCP connection establishment and teardown, and is an instance of the Sequential Change Point Detection [13]. To make the detection mechanism insensitive to site and access pattern, a non-parametric Cumulative Sum (CUSUM) method [19] is applied, thus making the detection mechanism robust, more generally applicable and its deployment much easier. The statelessness and low computation overhead of SYN-dog make itself immune to flooding attacks. The efficacy of SYN-dog is evaluated by extensive trace-driven simulations. The evaluation results show that SYN-dog has short detection latency and high detection accuracy. Moreover, due to its proximity to the flooding sources, SYN-dog not only sets alarm upon detection of ongoing SYN flooding attacks, but also reveals the location of the flooding sources without resorting to the IP traceback [14, 20, 31, 95, 99].

1.4.3 Hop-Count Filtering: An Effective Defense Against Spoofed DDoS Traffic

IP spoofing has been exploited by DDoS attacks to (1) conceal flooding sources and localities in flooding traffic, and (2) coax legitimate hosts into becoming reflectors, redirecting and amplifying flooding traffic. Thus, the ability to filter spoofed IP packets near victims is essential to their own protection as well as to their avoidance of becoming involuntary DoS reflectors. Although attackers can forge any field in the IP header, they cannot falsify the number of hops an IP packet takes to reach its destination. This hop-count information can be inferred from the Time-to-Live (TTL) value in the IP header. Using a mapping between IP addresses and their hop-counts to an Internet server, the server can distinguish spoofed IP packets from legitimate ones. Based on this observation, we present a novel filtering technique that is immediately deployable to weed out spoofed IP packets. Through analysis using network measurement data, we show that *Hop-Count Filtering* (HCF) can identify close to 90% of spoofed IP packets, and then discard them with little collateral damage. We implement and evaluate HCF in the Linux kernel, demonstrating its benefits using experimental measurements.

1.4.4 IP Easy-pass: Edge Resource Access Control

Providing real-time communication services to multimedia applications and subscription-based Internet access often requires sufficient network resources to be reserved for real-time traffic. However, the reserved network resource is susceptible to resource theft and abuse. Without a resource access control mechanism that can efficiently differentiate legitimate real-time traffic from attacking packets, the traffic conditioning and policing conducted at ISP (Internet Service Provider) edge routers cannot protect the reserved network resource from embezzlement. On the contrary, the traffic policing at edge routers aggravates their vulnerability to flooding attacks by blindly dropping packets. To counter such DQoS (Denial of Quality of Service) attacks, we develop a fast and lightweight IP network-edge resource access control mechanism, called *IP Easy-pass*, which blocks unauthorized access to reserved network resources at edge devices. We attach a unique *pass* to each legitimate real-time packet so that an ISP edge router can validate the legitimacy of an incoming IP packet very quickly and simply by checking its pass. We present the generation of Easy-pass, its embedding, and verification procedures. We implement the IP Easy-pass mechanism in the Linux kernel, analyze its effectiveness against DQoS attacks. Finally, we measure its overhead and performance.

1.5 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 presents a transport-aware IP (*t*IP) router architecture, which provides fine-grained service differentiation and resource isolation among different classes of traffic aggregates, can be utilized as a built-in mechanism to counter DDoS attacks. Chapter 3 describes *SYN-dog*, a simple and robust detection mechanism to sniff SYN flooding sources. Based on the distinct protocol behavior of TCP connection establishment and teardown, the SYN flooding detection is treated as an instance of the Sequential Change Point Detection. A non-parametric Cumulative Sum method is applied, making the detection mechanism insensitive to sites and access pattern. Chapter 4 develops a novel hop-count-based filter to weed out spoofed IP packets. The hop-count information is inferred from the Time-to-Live (TTL) value in the IP header. Using a mapping between IP addresses and their hop-counts to an Internet server, the server can distinguish spoofed IP packets from legitimate ones. Chapter 5 presents a fast and lightweight IP network-edge resource access control mechanism, called IP *Easy-pass*, to defend against DQoS attacks in the data plane. Each real-time packet is attached with a unique *pass*. The packets with no pass or stale passes are discarded, which prevents unauthorized access to reserved network resources at edge devices. Chapter 6 draws conclusions, and describes possible future research directions.

CHAPTER II

TRANSPORT-AWARE IP ROUTERS: A BUILT-IN PROTECTION MECHANISM

2.1 Introduction

Like the computing resources of an end-server, there is only a limited amount of network resources — such as bandwidth, buffer, and processing power of routers — available to Internet users. The vulnerability of Internet to DDoS attacks roots in its best-effort model that provides no resource isolation among different IP flows, making it easy for attacking traffic to hog network resources. Having sufficient service differentiation and resource isolation at IP routers is essential not only to provide network Quality of Service (QoS) to end-users, but also to counter DDoS attacks as a powerful built-in protection mechanism inside the Internet.

To support network QoS, the Differentiated Service (DiffServ) infrastructure [38] has been proposed as a promising solution due mainly to its scalability and robustness. Based on the DS field in the IP header, IP flows are classified into different Behavior Aggregates (BAs). Services are provided for aggregates, not for individual flows, and defined by a small set of Per-Hop Behaviors (PHBs), which are the forwarding behaviors applied to different aggregates at IP routers. According to the three different services provided by DiffServ, three types of PHBs are specified: *Expedited Forwarding* (EF), *Assured For-*

warding (AF), and *Best-Effort* (BE). Although EF traffic is strictly policed and conditioned at edge routers, which protects the rest of network resources from the flooding EF traffic, the violated AF traffic is only re-marked without strict policing at network edges. More importantly, no conditioning is applied to BE traffic that is the main component of the Internet traffic. Compared to the best-effort service model, DiffServ is more resilient against DDoS attacks, but it is is still susceptible to DDoS attacks, especially the BE flooding traffic.

In the current DiffServ architecture, the QoS classification at core routers depends solely upon the DS field in the IP header,[1] yielding only coarse-grained service differentiation and resource isolation. No further service differentiation and resource isolation are provided among different transport-layer protocols within a BA. On the other hand, UDP and TCP are two dominant transport-layer protocols in the current Internet, but their services and traffic behaviors are quite different. Furthermore, UDP and ICMP flooding attacks have been widely used for stealing network bandwidth and disabling a victim server. It is necessary to provide resource isolation among TCP, UDP and ICMP traffic, and the resource consumption of UDP and ICMP traffic should be bounded. Besides meeting the requirement of the bi-directional service differentiation to TCP sessions, which the current DiffServ fails to achieve [82, 117], there are three reasons for differentiating TCP control segments — SYNs, FINs, ACKs and RSTs — from data segments, especially in the best-effort service model.

R1. Usually TCP control segments have much smaller packet size than data segments, so they consume much less network bandwidth than data segments.

R2. The loss of a TCP control segment, especially SYNs, incurs more serious performance degradation than the loss of a data segment.

[1] Multi-field packet classification is limited to edge routers in the DiffServ architecture.

R3. DDoS attack tools usually utilize TCP control segments for generation of DoS attacks, such as TCP SYN and ACK flooding attacks.

In other words, the coarse-grained service differentiation and the lack of resource isolation on meta-data packets not only degrade the assured service of TCP sessions, but also expose the vulnerability of Internet to DDoS attacks [44].

In this chapter, we propose a transport-aware IP (*t*IP) router architecture to provide fine-grained service differentiation and resource isolation among thinner aggregates without compromising scalability. The basic concept employed is layer-4 service differentiation and resource isolation, in which the transport-layer information is inferred from the IP headers and used for packet classification and resource management at IP routers. To support layer-4 service differentiation and resource isolation, we present a fine-grained QoS classifier and an adaptive weight-based resource manager, with which the *t*IP router infrastructure is built. The fine-grained QoS classifier divides each BA into thinner aggregates, and the adaptive weight-based resource manager provides service differentiation and resource isolation among these thinner aggregates. At core routers, we employ a two-stage packet classification mechanism to decouple the routing lookup from QoS lookup. The first stage performs the routing lookup at the input port, and the second stage performs the fine-grained QoS lookup at the output port after the packet is routed through the switching fabric.

The performance of the *t*IP router architecture is evaluated by simulation. The simulation results show that (1) the resource isolation provided by the *t*IP router significantly throttles the flooding traffic received by the victim server; (2) most of the flooding traffic is dropped close to the attacking sources, thus confining flooding damage and saving network resources; (3) the flooding traffic has little impact on the normal traffic that belongs to a different transport protocol, e.g., the UDP flooding or ICMP flooding traffic cannot interfere with the transmission of normal TCP traffic. Therefore, it can be utilized as a built-in protection mechanism of IP routers to counter DDoS attacks. Moreover, the *t*IP router provides better end-to-end TCP performance to applications: a high-tiered TCP session is guaranteed to have lower ACK loss rate and higher effective throughput than a low-tiered one.

Note that *t*IP routers do not require the support of DiffServ infrastructure, and they also work under the best-effort model. Actually, the best-effort model can be viewed as a simplified case of the DiffServ model, in which only single BA (best-effort) exists. To make our work more general and applicable in future, we investigate *t*IP routers with a more sophisticated DiffServ model. The subset of our results, which is based on the study of best-effort aggregates, can be directly applied to the current Internet.

The remainder of the chapter is organized as follows. Section 2.2 presents our initial motivation. Section 2.3 describes a fine-grained QoS classifier and an adaptive weight-based resource manager, the key components of the *t*IP router architecture. Section 2.4 evaluates the performance of the *t*IP router architecture. Finally, the chapter concludes with Section 2.5.

2.2 Initial Motivation

Our work was initially motivated by the desire of providing preferential treatments to TCP ACKs and achieving bi-directional differentiated service for each TCP session. By default, the current DiffServ architecture treats TCP ACKs from different user classes as BE traffic, sharing the same FIFO queue with BE data packets. The congestion caused by BE data packets results in buffer overflow at routers, and hence, bursty ACK losses. Especially under the condition that the TCP sender's congestion window size is small, the TCP sender is more vulnerable to ACK losses in the backward path, and its data transmission is interrupted by retransmission timeouts [113]. Therefore, providing service differentiation for ACK flows is essential to TCP-based applications.

One simple way to achieve this is that end-hosts mark each ACK as a premium, assured or best-effort packet, corresponding to the class of the data packet being acknowledged, but no enhanced mechanisms implemented at IP routers to distinguish ACKs from data segments. If the network resources is over-provisioned, the validity of this simple marking scheme has been confirmed by the authors of [82]. However, without proper resource provisioning and traffic conditioning for ACK aggregates, the ACKs and data segments that share the same queue could interfere with each other. Our simulation results in Section 2.4.3 show that the simple ACK marking scheme provides insufficient service differentiation and isolation to ACK flows when network resources are under-provisioned.

Furthermore, there are two serious drawbacks with this simple marking scheme: (1) the best-effort TCP traffic, which will continue to be the dominant load in the Internet, will not receive any performance improvement with the simple marking scheme; and (2) DDoS attacks in the Internet make the simple marking scheme much less attractive, since it is more vulnerable to various TCP flooding attacks. The simulation results in Section 2.4.2 confirm this claim. Therefore, to achieve better network QoS and counter DDoS attacks, we need to differentiate the TCP control segments from data segments and provide resource isolation between them at IP routers.

In the previous work [10], in order to improve the TCP performance in the context of network asymmetry, the *acks-first* scheduling scheme has been proposed, giving TCP ACKs priority over TCP data segments. So, the router always forwards ACKs before data segments. However, this *acks-first* scheme could cause starvation of data packets and violation of traffic profiles, especially under ACK flooding attacks. Also, no ACK identification scheme at routers was provided there.

This initial motivation differentiates our work from the previous work for countering DDoS attacks, because it is not a pure security mechanism. It can significantly improve the end-to-end TCP performance for clients, justifying the need for the wide deployment of *t*IP router architecture.

2.3 Transport-aware IP Router Architecture

To provide layer-4 service differentiation and resource isolation, we propose a fine-grained QoS classifier and an adaptive weight-based resource manager, both of which are essential to the *t*IP router architecture. The granularity of the classifier is still based on aggregates, not individual flows, and the resource manager is stateless, thus preserving the scalability and robustness of the original DiffServ or current Internet infrastructure. Moreover, at core routers the QoS lookup is decoupled from the routing lookup by employing a two-stage packet classification mechanism. The set of QoS filtering rules is small, and has the same/similar size at both edge and core routers. In contrast to routing lookup, QoS lookup is independent of network size and does not cause any scalability problem in packet classification. The *t*IP router architecture is detailed in the remainder of this section.

2.3.1 Two-stage packet classification

For service differentiation, layer-4 switching [69, 103] has been proposed, in which routing decisions are made based on the destination address as well as on the header fields at the transport or higher layer. Routing and QoS lookups are integrated into a single framework to fulfill layer-4 switching, and therefore, the forwarding database of a router consists of a large number of filters to be applied on multiple header fields. The deployment of a large-scale packet filtering mechanism [49, 61, 103] makes it feasible to implement layer-4 or higher switching at edge routers or at the front-end of server farms. However, layer-4 switching has primarily been used for load balancing by connection routers in server farms. It is very difficult to implement layer-4 switching at core routers due mainly to security and scalability difficulties. Even with fast and scalable packet classification, the problems with layer-4 switching at core routers are: (1) addition of higher-layer information — such as port numbers — and more routing entries enlarges the routing table at core routers, causing the routing lookup to require more memory and time; (2) when IP payload is encrypted, higher-layer headers become inaccessible.

To support layer-4 service differentiation and resource isolation, the fine-grained QoS classification has to work well at both edge and core routers. Therefore, we decouple the fine-grained QoS lookup from the routing lookup at core routers by employing a two-stage packet classification mechanism. In addition to overcoming the scalability problem at core routers, there are several other reasons for this decoupling as follows.

- Routing decisions must be made at the input port, but most of service differentiation — buffer management and packet scheduling — is performed at the output port.

- There is a large difference between the search spaces of routing lookup and QoS lookup. The size of routing table is very large and ever-increasing with the growth of Internet, but the filtering rule set of QoS classification is small and remains stable.

- Conventional routing lookup is based solely on destination addresses, which is a one-dimensional search, but QoS lookup is based on multiple fields, which is a multi-dimensional search.

Note that at core routers, our QoS lookup is restricted to the IP header fields, and the transport-layer information is accessed only at edge routers if necessary. The proposed QoS lookup should not become the performance bottleneck inside the router. Moreover, the decoupling greatly simplifies the implementation of packet classification. The architecture of the two-stage packet classification is illustrated in Figure 2.1, where the forwarding

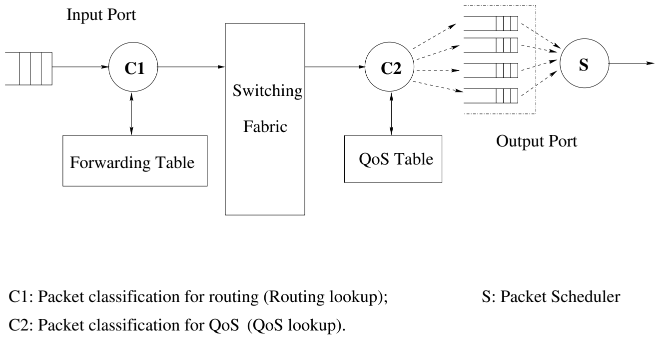


Figure 2.1: The architecture of the two-stage packet classifier at core routers

table is the local version of routing table in the line card. With the forwarding table, the routing/switching decision can be made locally at each input port.

2.3.2 Fine-grained QoS classifier

As the key component of the *t*IP router architecture, the proposed QoS classifier at routers uses several fields in the IP header for QoS classification in addition to the DS field. Transport-layer information is extracted to further divide a BA into a UDP aggregate, a TCP aggregate and a ICMP aggregate, and then to distinguish TCP control segments that mainly consist of ACKs, from TCP data segments in the TCP aggregate. QoS classification can be modeled at three different hierarchical levels as shown in Figure 2.2.

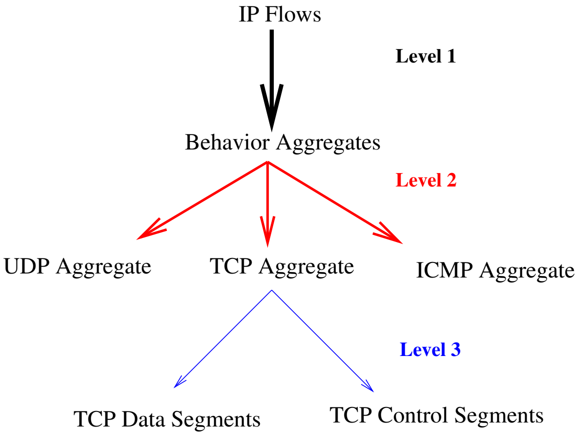


Figure 2.2: The three-level QoS classification

At the first two levels, it is straightforward to set the filtering rules. By checking DS and *protocol type* fields in the IP header against the filtering rules, the QoS classification is simple and does not cause any ambiguity. However, the accurate TCP control segment identification at level 3 could be much more complex. To implement the identification of TCP control segments at IP routers, the easiest way is to utilize one of the unused bits of DS field in the IP header. However, the problem with this solution is that it requires the modification of the IP header and cooperation from end-hosts. Moreover, the IETF has proposed use of the two unused bits of DS field for the deployment of the Explicit Congestion Notification (ECN) mechanism [91] at routers.

So, we propose size- and port- based identification of TCP control segments without requiring any new bit. Note that we present below a detailed description of a general TCP control segment identification, which can be applied to the identification of each individual TCP control segment like SYN, FIN, ACK and RST. The only difference at the port-based identification is to check different bits in the 6-bit flag field.

TCP Control Segment Identification

Initially, each TCP segment is encapsulated in a single IP packet, but IP fragmentation could occur at intermediate routers. Once the IP packet is fragmented, only the first fragment contains the TCP header. So, the IP packet that contains the TCP header must have a zero fragmentation offset. By checking the *fragmentation offset* field in the IP header, the ambiguity caused by IP fragmentation is eliminated. Theoretically, if an IP packet encapsulates a TCP control segment, it should meet the following requirements: (1) its protocol type is TCP; (2) its fragmentation offset is zero; and (3) the corresponding flags in the TCP header are ON. The first two conditions can be verified by checking the IP header, but the validation of the third condition needs to access the TCP header. Then, based on whether the TCP header is accessed or not, we have two versions of TCP control segment identification: *lightweight* and *heavyweight*.

Recent Internet traffic measurements [3, 109] have shown that about 40% of all IP packets are 40 bytes long, most of which are TCP control segments, implying that an overwhelming majority of TCP control segments are 40 bytes long. Since IP options are included primarily for network testing or debugging, and the fraction of packets with IP header options are typically less than 0.003% [68], it is reasonable to assume that no IP option fields are attached to TCP control segments.

The lightweight version is a size-based classifier. It takes advantage of the above observations. By checking the *total length* field in the IP header, it can tell TCP control segments from data segments without accessing the TCP header. The base filtering rule of TCP control segment identification is that the TCP segments whose total length are 40 bytes, are classified as TCP control segments. The rationale behind this is that, since the IP header without options is 20 bytes and the TCP header without options is 20 bytes, a total of 40 bytes is the minimum size of an IP packet that encapsulated a TCP segment (without any payload). Considering TCP options — MSS option (4 bytes), Window scale factor option (3 bytes), Timestamp option (10 bytes) and Selective Acknowledgment option (10/18/26 bytes) — that can appear in the TCP control segments like SYNs or ACKs and the requirement of a 4-byte boundary by padding NOOP options, the complete filtering rule of TCP control segment identification is set as:

$$Rule : \{\ X \mid X = 4 \cdot n;\ 10 \leq n \leq 20\}$$

where X is the total length of an IP packet and n is an integer. Since the maximum space that TCP options can use is 40 bytes, the maximum packet length of a TCP control segment is 80 bytes.

The main advantage of the lightweight version is that there is no need to access the TCP header, thus reducing overhead significantly. Its chief disadvantage is inaccuracy. The tiny TCP data segments that meet the filtering rules will be mis-classified as TCP control segments. However, because the tiny TCP data segments are most likely to belong to interactive TCP sessions, they have similar features of TCP control segments, i.e., small size and loss-sensitive TCP performance. For end-to-end TCP performance it is beneficial to separate them from, and give priority over, other TCP data segments. Moreover, the proposed adaptive weight-based resource manager has the ability to cope with this inaccuracy.

Since the lightweight version does not access the TCP header, it cannot further differentiate TCP control segments into SYNs, FINs, ACKs and RSTs. To achieve accurate and fine-grained TCP control segment identification, the TCP header needs to be accessed. The heavyweight version of TCP control segment identification is a port-based classifier. The matching scope is outside the IP header, and hence, the TCP flags of the TCP header are checked. Besides the additional overhead in accessing the TCP header, IPSec makes the port-based classification difficult. However, a multi-layer IPSec protocol [122] has been proposed, which allows trusted routers to access the transport-layer information.

Even in the heavyweight version of TCP control segment identification, especially for ACK identification, we still need to rely on the *total length* field in the IP header to eliminate the ambiguity caused by the following two facts:

- Some TCP implementations always set the ACK-flag bit ON, once the TCP connection is established [104]. Furthermore, some malicious TCP senders could intentionally set TCP flag bit ON in its TCP data segments.

- The piggyback mechanism in which ACKs are sent along with a reverse-direction data flow, results in a packet that could be interpreted as a TCP data segment or TCP ACK.

The ambiguity caused by always-on ACK-flag and piggybacking can be solved by simply checking the total IP packet length. The large packet with ACK-flag ON is classified as a TCP data segment, but the small packet with ACK-flag ON is classified as a TCP ACK. Considering the addition of TCP option fields like the Timestamp in the TCP header, we set the threshold to 80 bytes, as the available bytes for TCP options is 40. If the total packet length is larger than 80 bytes, the packet is classified as a TCP data segment. Otherwise, it is classified as a TCP ACK.

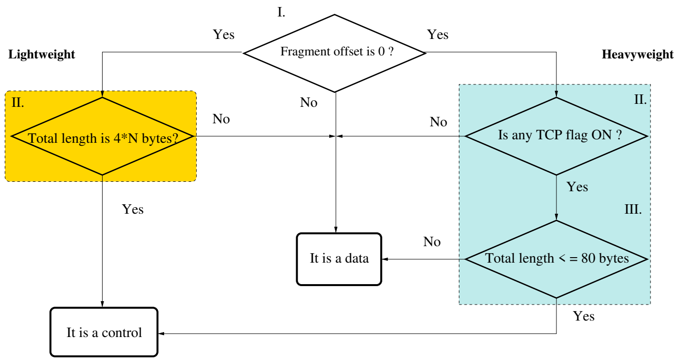


Figure 2.3: The flowchart of the TCP control segment identification algorithm

In summary, we use a lightweight version of flow identification at core routers. All transport-layer information is derived from the IP header only, and there is no need to access the TCP header. So, it does not violate the layering concept. At edge routers or firewalls, layer-4 or 7 switching and packet filtering have been implemented and deployed by many vendors. We provide a heavyweight identification scheme which accesses the TCP header, only as an option at edge routers. Therefore, even if the format of TCP header changes in future, only the heavyweight identification scheme, if employed at edge routers, needs to be modified. We believe, however, that the format of TCP header is unlikely to change due mainly to the compatibility problems with the already widely-deployed TCP.

A detailed description of the complete TCP control segment identification algorithm is given in Figure 2.3, where N is an integer such that $10 \leq N \leq 20$. There are three major steps in the heavyweight version, but only two steps in the lightweight version are given in Figure 2.3.

Validating lightweight identification

To validate the lightweight version of TCP control segment identification algorithm, six Internet traces taken at three different sites [5] are used. All the traces were collected between February 2002 and March 2002. The three chosen sites are located at high-bandwidth interconnection points. ADV represents the site where the OC3c (155Mbps) PoS (Packet over Sonet) link connects the Advanced Network and Services premises in Armonk, NY, to their ISP and the Internet2/Abilene network. ANL is referred to the OC3c link between the Argonne National Laboratory and the Ameritech Network Access Point (NAP) in Chicago. BUF is the site where the OC3 PoS link connects NYSERnet's router and State University of New York at Buffalo's router.

Traces	Mis-classified	Total Number	Error ratio
ADV-1	665	97053	0.68%
ADV-2	354	45463	0.78%
ANL-1	4351	560223	0.77%
ANL-2	2665	437085	0.61%
BUF-1	2218	208301	1.06%
BUF-2	2390	605127	0.39%

Table 2.1: Effect of mis-classification

From these traces, we found that no TCP control segment is mis-classified as a data segment and only an insignificant number of tiny TCP data segments are mis-classified as control segments. The filtering rule of 4-byte boundary significantly reduces the inclusion of tiny data segments (less than 80 bytes) in control segments. For example, in BUF-1 trace, without this rule, there would be 13318 tiny data segments that are mis-classified. However, after applying this rule, the number of mis-classified data segments is reduced to 2218. Table 2.1 gives the percentage of mis-classification of TCP data segments vs. the total data segments in each trace. Moreover, over 97% of the mis-classified tiny TCP data segments are with “PSH” flag ON in its TCP header, indicating the quick delivery requirement of these segments.

Discussion

Currently, only a negligible part of the IP traffic is reported to belong to IPv6 [3, 109], so the proposed QoS classifier is based only on IPv4. Compared to IPv4, the most important changes in IPv6 lie in the packet format. The header format of IPv6 is very different from that of IPv4. However, the following two features of IPv6 will make the TCP control segment identification even easier and faster: (1) IPv4’s variable-length options field is replaced by a series of fixed-format headers, and each IP packet has a base header followed by zero or more extension headers; (2) no fragmentation occurs at intermediate routers, and all the fragmentation and reassembly are restricted to end-hosts. So, it is easy to adjust the proposed QoS classifier to work properly in the context of IPv6.

The emergence of Internet Telephony — also called voice over Internet Protocol (VoIP) — does not pose any difficulty on the identification of lightweight version TCP control segments, since VoIP data streams are carried by Real-Time Transport Protocol (RTP) that is running on top of UDP, instead of TCP [94]. Moreover, the real audio streams show a significant regularity on packet lengths — concentrating on 244/254, 290/300 and 490/502 bytes [70], which are much larger than 80 bytes.

2.3.3 Adaptive weight-based resource manager

To enable better service differentiation and resource isolation between thinner aggregates, we propose an adaptive weight-based resource manager for IP routers. A hierarchical link-sharing structure is built, which is similar to the hierarchy of QoS classification and the class-based queueing management [41]. As shown in Figure 2.4, the root of the resource tree is the total link capacity. As the level of the resource tree gets lower, the IP flows that share the link are split into thinner aggregates. Each leaf node has its own queue, and every classified incoming packet is then inserted into the appropriate queue. Subsequently, the weighted round-robin scheduler will take care of these queued packets and select the next packet for transmission.

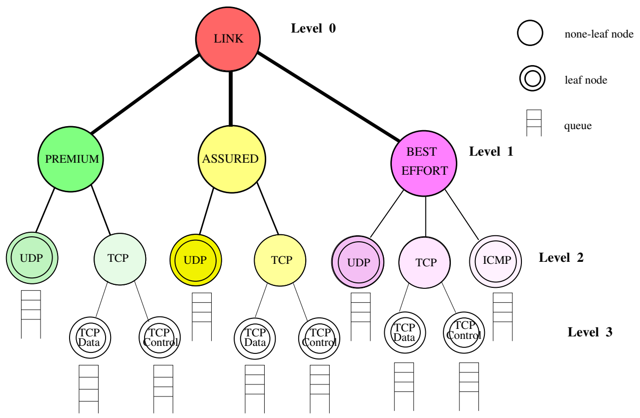


Figure 2.4: The link-sharing framework

At level 1 of the resource tree, each node represents the allocated bandwidth to each BA. The bandwidth allocation and priority assignment at the BA level are done by the Bandwidth Broker (BB) [80] via Service Level Agreements (SLAs). At level 2 of the resource tree, the BA is divided further into a UDP aggregate and a TCP aggregate. Each node at level 2 corresponds to the bandwidth allocated to the thinner UDP/TCP aggregate. Within each BA, the UDP aggregate and the TCP aggregate have the same priority and a weighted round-robin scheduling scheme is used between them. The composition of each BA is different. Most of an EF (Expedited Forwarding) aggregate is UDP real-time audio/video. However, in the AF (Assured Forwarding) and BE (Best-Effort) aggregates, the majority of packets belong to TCP. The weights assigned to the thinner aggregates are based on the recent empirical studies reported in [3, 68, 70, 109], which are listed in Table 2.2. Here we assume that the total weight of each BA is 1. The total weight assigned to UDP and TCP aggregates in BE is 0.99, since ICMP packets account for less than 1% of all IP packets and, by default, these ICMP packets are treated as BE traffic. Note that the weights at level 2 are tunable parameters that can be adjusted by network administrators to meet their local requirements. The weight assignment is strictly enforced only when there is no empty queue. Once a queue is empty, its assigned weight can be temporarily shared by other non-empty queues until the next packet enters the queue.

Type	Expedited	Assured	Best-Effort
UDP	0.7	0.05	0.04
TCP	0.3	0.95	0.95
ICMP	0	0	0.01
Total	1	1	1

Table 2.2: Level-2 weight distribution in different BAs

Each node at level 3 of the resource tree represents the bandwidth allocated to TCP data or control segments. The TCP data and control segments within the same TCP aggregate are also scheduled according to the weighted round-robin policy. The guiding principle for the weight setting at level 3 is that preference is given to control segments but there is a strict limit on the weight of control segments. The weight preference to control segments is the embodiment of resource overprovisioning as suggested in [82], and the strict limit prevents the misuse of preference. Because the overwhelming majority of the TCP control segments are TCP ACKs, we first present the rule of setting the weight of TCP ACKs, and then use the ACK weight as the baseline to derive the weights of all TCP control segments.

Setting the weight of ACKs

The rule of thumb for setting the weight for TCP ACKs is to approximate the upper bound of their bandwidth consumption. If the weight is measured in number of packets, the ratio of the weight of TCP ACKs to that TCP data segments is 1:1. This *one data segment vs. one ACK* policy is based on the fact that the transmission of a TCP data segment will later trigger a TCP ACK. Considering wide deployment of the delayed ACK mechanism at TCP receivers, weights are assigned to give preference to TCP ACKs, and this preference is also intended to provide a safety margin to other TCP control segments and tiny TCP data segments, in which the lightweight version of TCP control segment identification is used.

Adhering to the policy of *one data segment vs. one ACK*, if the weight is measured in number of bytes, then we should consider the average packet size. As mentioned earlier, a great majority of TCP ACKs are 40 bytes long, so the average size of TCP ACK is 40 bytes. Common MSSs of TCP implementations are 512, 536 and 1460 bytes [3, 109]. Including the 40 bytes of both the IP and TCP headers, the total packet lengths for these MSSs are 552, 576, and 1500 bytes, respectively. Their observed ratio is 1:1:2. Thus, the average size of a TCP data segment is 1K bytes and the ratio of the weight of ACKs to that of data segments is 1:25.

The traffic load distribution inside the DS domain will be balanced by routing algorithms and traffic engineering. At an interface of a core router, the volume of outgoing TCP data traffic equals that of incoming TCP data traffic, and each outgoing TCP data segment[2] implies an incoming TCP ACK to the interface. Therefore, this *one data segment vs. one ACK* policy is valid for core routers inside a DS domain.

[2]Or two outgoing TCP data segments if the delayed ACK mechanism is ON at the TCP receiver.

However, at a leaf router or a boundary router between two different DS domains, this policy is often invalid due to the asymmetry of traffic load. This traffic load asymmetry has been observed in traffic measurements of the trans-Atlantic link [109]. The weight of TCP ACKs should depend on the weight allocated to the TCP data segments in the reverse direction. The weight of the ACK aggregate at leaf or boundary routers can be set based on traffic measurements or with the help of the Bandwidth Broker. Like the weight setting at level 2, the weight assigned to ACKs, which is the baseline of TCP control segments, is also a tunable parameter and can be adjusted locally.

Once the weight of the ACK aggregate is set, we use a simple adaptive calibration scheme to derive the weight of all TCP control segments for which the ACK weight is used as the baseline. The mechanism of the weight calibration works similarly to the adaptive-weighted packet scheduling of EF traffic [114]. Its goal is to increase the flexibility of the resource manager to absorb bursty control traffic and the tiny data segments that are mis-classified as TCP control segments.

Adaptive weight calibration of TCP control aggregate

As in [112], we use the estimated average queue size of the TCP control aggregate to adaptively adjust the weight. The average queue size of the TCP control aggregate is calculated by using a low-pass filter with an exponentially-weighted moving average. Let *avg* be the average queue size, q the instantaneous queue size and f_l the parameter of the low-pass filter, then the average queue size of TCP control aggregate is estimated as:

$$avg \leftarrow (1 - f_l) \cdot avg + f_l \cdot q.$$

To reduce the instantaneous fluctuation of queue size, the parameter of the low-pass filter f_l is set to 0.01.

Assuming that the weight of TCP ACKs is w_a, we set the original weight of TCP control aggregate w_c to $1.2\ w_a$. To adaptively calibrate the weight of TCP control aggregate, two thresholds, min_{th} and max_{th}, are introduced. By keeping the average queue size of the TCP control aggregate below the maximum threshold, bursty losses of TCP control segments are prevented. To accomplish this, the weight of control aggregate should be proportionally increased once the average queue size of control aggregate exceeds the minimum threshold. The values of min_{th} and max_{th} are set to one fourth and three fourths of the buffer, respectively. The linear relationship between the weight and the average queue size of control aggregate is given by:

$$f(C) = \begin{cases} w_c, & C \in [0, min_{th}) \\ \frac{(U-w_c)\cdot(C-min_{th})}{max_{th}-min_{th}} + w_c, & C \in [min_{th}, max_{th}) \\ U, & C \in [max_{th}, full] \end{cases}$$

where $f(C)$ is the weight function of control aggregate, U is the upper limit that the weight of control aggregate can reach, and C is the average queue size of control aggregate. Since the total weight for TCP aggregates is fixed, the increase of control aggregate's weight must cause the same amount of decrease in the data aggregate's weight. However, once the average queue size of control aggregate reduces below max_{th}, the weights taken from data aggregate will be returned.

The weight calibration favors the control aggregate but disfavors the data aggregate, which is consistent with the guiding principle of the weight settings. The rationale behind this is that the bandwidth taken by the data aggregate is usually much more than the bandwidth consumed by the control aggregate; the small amount of bandwidth shift from the data aggregate to the control aggregate can prevent bursty losses of the control segments, but only leads to a single isolated data packet loss or just a longer queueing delay. However, the weight of control aggregate cannot exceed the upper limit, which prevents the abuse of preferential treatment of TCP control segments and protects the TCP data aggregate from starvation. U is set to $2w_c$ in our simulation.

2.4 Performance Evaluation

The proposed *t*IP router architecture is evaluated by simulation with *ns-2* [77, 111]. According to the purpose of simulation, we categorize the simulation experiments into two different classes. One is used for evaluating the capability of resource isolation under flooding attacks, the other is used for evaluating the capability of service differentiation for end-to-end TCP performance. The network topologies of the two classes are different. In the presentation of simulation results, *Existing* refers to the current DiffServ (or Internet) architecture, *Marking* refers to the ACK marking scheme proposed in [82] and *Reserved* refers to the reserved bandwidth for EF and AF flows.

2.4.1 The simulation setup

In our simulation experiments, each end-host is connected to its respective edge router, and the edge routers are connected via core routers. The link capacity and one-way propagation delay between an end-host and an edge router are 10 Mbps and 1 ms, respectively. The one-way propagation delay between an edge router and a core router is 8 ms, but that between two core routers is 16 ms. The UDP/TCP data segment size is set to 1000 bytes, and the TCP control segment size is set to 40 bytes. The version of TCP used in the simulation is TCP New-Reno since it has been widely deployed in the Internet, and the delayed-ACK mechanism is ON.

2.4.2 Resource Isolation

In the flooding experiments, the link capacity between an edge router and a core router is 6 Mbps, but that between two core routers is 5 Mbps. As the network topology for the purpose of a DDoS attack, it is convenient to consider the network topology in terms of a tree graph. The victim's machine is at the root of the tree, with network routers being intermediate nodes in the tree. The leaf nodes of the tree are the flooding sources and normal end-hosts. The simulated network topology for DDoS attacks is shown in Figure 2.5. In our flooding experiments, there is a flooding source in each stub network except for the one that the victim belongs to. The flooding rate at each source is constant and set to 5000 packets per second in SYN and ACK flooding attacks, but 500 packets per second in UDP and ICMP flooding attacks. At the same time, there are TCP connections running from normal end-hosts to the victim as the background traffic.

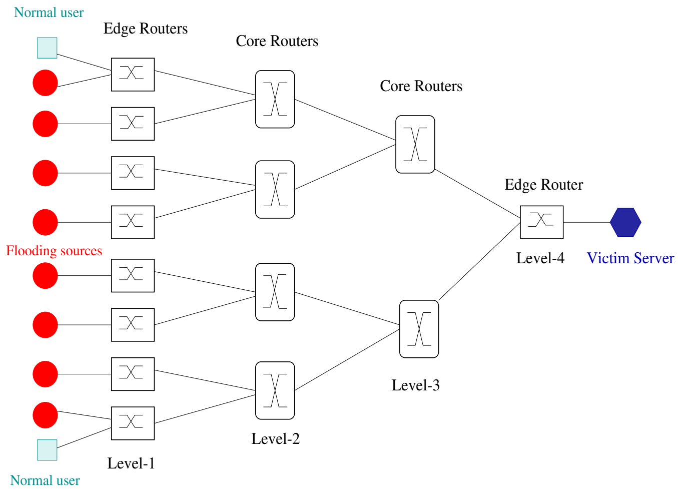


Figure 2.5: The simulated network topology used for resource isolation

We first measure the volume of the flooding traffic that reaches the victim under different IP architectures. Four types of flooding attacks — SYN, ACK, UDP, and ICMP flooding — are simulated. All the flooding traffic is transported by the BE service. Since there is no difference in treating the BE traffic in the current Internet and DiffServ architectures, we normalize the various flooding traffic reached the victim in these architectures to 1 to make the presentation easier. Then, the flooding traffic received at the victim in the *t*IP router architecture is properly scaled based on the normalization. Figure 2.6 shows that the *t*IP router throttles the flooding volume that reaches the victim and effectively protects the victim from flooding attacks. More importantly, most of the flooding traffic will be dropped by the first few routers before they reach the core of the network, thus confining the damage caused by the flooding source mainly to the local stub network where it originated. The cascaded throttling of flooding traffic at a first few routers shields the rest of the Internet, and saves the network bandwidth. The cascaded throttling of flooding traffic is depicted in Figure 2.7.

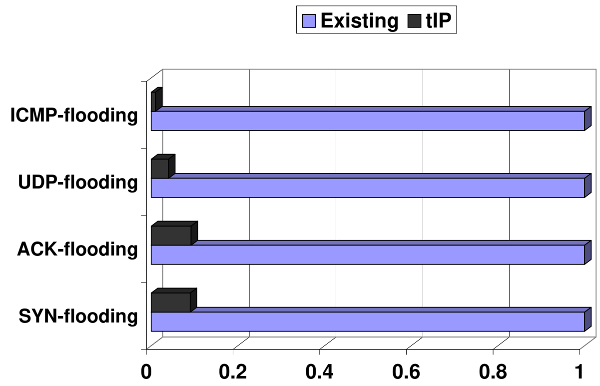


Figure 2.6: The flooding traffic volume reached the victim

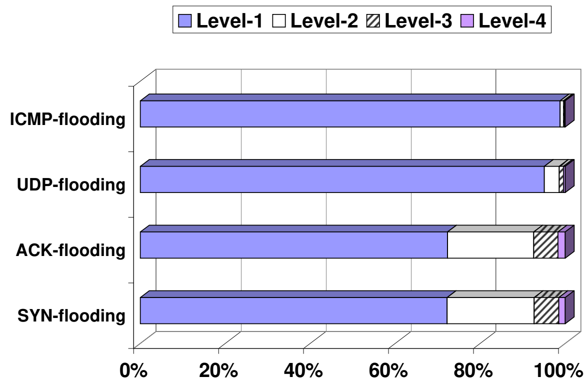


Figure 2.7: The distribution of dropped packets at different level routers

Moreover, during flooding attacks, the effective TCP throughputs in *Existing* and *Mark-*

ing are reduced almost to zero, but the one in *t*IP router can still achieve 95% (in the cases of UDP and ICMP flooding) and 85% (in the cases of SYN and ACK flooding) of the bandwidth assigned to the entire BE traffic, thanks to the layer-4 resource isolation. Figure 2.8 illustrates the dynamics of average throughput of a background TCP connection under the UDP or ICMP flooding attack that starts at 0.2s. Since the TCP connection is not interfered with by UDP or ICMP floods, its average throughput follows a typical sawtooth-like curve.

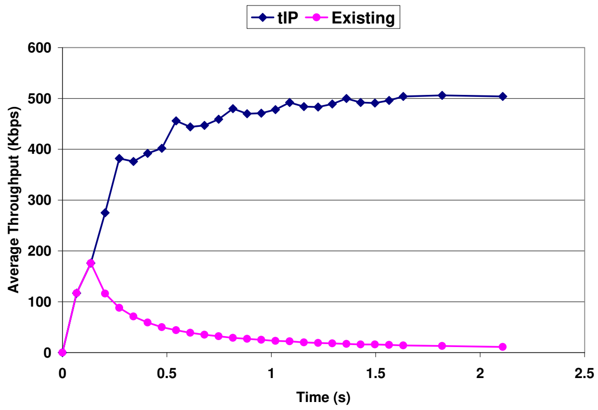


Figure 2.8: Average throughput of the background TCP connection

Our simulation results show that *t*IP router provides a built-in protection mechanism to counter DDoS attacks. Splitting layer-4 traffic greatly reduces the performance degradation caused by such DDoS attacks as flooding of UDP, ICMP, TCP SYN and ACK traffic. The resource isolation provided inside the BE traffic class is especially valuable, since the edge routers in the DiffServ architecture perform traffic conditioning and policing on EF and AF traffic, but not on BE traffic.

For EF or AF traffic, both UDP and ICMP flooding do not cause much damage in all DiffServ architectures because the edge routers perform traffic conditioning and policing

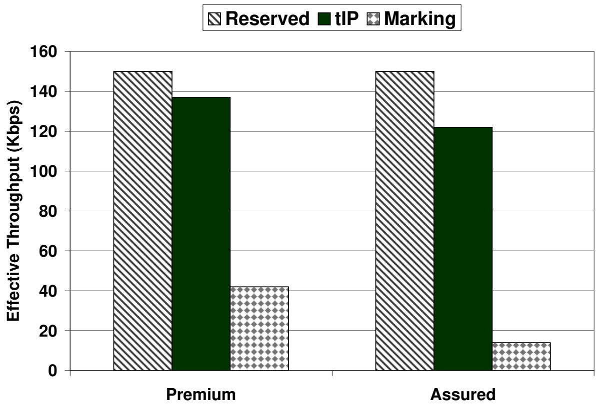


Figure 2.9: The goodput of conformant EF and AF flows under the ACK flooding attack

on EF and AF traffic. However, the simple *Marking* scheme exposes more vulnerability to the ACK flooding attacks. In this architecture, the ACK flows are accepted without strict policing based on the belief that the small bandwidth requirement by ACK flows can be absorbed by over-provisioning. The flooding ACK flows marked as EF or AF traffic can seriously violate the traffic profile between the stub network and the leaf router that connects the stub network to the Internet. In large-scale DDoS attacks, even if only a small number of ACKs are flooded from each attacking source, once these ACKs are aggregated at core routers where no traffic conditioning is performed, the flooding ACK aggregates can "steal" the reserved bandwidth from the conformant aggregates. Figure 2.9 charts the goodputs of conformant EF and AF flows from a normal end-host to the victim in *Marking* and *t*IP router, and clearly shows the vulnerability of *Marking* to the ACK flooding attack and the robustness of *t*IP router to the same attack. Note that the conformant EF flow is carried by UDP, but the AF one is carried by TCP.

2.4.3 Service differentiation

In the experiments of service differentiation, the link capacity between an edge router and a core router is 3 Mbps, but that between two core routers is 1 Mbps. We study the

TCP flows and measure their ACK loss rate and effective throughput, where we not only compare the *t*IP router with the existing DiffServ, but also with the marking scheme for TCP ACKs proposed in [82]. The simulation network topology for service differentiation is shown in Figure 2.10, which is a relatively simple, yet sufficiently representative topology for validating end-to-end TCP performance. The experimental configuration is as follows. Three targeted TCP connections are established from S_1 to C_1, which receive premium, assured and best-effort services, respectively. In addition to the three targeted TCP connections, two more TCP connections carry BE data from S_2 to C_2. All of them have infinite amounts of data to send, i.e., with commonly-used "persistent" sources.

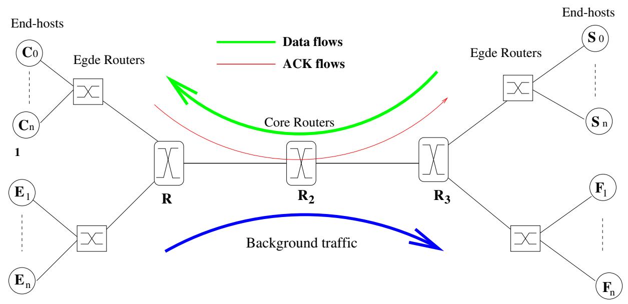


Figure 2.10: The simulated network topology for service differentiation

With respect to the direction of targeted TCP data flows, we name the path $R_3 \rightarrow R_1$ as the *forward* path and the path $R_1 \rightarrow R_3$ as the *backward* path. The resources along the forward path $R_3 \rightarrow R_1$ is properly provisioned for the premium and assured forwarding traffic, but the remaining network resources are periodically exhausted by the BE traffic, causing random data losses to occur in the forward path.

On the backward path $R_1 \rightarrow R_3$, similar background traffic is generated between E_i to F_i, and shares the same path with the targeted ACK flows. The background traffic is a mixture of premium, assured and BE traffic. Compared to the simulation configuration in the forward path, there are two key differences in the backward path:

- the network resources for the premium and assured services in the background traffic are under-provisioned; and
- the BE traffic consists of not only TCP flows but also UDP flows, which causes severe congestion in the backward path, thus resulting in bursty packet losses.

The Simulation Results

The ACK loss rates of targeted TCP connections are plotted in Figure 2.11, and the effective throughputs of the targeted TCP connections are plotted in Figure 2.12. The simulation results show us that:

- the *t*IP router provides better service isolation for ACK flows, significantly lowering the ACK loss rate and increasing effective throughput; and
- the ACK marking scheme cannot support service isolation for ACK flows when network resources are under-provisioned, thus resulting in bursty ACK losses and hence degrading TCP performance significantly.

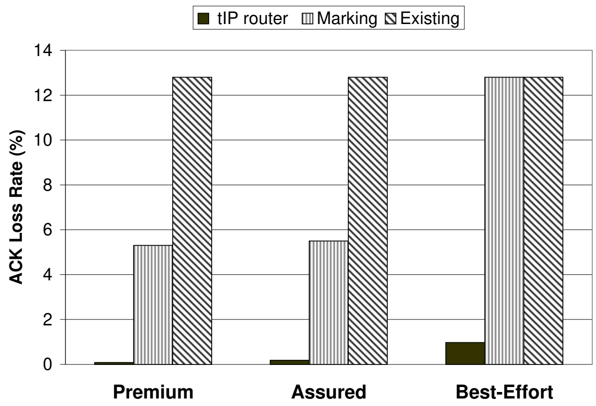


Figure 2.11: The ACK loss rate in different DiffServ architectures

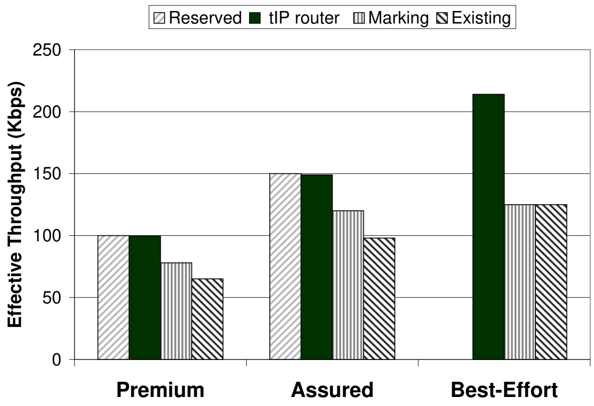


Figure 2.12: The effective throughput in different DiffServ architectures

For EF and AF traffic, the ACK loss rates of *Marking* are much lower than those of *Existing*, but are much higher than those of the *t*IP router. Moreover, most of ACK losses are bursty rather than random, lowering effective throughput. For EF traffic, the main reasons for bursty ACK losses are: (1) to support low delay for EF traffic, the buffer space for premium service is very small, and can only accommodate one or two data packets; (2) the size of a data segment is much larger than that of an ACK. Once the buffer has been filled with one or two data segments, the subsequent incoming ACKs will be dropped.

In the *Existing* and *Marking* DiffServ architectures, AF traffic shares the same FIFO queue with BE traffic, but AF packets are much less likely to be dropped than BE ones. However, without proper resource provisioning for ACK flows, the ACKs are more likely to be marked as high drop-precedence packets at edge routers due to the corresponding traffic profile violation. Under a severe congestion, all the packets marked with high drop-precedence will be dropped, causing bursty ACK losses. Since bursty ACK losses cause much severer degradation to TCP performance than random ACK losses, even modest ACK loss rates for EF and AF can greatly reduce their effective throughput.

As the *Existing* DiffServ, the ACK marking scheme provides no improvement to BE traffic. BE ACKs experience a high loss rate in the backward path because of the congestion caused by the UDP flows in the background traffic. Furthermore, due to data losses in the forward path, an ACK loss for retransmission in the backward path leads to a timeout, reducing $cwnd$ to 1, triggering a slow-start, and hence, degrading effective throughput significantly.

In contrast, the *t*IP router architecture significantly improves the performance of BE TCP traffic, thanks to its resource isolation between UDP and TCP flows, as well as between ACKs and TCP data segments within the BE class. The *t*IP router not only provides better service quality to high-tiered services, but also significantly improves the performance of BE TCP sessions.

Note that, although the TCP version in our simulation experiment is New-Reno, most of the simulation results in this chapter are applicable to all TCP variants for the following reasons. First, the TCP behaviors after a retransmission timeout for all of these schemes are similar. TCP variants differ only in the way of recovering from packet losses after a fast retransmit. Second, the ACK losses in the reverse path only lead to a timeout or slower congestion window growth, but cannot trigger a fast retransmit.

2.5 Conclusions

We presented a transport-aware IP router architecture to provide layer-4 service differentiation and resource isolation. The key components of the *t*IP router architecture are the fine-grained QoS classifier and the adaptive weight-based resource manager. A two-stage packet classification mechanism is devised to decouple the fine-grained QoS lookup from the routing lookup at core routers. BAs are further divided into thinner aggregates. By using separate queues and adaptive-weighted bandwidth allocation, better service differentiation and resource isolation are achieved for these thinner aggregates.

We evaluated the performance of the *t*IP router architecture by simulation. The simulation results show that:

- it provides a built-in protection mechanism to counter DDoS attacks: the flooding traffic is significantly throttled and most of the traffic is dropped in a close proximity to their sources;
- the resource isolation of the *t*IP router protects the normal traffic from the flooding traffic that belongs to a different transport protocol;
- the *t*IP router guarantees that high-tiered TCP sessions receive better service and hence yield better performance in terms of loss rate, end-to-end delay and effective throughput, than low-tiered TCP sessions;
- it not only achieves better service quality for high-tiered services, but also significantly improves the performance of BE TCP sessions.

Furthermore, the simulation results demonstrate that a simple ACK marking scheme does not provide good service differentiation and resource isolation for ACK flows when network resources are under-provisioned. It exposes the vulnerability of EF and AF traffic to the ACK flooding attacks. The *t*IP router architecture is therefore necessary to provide better network QoS to TCP sessions, and is a simple yet powerful built-in protection mechanism to counter DDoS attacks.

CHAPTER III

SYN-DOG: SNIFFING SYN FLOODING ATTACKS

3.1 Introduction

The latest research results have shown that more than 90% of the DoS attacks use TCP [75], and TCP SYN flooding dominates in the available attacking tools and the number of DoS attacks known to date [75]. TCP SYN flooding consists of a stream of spoofed TCP SYN packets directed to a listening TCP port of the victim. Not only the Web servers but also any system connected to the Internet providing TCP-based network services, such as FTP or mail servers, are susceptible to TCP SYN flooding attacks.

The stateless and destination-based nature of Internet routing infrastructure cannot differentiate a legitimate SYN packet from a spoofed one, and TCP does not perform strong authentication on SYN packets. Therefore, under a SYN flooding attack, the victim server cannot single out, and respond only to, legitimate connection requests while dismissing the spoofed. Moreover, the spoofed source address conceals the true origin of flooding sources, thus making it difficult to identify, and trace back, the flooding sources.

Most of the previous related work — such as Syn cache [62], Syn cookies [15], Syn-Defender [65], Syn proxying [4], and Synkill [96] — focused on mitigating the effect of SYN flooding attacks on the victim in order to sustain service availability. These defense mechanisms can be characterized by two common features.

- They are installed at the firewall of, or inside, the victim server, hence providing no hint about the location of the SYN flooding source(s). They have to rely on the IP traceback [14, 83, 95, 99, 100] to locate the flooding source(s), which is a time-consuming process .

- They require to maintain or compute states for each TCP connection, which not only makes themselves vulnerable to SYN flooding attacks but also degrades end-to-end TCP performance, e.g., longer delays in setting up TCP connections.

Recent experiments have shown that even a firewall designed specifically to resist SYN floods, became futile under a flood of 22,000 packets per second [81]. On the other hand, in the absence of SYN flooding attacks, such a defense mechanism just wastes resources. By checking only its backlog queue, the victim server cannot tell whether it is under a flooding attack or simply overloaded with legitimate SYN packets. Without an efficient detection mechanism, the defense mechanism installed at a server consumes additional server resources, making the already-overloaded server even less responsive to legitimate requests.

Furthermore, instead of consuming a server's resource, the latest reflected SYN/ACK attacks [45] seize the link bandwidth between the victim and its Internet Service Provider (ISP) by flooding SYN/ACK responses. The above-mentioned defense mechanisms cannot provide any protection for such a "bandwidth consumption" attack, and innocent reflectors make IP traceback more difficult to locate the true flooding sources.

We, therefore, need a simple stateless mechanism to sniff these SYN (including reflected SYN/ACK) flooding attacks. It should also be immune to any type of flooding attacks, and should not degrade end-to-end TCP performance either. By "sniff" we mean not only detect an attack, but also easily uncover the flooding sources without resorting to

IP traceback. The sniffing mechanism is orthogonal to the defense mechanisms mentioned earlier. The relationship between the two is analogous to the one between a surveillance radar and surface-to-air missiles within an anti-aircraft defense system, where the radar always checks for intruding enemy aircraft, but the missiles stay inactive until an enemy intrusion is detected.

In this chapter, we propose a simple and robust mechanism, called *SYN-dog*, to sniff SYN flooding attacks. While SYN-dog monitors the TCP traffic constantly, the defense system can be turned off until a warning is issued by the SYN-dog. Instead of monitoring the ongoing traffic at the front end or the victim server itself, SYN-dog is installed at a leaf router that connects a stub network[1] to the Internet. The simplicity (hence the attractiveness) of SYN-dog lies in its statelessness and low computation overhead. Its key features include:

- SYN-dog utilizes the distinct protocol behavior of TCP connection establishment and teardown, which consists of SYN–FIN pair and SYN–SYN/ACK pair, for SYN flooding detection.

- SYN-dog is insensitive to sites and access patterns: the non-parametric Cumulative Sum (CUSUM) method [19] is applied, making SYN-dog robust, much more generally applicable and its deployment easier.

- SYN-dog plays a dual role in sniffing SYN flooding attacks: the first-mile SYN-dog and the last-mile SYN-dog. Due to its close proximity to the flooding sources, the first-mile SYN-dog not only alarms on the ongoing SYN flooding attacks, but also reveals the location of the flooding sources.

[1] 1.0 A stub network only carries packets to and from local hosts.

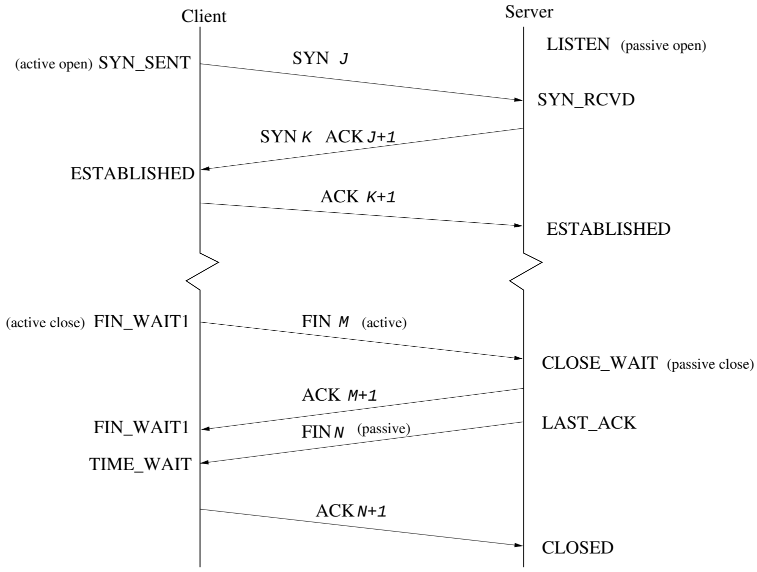


Figure 3.1: TCP states corresponding to normal connection establishment and teardown (from [104])

As shown in Figure 3.1 which is borrowed from [104], SYN and FIN packets delimit the beginning (SYN) and end (FIN) of each TCP connection in the same direction. In contrast, SYN and SYN/ACK packets signal the start of a TCP connection establishment in two opposing directions. Under the normal condition, one appearance of a SYN packet results in: (1) the eventual return of a FIN packet in the same direction; (2) the corresponding transmission of a SYN/ACK packet in the reverse direction within one round-trip time (RTT). Thus, the difference between the number of SYN and FIN (or SYN/ACK) packets can be utilized to sniff SYN flooding attacks, and the SYN-dog is an instance of the Sequential Change Point Detection [13].

Only a few additional variables are introduced to count the number of received TCP SYN, SYN/ACK, FIN and RST packets at each interface of a leaf router. We rely on packet classification to differentiate the various TCP control packets at leaf routers. Large-scale packet classification mechanisms [49, 61, 103] have been proposed and implemented, making it possible for routers to distinguish TCP control packets from others at a very high speed. Therefore, the SYN-dog's capability to withstand any flooding attacks depends on the ability of a leaf router in classifying and forwarding packets, typically at the rate of a million packets per second [61]. No per-connection state or state computation is involved in SYN-dog. Unlike the other network intrusion detection systems that maintain state for *each* TCP connection, SYN-dog does not have the cold-start problem[2] mentioned in [51].

SYN-dog is, in some sense, a by-product of the router infrastructure that can differentiate TCP control packets from data packets [114]. As SYN-dog tells TCP control packets from data packets, fine-grained service differentiation and resource isolation can be made on TCP flows. End-to-end TCP performance can be improved significantly as shown in [114]. Therefore, SYN-dog benefits not only victim servers but also the clients inside the stub network, making it attractive for wide deployment.

The efficacy of SYN-dog is evaluated by extensive trace-driven simulations. Traces taken from different sites at different times are used to test the sensitivity of SYN-dog. The evaluation results show that SYN-dog has short detection latency and high detection accuracy. Furthermore, SYN-dog is incrementally deployable and its implementation overhead is very low. The scheme of utilizing distinct protocol behaviors and the CUSUM algorithm can also be used for sniffing other types of flooding attacks.

The remainder of the chapter is organized as follows. Section 3.2 briefly describes the working mechanisms of SYN and reflected SYN/ACK flooding attacks. Section 3.3 presents the SYN-dog framework, including the structure and placement of SYN-dog. Section 3.4 describes two different detection methods with respect to the different SYN pairs utilized. Section 3.5 details the proposed CUSUM algorithm for detecting abnormal

[2] 1.0 A "cold start" refers to the situation when a network intrusion detection system begins to run, or after it is restarted, it doesn't know how to deal with the incoming TCP traffic that belongs to the connections established earlier.

traffic behaviors, and discusses its robustness against network anomalies. Section 3.6 evaluates the performance of SYN-dog using trace-driven simulations. Finally, conclusions are drawn in Section 3.7.

3.2 SYN and Reflected SYN/ACK Flooding

SYN flooding attacks exploit the TCP's three-way handshake mechanism and its limitation in maintaining half-open connections. When a server receives a SYN request, it returns a SYN/ACK packet to the client. Until the SYN/ACK packet is acknowledged by the client, the connection remains in half-open state for a period of up to the TCP connection timeout, which is typically set to 75 seconds. The server has built in its system memory a backlog queue to maintain all half-open connections. Since this backlog queue is of finite size, once the backlog queue limit is reached, all connection requests will be dropped. If a SYN request is spoofed, the victim server will never receive the final ACK packet to complete the three-way handshake. Flooding spoofed SYN requests can easily exhaust the victim server's backlog queue, causing all the incoming SYN requests to be dropped. Note that the spoofed source address should be an invalid IP address (or the corresponding end-host is disconnected from the Internet at that time) so that it may be unreachable from the victim; otherwise, any end-host that receives the SYN/ACKs from the victim would send a RST to the victim. A RST packet is issued when the receiving host does not know what to do with the packet it received. Arrival of a RST causes the connection to be reset, foiling the flooding attack.

While the conventional SYN flooding is an attack of "system resource consumption," the recent reflected SYN/ACK flooding attacks [45] virtually "disconnect" a victim server from the Internet by hogging the link bandwidth between the victim and its ISP with an excessive number of SYN/ACK packets (a.k.a. bandwidth consumption attack). It is a kind of Distributed Reflection DoS (DRDoS) attacks [86]. In reflected SYN/ACK flooding attacks, a large number of innocent BGP routers (service port 179) and well-known TCP servers are exploited as the reflectors. The attacker "sprays" the spoofed SYN packets, whose source IP addresses are falsified as the victim's IP address, across a large number of reflectors. Each reflector alone only receives a moderate flux of spoofed SYN packets so that it can easily sustain the availability of its normal service. However, these innocent reflectors involuntarily reflect and amplify the malicious SYN packets. Their SYN/ACK responses, which are aggregated and flooded to the victim, are excessive, using up the link bandwidth between the victim and its ISP.

Note that in reflected SYN/ACK flooding attacks, all malicious SYN packets from the attacker must traverse the leaf router that connects the attacker to the Internet, in order to reach the Internet and then get sprayed across the numerous reflectors. The SYN-dog installed at this leaf router can detect the flow of these malicious SYNs, since no SYN/ACKs return to the attacker and the total number of malicious SYNs is still very large. So, the same method for sniffing SYN flooding attacks can be applied to sniff reflected SYN/ACK flooding attacks. To SYN-dog, the reflected SYN/ACK flooding attack is just a variation of conventional SYN flooding attack.

3.3 The Framework of SYN-dog

In this section, we first briefly describe the structure of SYN-dog, and then discuss where to place SYN-dog.

3.3.1 Structure of SYN-dog

SYN-dog consists of two *sniffers*, one at the in-bound interface and the other at the out-bound interface of a leaf router. We refer to the traffic from the Internet to the stub network as *in-bound*, and the traffic in the other direction as *out-bound*. The *in-bound sniffer* (*out-*

bound sniffer) monitors the incoming (outgoing) TCP traffic. Figure 4.12 illustrates the structure of SYN-dog placed at a leaf router.

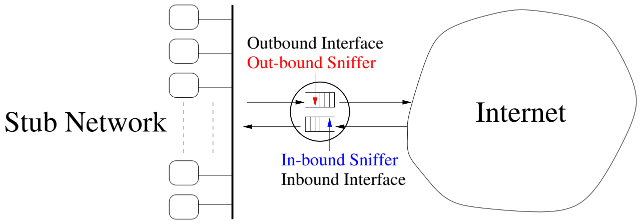


Figure 3.2: The structure of SYN-dog placed at a leaf router

Each leaf router can be either the first-mile or the last-mile router, depending on the direction of traffic between the stub network and the Internet. The SYN-dog at a leaf router, therefore, plays a dual role in detecting SYN flooding attacks:

- as the first-mile SYN-dog, it detects the flooding attacks that are originated from its local stub network and traces the flooding sources inside the local stub network; and
- as the last-mile SYN-dog, it detects flooding attacks on a server that resides inside the local stub network, and issues a warning signal upon detection of an attack.

The first-mile SYN-dog plays the primary role in sniffing a flooding attack, due mainly to its proximity to the flooding sources. Once an ongoing SYN flooding attack is detected, the first-mile SYN-dog's warning signal automatically indicates the flooding sources to be inside the stub network to which the SYN-dog is connected, without resorting to the IP traceback. However, the detection sensitivity of the first-mile SYN-dog may diminish as the number of attackers gets larger. In a large-scale DDoS attack, the flooding sources can be orchestrated so that each flooding traffic source may cause only an insignificant deviation from the normal traffic pattern.

In contrast, the last-mile SYN-dog can quickly detect the flooding attacks as all of the flooding traffic streams are merged at the last-mile router. Although it cannot provide any hint about the flooding sources, upon receipt of the last-mile SYN-dog's warning signal for an attack, the defense system like SynDefender can be triggered to protect the victim. To bring down the victim under protection, the flooding sources have to increase their flooding rates significantly, but this will make it easier for the first-mile SYN-dog to detect the flooding attack and locate its source(s). So, the last-mile SYN-dog plays an important complementary role in sniffing SYN flooding attacks.

3.3.2 Placement of SYN-dog

In addition to its installation at leaf routers, SYN-dog can also be placed at the firewall or the proxy server of a large organization which has only a single connection to the external world. All packets of a TCP session must pass through the same SYN-dog. However, we do not recommend the SYN-dog to be installed at core routers mainly because (1) it is close to neither flooding sources nor the victim; and (2) packets of the same flow could traverse different paths; (3) it is not always possible to accurately classify different TCP control packets at core routers due to the possible use of IPSec; (4) it cannot detect the reflected SYN/ACK flooding attacks easily, since malicious SYNs are diffused before reaching the core router.

As has been done in most intrusion detection (ID) systems, SYN-dog can be placed on the link that connects a stub network to the Internet by monitoring the bidirectional traffic on that link. However, besides the extra specialized equipment and manpower required, during high peak (near saturation) flow rates, almost no event of any kind would be logged by an ID system — they either have to drop packets at a very high rate or require a high-performance multi-CPU architecture for packet state analysis. As the link speed continues to improve, it will be more difficult for network flow monitors (that run on a typical PC) to pace with the network's packet transfer rate.

Recently, multi-homed ASs become necessary to improve availability, reliability and load-balancing. In such a case, the stub network is connected to the Internet by multiple leaf routers. However, as long as the packets that belong to the same TCP session go through the same leaf router, SYN-dog still works. If the packets of the same TCP session go through different leaf routers, we need a loose synchronization mechanism between the SYN-dogs in these leaf routers, which is the subject of our future work.

3.4 Two Detection Methods

Based on the distinct protocol behavior of TCP connection establishment and tear-down, we utilize two types of packet pairs — SYN–FIN and SYN–SYN/ACK pairs — to detect SYN flooding attacks. According to the type of packet pairs used, we devise two different methods for detecting flooding attacks.

One may conceive a third way to detect a SYN flooding attack by using the connection state information and associating each SYN packet with the corresponding ACK that completes the three-way handshake. However, as SYN-dog is stateless, it cannot tell if the ACK is for the received SYN/ACK or just for a data packet received. Therefore, the SYN–ACK pair scheme does not work in case of SYN-dog.

3.4.1 SYN–FIN pairs

The first detection method utilizes the SYN–FIN pairs. Because a SYN packet and the corresponding FIN pass through the leaf router in the same direction (i.e., the same interface as shown in Figures 3.1 and 4.12), the SYN–FIN pair can be monitored by the same sniffer. No coordination and communication between these two sniffers are required. The first-mile SYN-dog employs only the out-bound sniffer, while the last-mile SYN-dog uses the in-bound sniffer only. Although SYN packets can be distinguished from SYN/ACK packets, there is no way to discriminate active FINs from passive FINs, since each end-host behind a leaf router may be either a client or a server. So, the SYN–FIN pairs refer to both (SYN, FIN) and (SYN/ACK, FIN). In this detection method, the SYN packets are "generalized" to include the pure SYN and SYN/ACK packets.

Under a long-running normal condition, the TCP semantics has the one-to-one correspondence between SYNs and FINs. However, in reality there can always be a discrepancy between the number of SYNs and FINs. Besides the small number of long-lived TCP sessions, the other major cause of this discrepancy lies in the occurrence of RST packets. A single RST packet can terminate a TCP session without generating any FIN packet, which violates the SYN–FIN pair's protocol behavior. RSTs are generated for two reasons: (1) *passive*, or the RST is transmitted upon arrival of a packet at a closed port; (2) *active*, or the RST is initiated by a client to abort a TCP connection.[3] Each active RST is associated with the SYN from the same session, and both of them can be seen by the same sniffer. In contrast, a passive RST cannot be associated with any SYN seen by the same sniffer as the passive RST and its corresponding SYN must go through different sniffers. Furthermore, passive RSTs may even have nothing to do with SYNs. For instance, late arrival of a data packet at the port that has already been closed, will trigger the transmission of a RST. We treat passive RSTs as a background noise.

In general, three types of SYN pairs are considered as the normal behavior of TCP in the first detection method: (SYN, FIN), (SYN/ACK, FIN) and (SYN, $\text{RST}_{\text{active}}$). Unfortunately, SYN-dog cannot distinguish active RSTs from passive ones. There are two simple but extreme ways to resolve this problem: one is to treat all RSTs as active, and the other is to treat all RSTs as passive. The first approach degrades the SYN-dog detection sensitivity, while the second raises the SYN-dog false alarm rate. To make a trade-off between detection sensitivity and false alarm rate, it is necessary to set an appropriate

[3] 1.0 Active RSTs are issued mostly by clients. In its own best interest, a server rarely sends RST packets to clients once the TCP connection is established.

threshold to filter most of the background noise. Based on our observation, under the normal condition: (1) SYNs and RSTs have a strong positive correlation; (2) the difference between the number of SYNs and that of FINs is close to the number of RSTs. These imply that passive RSTs constitute only a small percentage of the entire RSTs. So, we set the threshold to 75%, i.e., 3 out of 4 RSTs are treated as active. Moreover, for the following reason, SYN-dog can withstand the negative impact of passive RSTs that are incorrectly classified as active ones: at the end of each observation period, the CUSUM algorithm resets any negative difference between the number of SYNs and that of FINs (RSTs) to zero, so the spike of background noise is confined to one observation period only, preventing its cumulative effects.

The weakness of the SYN–FIN pairs scheme lies in its vulnerability to simple countermeasures. Once the attacker is aware of the presence of such a detection mechanism, it can paralyze the mechanism by flooding a mixture of SYNs and FINs (RSTs). Although one can argue that by doubling its flooding traffic, the attacker increases the possibility of being traced back, one may still wonder if there is a better way to overcome this shortcoming.

3.4.2 SYN–SYN/ACK pair

Fortunately, there is an alternative that is difficult for an attacker to counter. In the normal TCP three-way handshake, an out-bound SYN induces an in-bound SYN/ACK within a round-trip time. In contrast, for the flooded SYNs, because their spoofed IP source addresses are randomized, most of the corresponding SYN/ACKs will never return to the flooding sources, and hence, cannot go through the same leaf router as those flooding SYNs as shown in Figure 3.3 (mis-match part).

The second detection method makes use of SYN–SYN/ACK pairs to sniff flooding attacks. Since SYN/ACK packets are generated by the victim server, it is much more difficult for the flooding sources to evade the SYN-dog. Moreover, as compared to the SYN–FIN pair scheme, the interval between SYN and SYN/ACK is bounded by one RTT, not by the duration of a TCP session that has much larger variations.

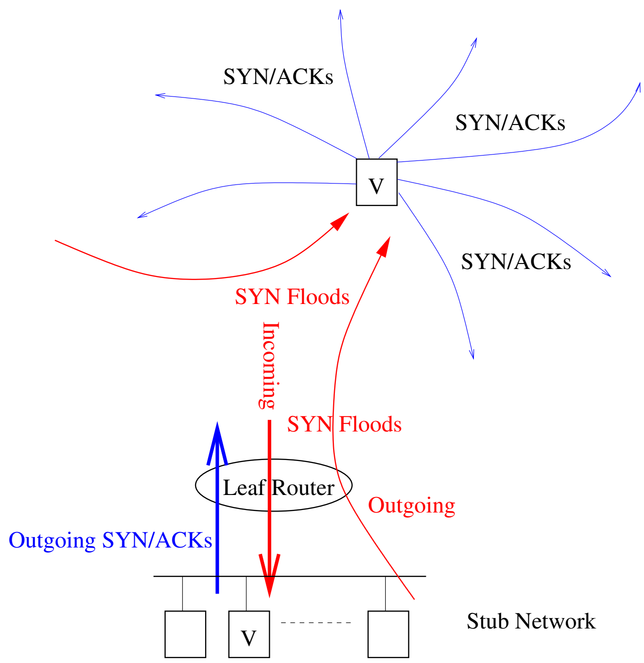


Figure 3.3: Match and mis-match between SYN–SYN/ACK pair at a leaf router

On the other hand, there are two disadvantages of the second detection method. First, unlike the first detection method, the out-bound sniffer and the in-bound sniffer must be coordinated. The out-bound sniffer maintains the count of outgoing SYNs and the in-bound sniffer keeps track of incoming SYN/ACK packets. At the end of each observation period, the count information must be exchanged between the two sniffers. Second, the SYN–SYN/ACK pair scheme is restricted to be used by the first-mile SYN-dog only, which sniffs the flooding sources inside the local stub network. It lacks the capability to issue a timely last-mile flooding warning to the network administrator of the local stub network that is under attack. The reason for this is that (1) each victim server generates a SYN/ACK in response to each SYN it received, regardless whether it is spoofed or not; (2) the incoming SYN flood and the outgoing SYN/ACKs pass through the same local leaf router. This phenomenon is illustrated in Figure 3.3 (match part). So, there is no noticeable difference between the number of incoming SYN packets and that of the outgoing SYN/ACKs generated by the victim servers until the victim servers are totally shut down and no more SYN/ACKs are generated.

It is, however, very important for the last-mile SYN-dog to detect the ongoing flooding attack in a timely manner. Therefore, the last-mile SYN-dog will rely on SYN–FIN pairs for timely detection of an incoming SYN flooding attack.

3.4.3 Summary of SYN-dog

In summary, SYN-dog plays a dual role: one as the first-mile SYN-dog for sniffing flooding sources, and the other as the last-mile SYN-dog for issuing attack warnings. To build a robust and powerful SYN-dog, both flooding detection methods are included in the SYN-dog. The SYN–SYN/ACK pair method is employed by the first-mile SYN-dog to sniff flooding sources inside the local stub network, while the SYN–FIN pairs method is used by the last-mile SYN-dog to detect incipient flooding attacks, and issue a warning to the local network administrator.

3.5 Statistical Attack Detection

SYN-dog belongs to the commonly-known network-based ID system: an intruder will be singled out if its behavior is noticeably different from that of a legitimate user. Like most statistical anomaly-detection systems, we compare the observed sequence with the profile that represents the user's normal behavior, and detect any significant deviation from the normal behavior. However, unlike the traditional network ID system that *passively* monitors bidirectional traffic streams on network links, SYN-dog is transparently interposed at a leaf router and is implemented as an integrated component of the leaf router.

The burstiness of TCP connection request arrivals [39] makes the detection of attacks much harder, since there is no natural length of burst for self-similar traffic. Furthermore, the normal TCP-arrival pattern is site- and time-dependent. However, the strong positive correlation between SYN–FIN (RST) and SYN–SYN/ACK pairs clearly indicates the presence of SYN flooding. According to the specification of TCP/IP protocol [104], in its normal operation, a FIN (RST) is paired with a SYN at the end of data transmission, and the arrival of a SYN/ACK in the reverse direction follows the transmission of a SYN within one RTT. However, under a SYN flooding attack, this SYN–FIN (RST) or SYN–SYN/ACK pair's behavior will deviate significantly from the normal operation.

3.5.1 Data Sampling and Detection Mechanism

We collect the numbers of SYN, SYN/ACK, and FIN (RST) packets during every observation period t_0 at leaf routers, which determines the detection resolution. In order to relate the SYN and FIN (RST) packets of the same connection, the sampling time of FIN (RST) is delayed by t_d after SYN is sampled, where t_d is so chosen that a significant portion of connections requested during the SYN sampling period terminate in the corresponding FIN (RST) sampling period. Recent Internet traffic measurements [109] have shown that most of TCP connections last 12–19 seconds, so we set the sampling delay t_d to 10 seconds. In contrast, since most RTTs are less than 0.5 second, we start the collection of SYNs at the out-bound sniffer and SYN/ACKs at the in-bound sniffer simultaneously. To balance the detection resolution and the algorithm's stability and accuracy, we set t_0 to 10 seconds. Note, however, that both parameters are tunable and our algorithm is not very sensitive to this choice.

Under the normal condition, the difference between the collected number of SYNs and FINs (RSTs) or SYN/ACKs is very small, as compared to the total number of TCP connection requests. This observation holds in spite of the fact that the total number of TCP connection requests may be bursty on a small time scale, and slowly-varying on a large time scale. In other words, the correlation between the numbers of SYNs and FINs (RSTs) or SYN/ACKs is not sensitive to the request arrival process. The results presented in Section 3.6.1 clearly show that the consistent synchronization between SYNs and FINs (RSTs) or SYN/ACKs is independent of the sites and time-of-day.

Under SYN flooding attacks, the flooding SYN traffic has significant regularity and semantics that can be filtered out. Recent experiments with SYN attacks on commercial platforms [81] show that the minimum flooding rate to overwhelm an unprotected server is 500 SYN packets per second. Even with a specialized firewall designed to resist against SYN flooding, a server can be disabled by a flood of 22,000 packets per second [81]. To bring down the victim server for 10 minutes, for example, attackers must collectively inject at least 300,000 SYN packets. During the same time period, however, the numbers of counted FINs (RSTs) and SYN/ACKs remain largely unchanged. Therefore, there will be much more SYNs than FINs (RSTs) or SYN/ACKs collected during the flooding period. The difference between the numbers of SYNs and FINs (RSTs) or SYN/ACKs will increase dramatically, and remain large during the whole flooding period, which typically lasts for several minutes [75]. So, the occurrence of a large difference between the numbers of SYNs and FINs (RSTs) or SYN/ACKs indicates a possible SYN flooding attack. This will be used in our attack detection.

3.5.2 The CUSUM Algorithm

The CUSUM algorithm is applied to both of the two detection methods. For ease of presentation, we only show how it works in the SYN–FIN pair scheme, which is similar to the SYN–SYN/ACK pair scheme except that the collection of FINs (RSTs) is replaced with that of SYN/ACKs.

Let $\{\Delta_n, n = 0, 1, \cdots\}$ be the number of SYNs minus that of the corresponding FINs (RSTs) collected within one sampling period. To further alleviate its dependence on the time, access pattern and size of the network, $\{\Delta_n\}$ is normalized by the average number, $\bar{F}$,[4] of FINs (RSTs) during each sampling period. $\bar{F}$ can be estimated in real time and updated periodically. An example of recursive estimation and update of $\bar{F}$ is:

$$\bar{F}(n) = \alpha\bar{F}(n-1) + (1-\alpha)\text{FIN (RST)}(n), \tag{3.1}$$

where n is the discrete time index and α is a constant, whose default value is 0.01, lying strictly between 0 and 1 that represents the memory in the estimation.

Let $X_n = \Delta_n/\bar{F}$. Under normal condition, the mean of X_n, denoted as c, is much less than 1 and close to 0. $\{X_n\}$ is no longer dependent on the network size or time-of-day. Its dynamics are solely the consequence of the TCP protocol specification. So, we can consider $\{X_n\}$ as a stationary random process.

Our attack detection algorithm is based on the Sequential Change Point Detection [13]. The objective of Change Point Detection is to determine if the observed time series is statistically homogeneous, and if not, to find the point in time when the change happens. It has been studied extensively by statisticians. See [13] and [19] for a good survey. There have been various tests for different problems. They can be largely divided into two categories: posterior and sequential. Posterior tests are done off-line where the entire data is collected first and then a decision of homogeneity or a change point is made based on the analysis of all the collected data. On the other hand, sequential tests are done on-line with the data presented sequentially and the decisions are made on-the-fly.

We adopt a sequential test for quicker response when an attack occurs. It also saves memory and computation. One difficulty, however, is the modeling of $\{X_n\}$. Despite a

[4] The average number of SYN/ACKs is represented as $\bar{K}$.

number of previous results on the modeling of TCP connection request arrivals [25, 28, 39, 87, 98], there is no consensus on whether it should be modeled as self-similar or Poisson. For dynamic and complex systems like the Internet, it may not be possible to model the total number of TCP connection request arrivals by a simple parametric description. So, we seek robust tests which are not model-specific. In fact, non-parametric methods fit this requirement very well. Specifically, we use the non-parametric CUSUM (Cumulative Sum) method [19] for detection of attacks. This method enjoys all the virtues of sequential and non-parametric tests, and the computation load is very light. When the time series is i.i.d. with a parametric model, CUSUM is asymptotically optimal for a wide range of Change Point Detection problems [13, 19].

$\{X_n\}$ is assumed to satisfy the following two conditions.

C1: $\{X_n\}$ is ψ-mixing, meaning that the $\psi(s)$ parameters, defined below, approach 0 as $s \to \infty$:

$$\psi(s) \stackrel{def}{=} \sup_{t \geq 1} \sup_{\substack{A \in \mathcal{F}_1^t, \\ B \in \mathcal{F}_{t+s}^{\infty}, \\ P(A)P(B) \neq 0}} \left|\frac{P(AB)}{P(A)P(B)} - 1\right|, \tag{3.2}$$

where $\mathcal{F}_1^t$ is the σ-algebra generated by $\{X_1, X_2, \cdots, X_t\}$ and $\mathcal{F}_{t+s}^{\infty}$ is the σ-algebra generated by $\{X_{t+s}, X_{t+s+1}, \cdots\}$. $\psi(s)$ is affected by the dependency among the $\{X_n\}$ samples: highly dependent $\{X_n\}$ has $\psi(s)$ that decays slowly as $s \to 0$.

C2: The marginal distribution of $\{X_n\}$ satisfies the following regularity condition: $\exists t > 0$ such that $E(e^{tX_n}) < \infty$.

The details of these conditions can be found in [19]. Note that ψ-mixing is a much looser requirement than independence, and X_n being ψ-mixing only indicates that X_n is not “extremely” dependent. In practice, both conditions are mild and easily satisfiable, even for long-range dependent arrival processes. In general, $E(X_n) = c \ll 1$. We choose a parameter a that is the upper bound of c, i.e., $a > c$, and define $\tilde{X}_n = X_n - a$ so that it has a negative mean during normal operation. When an attack takes place, $\tilde{X}_n$ will suddenly become a large positive number. Suppose, during an attack, the increase in the mean of $\tilde{X}_n$ can be lower-bounded by h. Our change detection is based on the observation of $h \gg c$.

Let

$$\begin{aligned} y_n &= (y_{n-1} + \tilde{X}_n)^+, \\ y_0 &= 0, \end{aligned} \tag{3.3}$$

where x^+ is equal to x if $x > 0$ and 0 otherwise. The meaning of y_n can also be understood as follows: if we define $S_k = \sum_{i=1}^{k} \tilde{X}_i$, with $S_0 = 0$ at the beginning, it is straightforward to show that

$$y_n = S_n - \min_{1 \le k \le n} S_k, \tag{3.4}$$

i.e., the maximum continuous increment until time n. A large $\{y_n\}$ is a strong indication of an attack. Since Eq. (3.3) is recurrent and much easier to compute than Eq. (3.4), we will use it in making detection decisions.

Let $d_N(.)$ be the decision at time n: '0' for normal operation (homogeneity) and '1' for attack (a change occurs). Here N represents the flooding threshold:

$$d_N(y_n) = \begin{cases} 0 & \text{if } y_n \le N, \\ 1 & \text{if } y_n > N. \end{cases} \tag{3.5}$$

In other words, $d_N(y_n) = I(Y_n > N)$, where $I(.)$ is the indicator function. The purpose of introducing a is to offset the possible positive mean in $\{X_n\}$ caused by network anomalies so that the test statistic y_n will be reset to zero frequently and will not accumulate with time.

Let P_m and E_m be the probability measure and the expected value generated by $\{\tilde{X}_n\}$ with the attack occurring at time m (change point at time m); let P_∞ and E_∞ be the counterparts without any attack (no change point). There are two fundamental performance measures for the sequential change point detection.

False alarm time (the time without false alarm): the time duration with no false alarm reported when there is no attack.

Detection time: the detection delay after the attack starts.

One would want the second measure to be as small as possible while keeping the first measure as large as possible. However, they are conflicting goals and cannot be simultaneously achieved. Therefore, the design philosophy of a statistical change point detection is to minimize the detection time subject to a certain false alarm tolerance. In order to compare the performance of different detection schemes, some criteria of false alarms must be specified, like average time between two consecutive false alarms, worst-case false alarm time, and so on. An algorithm is said to be *optimal* with respect to a certain criterion if it minimizes the detection time for an attack among all the detection schemes subject to the false alarm constraint. The CUSUM rule has been shown to be asymptotically optimal with respect to the worst-case mean false alarm time in the change point detection problems involving a known parametric model and independent observations [13].

Due to the lack of a complete model for $\{\tilde{X}_n\}$, it is difficult to discuss optimality. The choice of CUSUM is based on its simplicity in computation and non-parametric implementation, as well as its generally excellent performance. It has been shown in [19] that, with the choice of a, the upper bound in case of normal operation, and N, the flooding threshold, as $N \to \infty$, we have

$$\sup_n |\ln P_\infty(d_N(n) = 1)| = O(N), \tag{3.6}$$

which is equivalent to

$$P_\infty\{d_N(n) = 1\} = c_1 \exp(-c_2 N). \tag{3.7}$$

where c_1 and c_2 are constants, depending on the marginal distribution and mixing coefficients of $\{\tilde{X}_n\}$. In other words, the time between consecutive false alarms grows exponentially with N. The burstiness of traffic is reflected by the mixing coefficients $\psi(s)$, and thus, does impact the detection performance. However, the constants c_1 and c_2 only play a secondary role and can be ignored in practice.

In order to study the detection time, let us define

$$\begin{aligned} \tau_N &= \inf\{n : d_N(.) = 1\}, \\ \rho_N &= \frac{(\tau_N - m)^+}{N}, \end{aligned} \tag{3.8}$$

where ρ_N represents the normalized detection time after a change occurs and m represents the starting time of the attack. In CUSUM, for any $m \geq 1$, if h is the actual increase in the mean of $\{\tilde{X}_n\}$ during an attack, we have

$$\rho_N \to \gamma = \frac{1}{h - |c - a|}, \tag{3.9}$$

where $h - |c - a|$ is the mean of $\{\tilde{X}_n\}$ when $n > m$ (after an attack starts). However, since h is a bound rather than a true value, the above is a conservative estimation (upper bound) of the actual detection time.

3.5.3 Robustness against Network Anomalies

While there is no strict one-to-one match, under the normal condition, a very strong positive correlation between the numbers of SYNs and FINs (RSTs) or SYN/ACKs does exist as shown in Section 3.6.1. The discrepancy between the numbers of SYN and FIN (RST) or SYN/ACK packets lies in SYN losses and subsequent retransmissions. The SYN losses are caused by various network anomalies, including network congestion, routing loops and link failures. Clearly, these network anomalies reduce the sniffing sensitivity of SYN-dog.

Fortunately, these network anomalies are triggered by unrelated events; and to date there exist few evidences indicating that these different network anomalies are closely correlated. The recent Internet measurement studies have shown that: (1) there is very few congestions inside the core of Internet, where the bandwidth over-provisioning has been widely used and the link utilization varies from 3 to 60% [16]; (2) the majority of routing loops last shorter than 10 seconds [53]; and (3) link failures are fairly well spread even in the time scale of minutes [52]. Therefore, these randomly-occurring network anomalies can be treated as white noise. To offset the effect of white noise, a safety margin can be added when the normal upper bound, a, is set, without jeopardizing the sniffing sensitivity. Only a severe network congestion, long lasting routing loops, and bursty occurrence of link failures — which rarely happen — can confuse the SYN-dog to issue false alarms.

3.5.4 Parameter Specification

To implement the CUSUM algorithm, we only need to specify three tunable parameters: a, the upper bound in case of normal operation; h, the lower bound of the increase in case of an attack; and, N, the flooding threshold.

The CUSUM algorithm requires $E(\tilde{X}_n) < 0$ before the change point, and $E(\tilde{X}_n) > 0$ after it, i.e., $a > c$ and $h > a$. Based on the discussion in the previous section, to ensure a long false alarm time and make it independent of network size and access pattern, we set $h = 2a$ in our design. As the last-mile SYN-dog that utilizes SYN–FIN pairs for flooding detection monitors the incoming traffic, all the SYN flooding packets converge, and therefore, a large difference between the numbers of SYN and FIN (RST) packets is easily observable with $h \gg c$. In this case, the detection is not sensitive to the choice of a. With a large safe margin, we can simply choose $a = 1$ and $h = 2$.

In contrast, as the first-mile SYN-dog that employs SYN–SYN/ACK pairs for flooding detection monitors the outgoing SYN and incoming SYN/ACK traffic, only part of the flooding SYN packets can be seen by each detector because the attack may be initiated from many sites simultaneously. Thus, a proper choice of a is more important. To balance the detection sensitivity and false alarm rate, we set $a = 0.35$ and $h = 0.7$. Note that the choices of a and h are independent of the network size and access pattern. In doing so, a universal false alarm rate can be realized for easy implementability of our detection and sniffing mechanism. On the other hand, in practice, the network administrator of the involved edge router can incorporate site-specific information so that the algorithm can achieve higher detection performance.

Based on a and h, the flooding threshold N can be specified as follows: (i) assume $c = 0$, and γ can thus be obtained from Eq. (3.9); and (ii) specify a target detection time (i.e., the product of γ and N) such that the flooding threshold N is determined by Eq. (8). We choose t_0 as the designed detection time for the last-mile SYN-dog, hence $\gamma = 1$ and $N = 1$. In contrast, we choose $3t_0$ as the counterpart for the first-mile SYN-dog, hence $\gamma = 2.86$ and $N = 1.05$.[5] Compared to the last-mile SYN-dog, the short detection time of the first-mile SYN-dog is not so crucial to the victim: the revelation of the flooding sources is more valuable, although it may take longer time. Note that the value of N is partially determined by the designed detection time, so it may not be larger than the value of h.

It is worth noting that our algorithm is to check the cumulative effect of an attack. So, it can detect attacks with the SYN flooding rate less than h at the expense of a longer response time. The actual lower bound of detection sensitivity in terms of SYN flooding rate, f_{min}, can be given as

$$f_{min} = (a - c) \cdot \frac{\bar{F}}{t_0}. \tag{3.10}$$

[5] N may not seem to be large but Eq. (3.9) can serve as an approximation.

Table 3.1: A summary of the trace features

Trace	Starting time	Traffic type
DEC	2:00, Thu Mar 9, 1995	Bi-directional
Harvard	12:39, Thu Mar 13, 1997	Bi-directional
UNC-in	19:30, Wed Sept 27, 2000	Uni-directional
UNC-out	19:30, Wed Sept 27, 2000	Uni-directional
Auckland-in	14:36, Thur, Dec 5, 2000	Uni-directional
Auckland-out	14:36, Thur Dec 5, 2000	Uni-directional

Furthermore, the detection capability is not sensitive to the flooding pattern: it can detect the attacks with both constant and bursty flooding rates. The effectiveness of this detection is evaluated by trace-driven simulations.

3.6 Performance Evaluation

To evaluate and validate the SYN-dog, we have conducted trace-driven simulation experiments. The trace data we used are collected from four different sites at different times. The first trace was gathered at DEC's (now HP) primary Internet access point, which is an Ethernet DMZ network. It contains an hour's worth of all wide-area traffic between DEC Western Research Lab and the Internet on March 9th, 1995. The second trace was taken on March 13th, 1997 on a 10 Mbps Ethernet connecting Harvard's main campus to the Internet, which is a half-hour trace. The third set was obtained by placing network monitors on the high-speed link (OC-12, 622 Mpbs) that connects the University of North Carolina at Chapel Hill (UNC) campus network to the rest of the world. The trace was collected on September 27th, 2000. The fourth set was collected at the Internet access link that connects the University of Auckland at New Zealand to the rest of the world. The tracing ran from 14:36 to 17:47 on Thursday, December 5th, 2000.

The traces used in our experiments are summarized in Table 3.1. Note that the DEC and Harvard traces are mixed traffic collections in both directions, but the UNC and Auckland traces are uni-directional: UNC-in and Auckland-in collected the traffic data from the Internet to the UNC and Auckland campus networks, respectively, while UNC-out and Auckland-out collected the traffic data from the UNC and Auckland campus networks to the Internet, respectively.

3.6.1 Normal Traffic Behavior

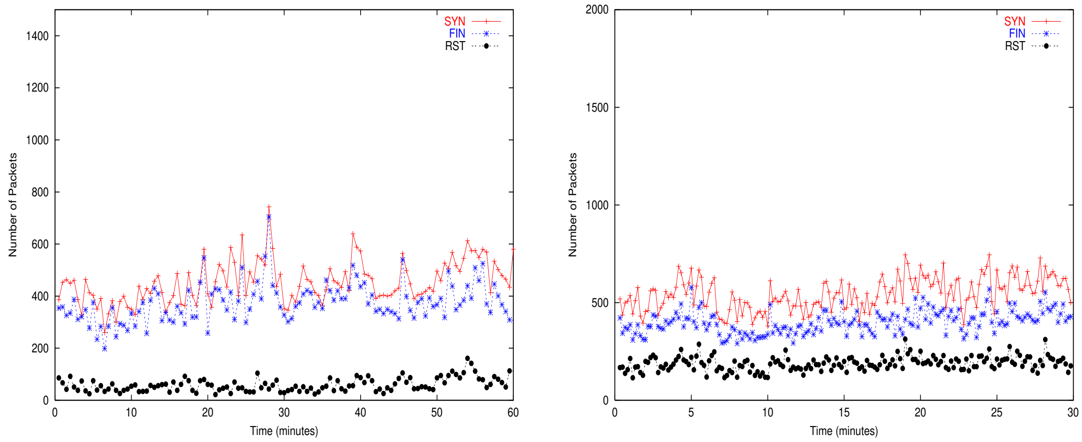
The three sets of traces represent the normal traffic behaviors at the exchange points between different stub networks and the Internet at different times. We parse the traces and extract the TCP SYN, SYN/ACK, FIN and RST packets from the TCP traffic.

SYN–FIN Pairs

The dynamics of SYN,[6] FIN and RST packets at the DEC site are illustrated in Figure 3.5 (a), and the corresponding result from the Harvard trace is illustrated in Figure 3.5 (b). Those from UNC-in and UNC-out are in Figures 3.5 (c) and 3.6 (a), and Auckland-in and Auckland-out are in Figures 3.6 (b) and (c), respectively. They clearly show the consistent synchronization between SYN and FIN (RST) packets. The consistency indicates that the synchronization is an inherent traffic behavior and independent of time and sites.

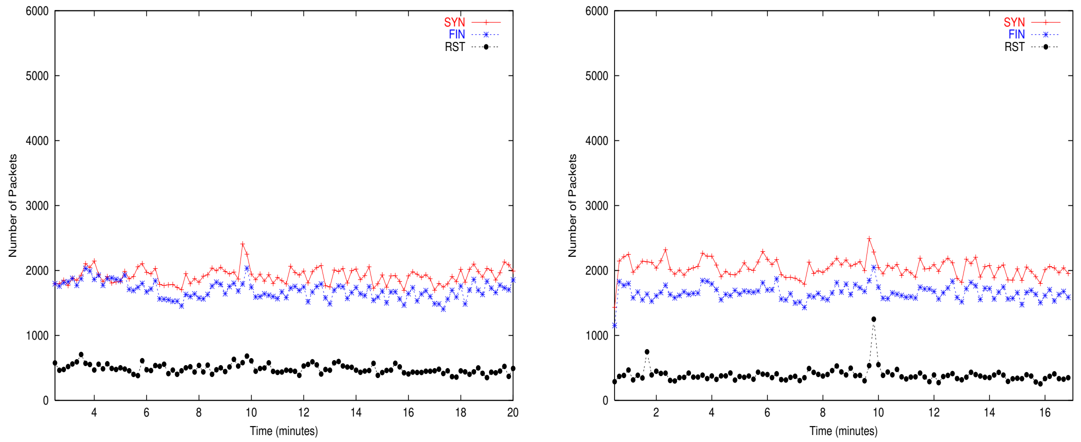
We have applied the CUSUM algorithm on all the available traces without adding attacks. The test statistics, $\{y_n\}$, for all traces are plotted in Figure 3.7. For the Harvard and UNC traces, y_n's are constantly zeros. For the Auckland traces, more than 99% y_n's stay at zero. The isolated bursts in y_n are always much smaller than the threshold $N = 1.05$: the maximal spikes of y_n in Auckland-in and Auckland-out are 0.32 and 0.27, respectively. So, no false alarms are reported.

[6]It is generalized SYNs, including SYNs and SYN/ACKs.



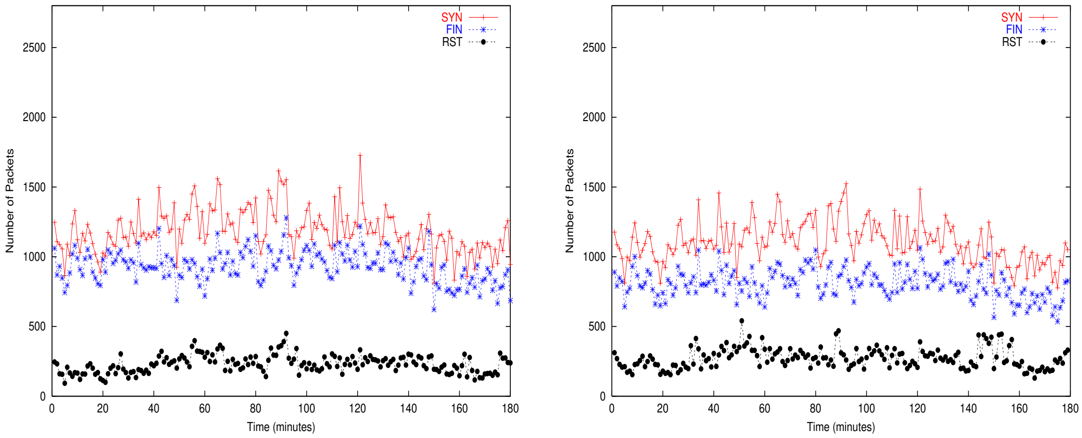
(a) DEC (b) Harvard

Figure 3.4: The dynamics of SYN and FIN (RST) packets (part I)



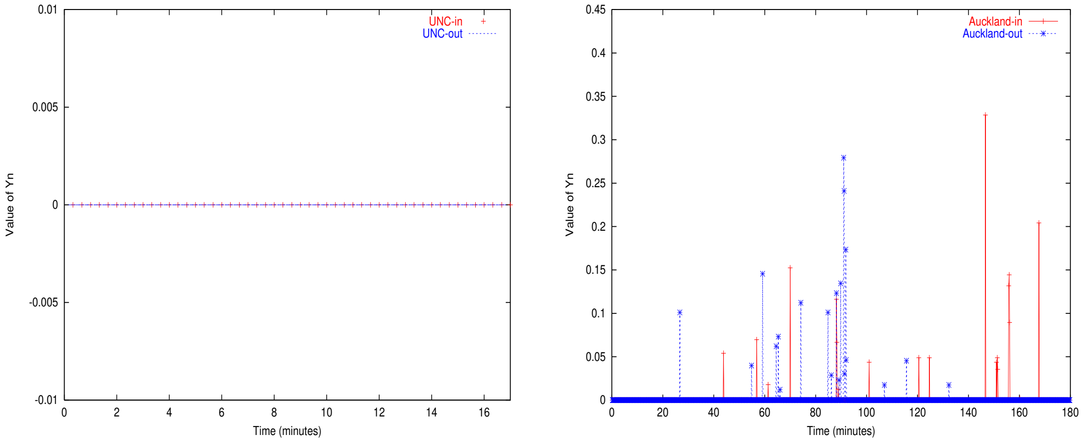
(a) UNC-in (b) UNC-out

Figure 3.5: The dynamics of SYN and FIN (RST) packets (part II)



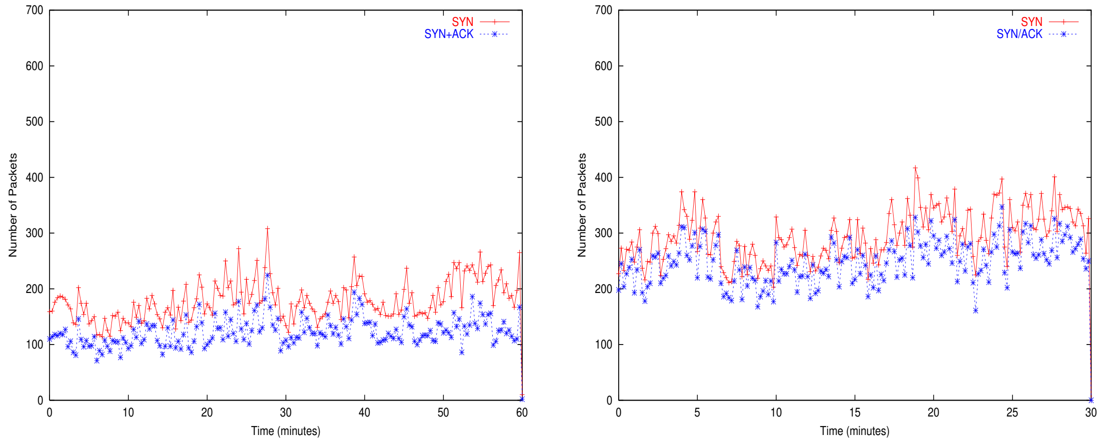
(a) Auckland-in (b) Auckland-out

Figure 3.6: The dynamics of SYN and FIN (RST) packets (part III)



(a) UNC (b) Auckland

Figure 3.7: CUSUM test statistics under normal operation



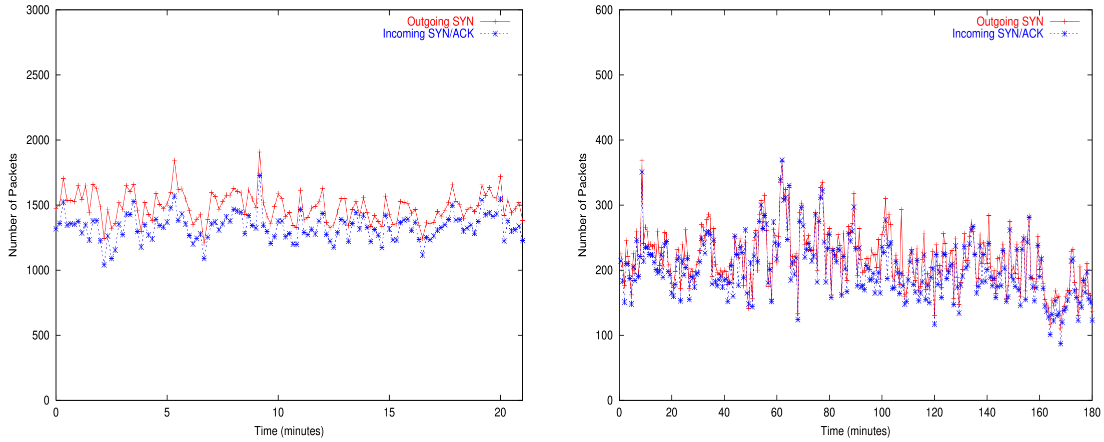
(a) DEC (b) Harvard

Figure 3.8: The dynamics of SYN and SYN/ACK packets at DEC and Harvard

SYN—SYN/ACK pair

The dynamics of SYN and SYN/ACK packets at the DEC and Harvard sites are illustrated in Figures 3.8 (a) and (b), respectively. The outgoing SYNs and incoming SYN/ACKs from the UNC and Auckland traces are shown in Figures 3.9 (a) and (b). As with SYN–FIN pairs, these figures clearly demonstrate a consistent positive correlation between SYN and SYN/ACK packets. The consistency indicates that the strong positive correlation is also a distinct protocol behavior and independent of time and sites. Note that in the figures of the DEC and Harvard traces, SYNs and SYN/ACKs are collected from both directions, instead of "Outgoing SYN" and "Incoming SYN/ACK" as shown in the UNC and Auckland traces.

Also, we have applied the CUSUM algorithm on the Harvard, UNC and Auckland traces without adding flooding attacks. The test statistics, $\{y_n\}$, for the Harvard and UNC traces are plotted in Figures 3.10 (a) and (b); that for the Auckland trace is plotted in Figure 3.10 (c). As expected, for all the traces tested, y_n's are mostly zeros. Among the isolated spikes of y_n in the Harvard trace, the maximum is about 0.05; the maximal



(a) UNC (b) Auckland

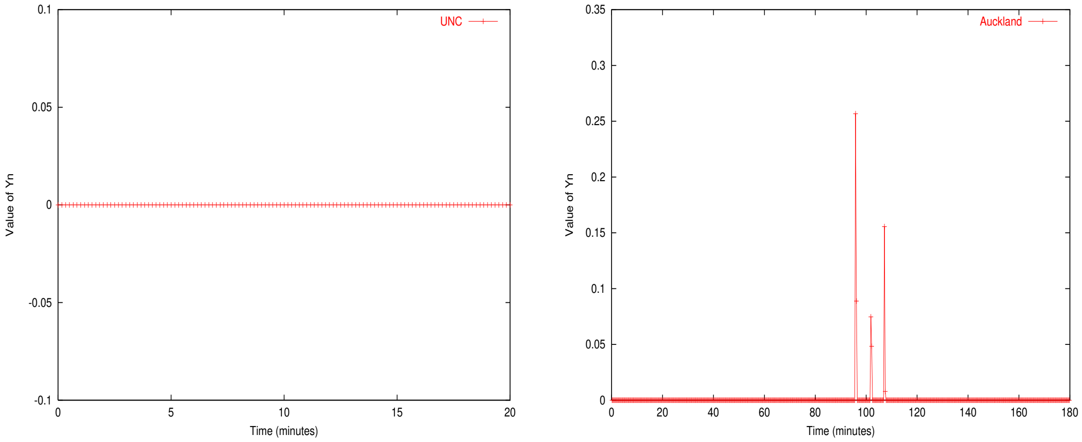
Figure 3.9: The dynamics of SYN and SYN/ACK packets at UNC and Auckland

spike of y_n in the Auckland trace is about 0.26. Both are much smaller than the flooding threshold $N = 1.05$. So, no false alarms are reported.

3.6.2 SYN Flooding Detection

With the appearance of Trinoo, which only implements UDP packet flooding, many tools have been developed to create DDoS attacks. Most of them, such as Tribe Flood Network (TFN), TFN2K, Stacheldraht, Trinity, Plague and Shaft, generate TCP SYN flooding attacks and randomize all 32 bits of the source IP address [32, 33]. Although these DDoS attack tools employ different ways to coordinate attacks with the goal of achieving robust and covert DDoS attacks, their flooding behaviors are similar in that the SYN packets are continuously bombarded to the victim.

In the SYN flooding detection experiments, the UNC and Auckland 2000 traces are used as the normal background traffic. Among them, UNC-in or Auckland-in is used for incoming background traffic, and UNC-out or Auckland-out is for outgoing background traffic. The flooding traffic is mixed with the normal traffic, and the SYN-dog at the leaf router is simulated, as shown in Figure 3.11. Because the non-parametric CUSUM method



(a) UNC (b) Auckland

Figure 3.10: CUSUM test statistics under normal operation at: UNC and Auckland

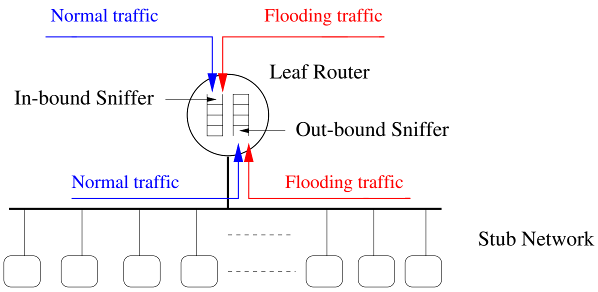
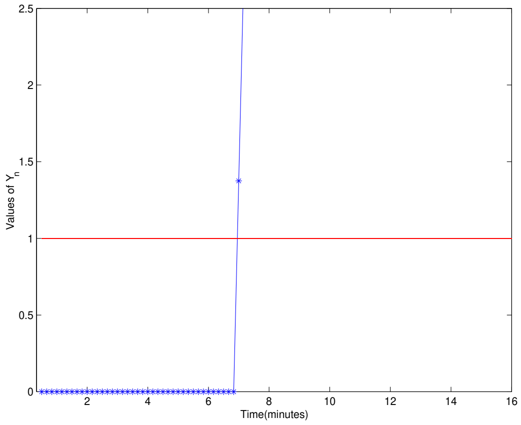
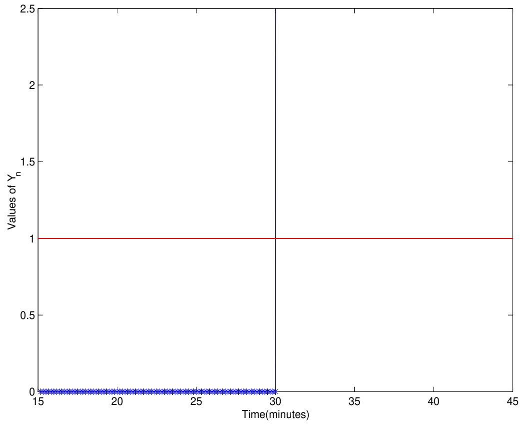


Figure 3.11: The trace-simulation flooding attack experiment



(a) UNC case



(b) Auckland case

Figure 3.12: SYN flooding detection sensitivity at the last-mile SYN-dog

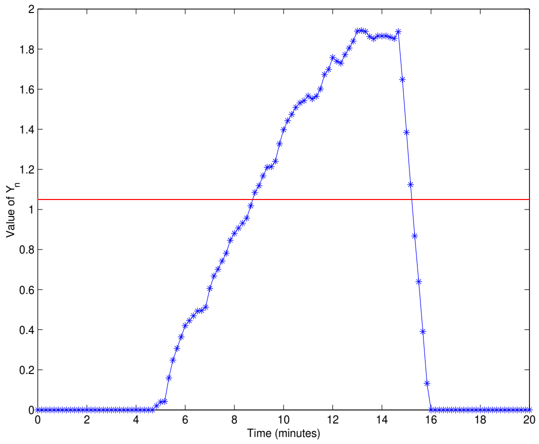
is used for detection of flooding attacks, the flooding traffic pattern or its transient behavior (bursty or not) does not affect the detection sensitivity. Rather, the detection sensitivity depends only on the total volume of flooding traffic. So, without loss of generality, we assume that the flooding rate is constant.

In DDoS attacks, the flooding traffic seen by the first-mile and the last-mile SYN-dogs is quite different. The flooding traffic passing through the last-mile SYN-dog is the aggregation of the flooding traffic from all distributed attack sources, allowing for much easier detection of an attack. However, the flooding detection at the first-mile SYN-dog is much more difficult. In a large-scale DDoS attack, the flooding sources can be so coordinated that the traffic from each flooding source may not be noticeable at all. Suppose the minimum SYN flooding traffic to bring down a TCP server is V packets per second. Then, the flooding rate at the last-mile SYN-dog is V, but the flooding rate seen by the first-mile SYN-dog may be much smaller than this.

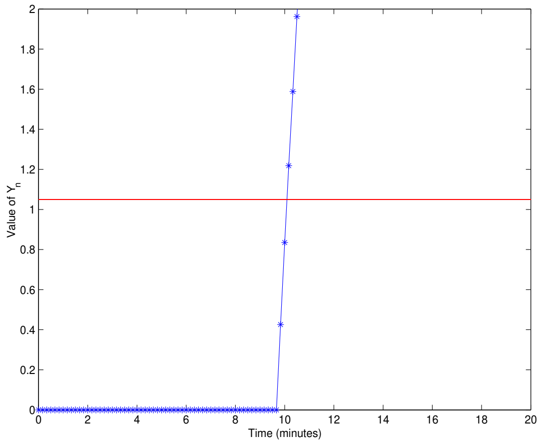
We assume that the flooding traffic is evenly distributed among different flooding sources and there is only one flooding source inside each stub network. The flooding rate seen by the first-mile SYN-dog, f_i, equals the individual flooding rate inside the same stub network. Therefore, f_i is determined by $\frac{V}{A_s}$, where A_s is the total number of the stub networks that contain flooding sources. This setting is intended to "hide" the attack from the first-mile SYN-dog. That is, the less the flooding sources inside the stub network, the less flooding traffic seen by the first-mile SYN-dog and the harder to detect the flooding attack. The flooding duration in all experiments is set to 10 minutes, a typical attack duration observed in the Internet [75]. The starting time of flooding attacks in the UNC traces is randomly chosen between 1 and 9 minutes, but the starting time in the Auckland traces lies between 3 and 166 minutes.

We first examine the detection sensitivity at the last-mile SYN-dog, which employs SYN–FIN pairs as its detection method. To demonstrate the high sensitivity of last-mile SYN-dog to SYN flooding, the flooding rate V is set to its minimum, 500 SYNs per second. The simulation results are plotted in Figures 3.12 (a) and (b), showing that the cumulative sum y_n exceeds the flooding threshold "1" in one observation period, i.e., the fastest response can be achieved, in the Auckland and UNC trace cases, respectively. So, the last-mile SYN-dog of the Auckland case can detect the SYN flooding attack in 10 seconds, and so does the last-mile of the UNC case. Once the flooding attack is detected, a defense system like SynDefender can be triggered to protect the victim from the flooding attack. To paralyze the defense system at the victim, attackers have to increase their flooding rate, and the first-mile SYN-dog will then be more likely to detect and locate the flooding sources inside the stub network.

To examine the detection sensitivity of the first-mile SYN-dog, which employs SYN–SYN/ACK pairs to detect attacks, we vary the flooding rate f_i seen by the first-mile SYN-dog, i.e., the individual flooding rate inside the stub network. As the last-mile detection is much easier than the first mile, we only study the detection probability and detection time for the latter. We conduct the SYN flooding detection experiments on the UNC and



(a) 35 SYNs per second



(b) 80 SYNs per second

Figure 3.13: SYN flooding detection sensitivity of the SYN-dog at UNC

Auckland traces.

The UNC Case

Using the UNC traces as the background traffic, we observe the dynamics of y_n. Figures 3.13 (a), (b) and (c) plot the dynamic behaviors of y_n when f_i is set to 35, 60 and 80 SYNs per second, respectively. The accumulative effects of SYN flooding are clearly shown in these figures. In the cases of 60 and 80 SYNs per second, the first-mile SYN-dog

Table 3.2: Detection Performance of the first-mile SYN-dog at UNC

f_i	Detection Prob.	Detection Time
35	0.8	24.43
37	1.0	18.75
40	1.0	12.25
50	1.0	5.95
60	1.0	4.05
70	1.0	2.80
80	1.0	2.05
100	1.0	1.45
120	1.0	1.0

Table 3.3: Detection Performance of the first-mile SYN-dog at Auckland

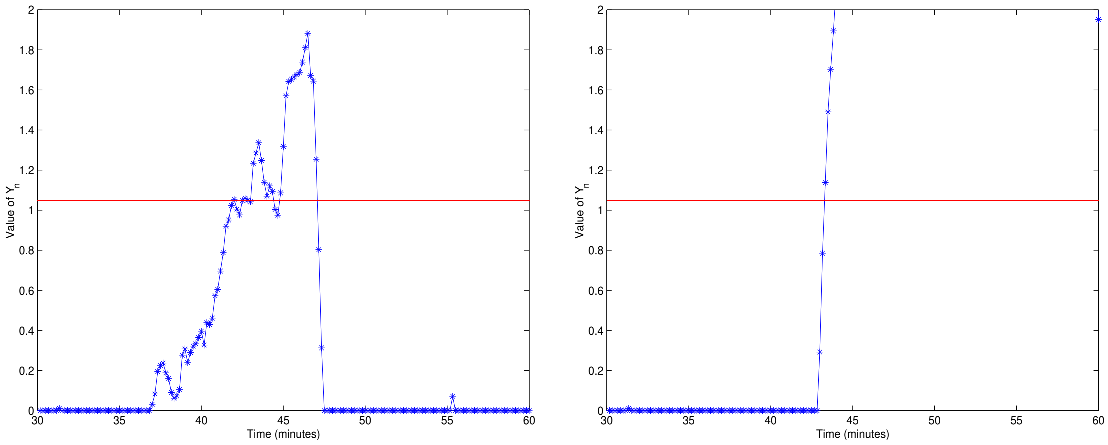
f_i	Detection Prob.	Detection Time
1.5	0.65	27.1
1.75	1.0	13.8
2	1.0	8.0
2.5	1.0	4.1
3	1.0	2.5
4	1.0	1.5
5	1.0	1.0

can detect the SYN flooding attack in 4 and 2 observation periods, respectively. However, in the case of 35 SYNs per second, the first-mile takes a much longer time (about 24 observation periods, i.e., 4 minutes) to exceed the flooding threshold of 1.05. The detection performance of the first-mile SYN-dog in the context of the UNC traces is summarized in Table 3.2, which lists the detection probabilities and detection times for different f_i values. The detection time is measured in number of the observation period t_0, which is set to 10 seconds.

Clearly, larger flooding rates lead to faster and easier detection of attacks. According to Eq. (3.10), the lower detection bound is about 37 SYNs per second in this simulation scenario. If we implement the same SYN-dog at a smaller subnet, $\bar{K}$ — the average number of incoming SYN/ACKs — will be smaller, so we can achieve higher detection sensitivity. This is confirmed by the study of the Auckland traces, which is presented in the next section.

The Auckland Case

In the case of Auckland traces, the dynamic behaviors of y_n are illustrated in Figure 3.14 when f_i is set to 1.5, 3 and 4 SYNs per second, respectively. In the case of 1.5 SYNs per second, the first-mile SYN-dog can detect the SYN flooding attack in about 27 observation periods. In contrast, at the flooding rate of 3 or 4 SYNs per second, the



(a) 1.5 SYNs per second

(b) 3 SYNs per second

Figure 3.14: SYN flooding detection sensitivity of the first-mile SYN-dog at Auckland

first-mile SYN-dog takes a much shorter time (3 or 2 observation periods, respectively) to detect the ongoing flooding. The detection performance of the first-mile SYN-dog for the context of Auckland traces is summarized in Table 3.3. Since $\bar{K}$ of the Auckland trace is much smaller than that of the UNC trace, the lower detection bound is reduced significantly from 35 to 1.5 SYNs per second.

Discussion

Due to its proximity to the flooding sources, once the first-mile SYN-dog detects the ongoing flooding traffic, it can take further steps to pinpoint the flooding sources inside the stub network, for example, by filtering the MAC addresses of IP packets whose source addresses are spoofed.

From the detectable flooding rate, we can determine the efficacy of our algorithm in detecting distributed SYN flooding attacks. To attack a protected server, the aggregate flooding rate V should be larger than 22,000 requests per second [81]. In the UNC case,

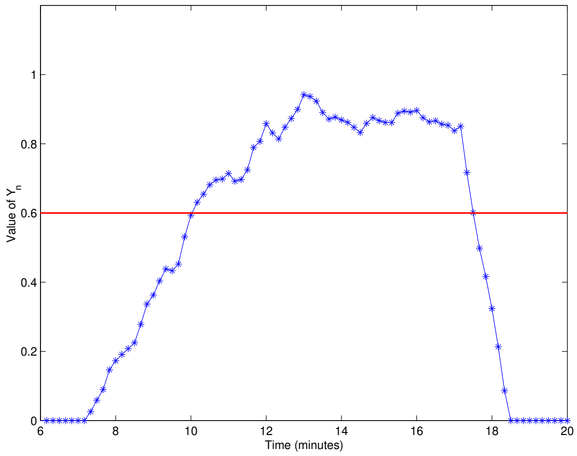


Figure 3.15: Improvement of flooding detection sensitivity

the lower detection bound is 35, and A_s can be as large as 628 stub networks like the UNC case. Considering the fact that the UNC stub network consists of over 35,000 users [98], it clearly demonstrates the utility and power of SYN-dog. In the Auckland case, the lower detection bound is 1.5, and hence, A_s has to be as large as 14,666 medium size stub networks like the Auckland case. Source address spoofing requires that the attack software open a *raw* network socket, so the attacker must have root access on end hosts. Although the attacker can simultaneously initiate the flooding attacks from (possibly many) machines in several ISPs, it is much harder to launch the attacks from a similar large number of stub networks due to access limit.

For the time being, we set the parameters to be independent of network size and access pattern. In practice, the network administrator of the involved leaf router can incorporate site-specific information so that the algorithm can achieve higher detection performance. For instance, in the UNC case, we can reduce a, the upper bound in case of normal operation, from 0.35 to 0.2 and N, the flooding threshold, from 1.05 to 0.6 without incurring additional false alarms. Then, the lower detection bound f_{min} decreases from 35 to 15 SYNs per second, and the detection sensitivity is greatly improved. The dynamics of y_n

for the case $f_i = 15$ is shown in Figure 3.15.

In summary, SYN-dog not only achieves fast detection and high detection accuracy, but also can be easily implementable and broadly applicable.

3.7 Conclusion

We developed and evaluated a simple and robust mechanism to sniff SYN (including reflected SYN/ACK) flooding attacks, which is installed at leaf routers. SYN-dog utilizes the SYN–FIN or SYN–SYN/ACK pair's behavior that is invariant under various arrival models and independent of sites and time-of-day. The distinct features of SYN-dog include: (1) it is stateless and requires low computation overhead, making itself immune to flooding attacks; (2) the non-parametric CUSUM method is employed, making the detection robust; (3) it is insensitive to site and access pattern; and (4) it does not degrade the end-to-end TCP performance.

The efficacy of SYN-dog is evaluated and validated by trace-driven simulations. The evaluation results show that the SYN-dog achieves high detection accuracy and short detection time. Moreover, once the first-mile SYN-dog detects the ongoing flooding traffic, information about the location of flooding sources is also revealed, thus saving most of IP traceback efforts that might otherwise be needed.

CHAPTER IV

HOP-COUNT FILTERING: AN EFFECTIVE DEFENSE AGAINST SPOOFED DDOS TRAFFIC

4.1 Introduction

An Internet host can spoof IP packets by using a raw socket to fill arbitrary source IP addresses into their IP headers [76]. IP spoofing is usually associated with malicious network behaviors, such as Distributed Denial of Service (DDoS) attacks. To conceal flooding sources and localities in flooding traffic, attackers often spoof IP addresses by randomizing the 32-bit source-address field in the IP header [32, 33]. Moreover, some known DDoS attacks, such as smurf [24] and more recent DRDoS (Distributed Reflection Denial of Service) attacks [45, 86], are not possible without IP spoofing. Such attacks masquerade the source IP address of each spoofed packet with the victim's IP address. It is difficult to counter IP spoofing because of the stateless and destination-based routing of the Internet. The IP protocol lacks the control to prevent a sender from hiding the origin of its packets. Furthermore, destination-based routing does not maintain state information on senders, and forwards each IP packet toward its destination without validating the packet's true origin. Overall, IP spoofing makes DDoS attacks much more difficult to defend against.

As the observed prevalence of DDoS attacks in the Internet [75], many router-based defense mechanisms have been proposed, including router filtering [40, 84], router throttling [120], Pushback [66, 55], traceback [14, 95, 99, 100, 105] and detection mechanisms [46, 116]. However, these solutions require not only router support, but also coordination among different routers and networks, and wide-spread deployment. While previous victim-based defense mechanisms [11, 17, 90, 101] can confine the scope of damage to the service under attack, it may not be able to sustain the availability of the service being attacked. In stark contrast, the server's ability to filter most, if not all, spoofed IP packets can help sustain service availability even under DDoS attacks.

Therefore, an immediately deployable victim-based filtering, which detects and discards spoofed traffic without any router support, is essential to protecting victims against DDoS attacks. We only utilize the information contained in the IP header for packet filtering. Although an attacker can forge any field in the IP header, he or she cannot falsify the number of hops an IP packet takes to reach its destination, which is solely determined by the Internet routing infrastructure. The hop-count information is indirectly reflected in the TTL field of the IP header, since each intermediate router decrements the TTL value by one before forwarding a packet to the next hop. The difference between the initial TTL (at the source) and the final TTL value (at the destination) is the hop-count between the source and the destination. By examining the TTL field of each arriving packet, the destination can infer its initial TTL value, and hence the hop-count from the source. Here we assume that attackers cannot sabotage routers to alter TTL values of IP packets that traverse them.

In this chapter, we propose a novel hop-count-based filter to weed out spoofed IP packets. The rationale behind hop-count filtering is that most spoofed IP packets, when arriving at victims, do not carry hop-count values that are consistent with the IP addresses being spoofed. *Hop-Count Filtering* (HCF) builds an accurate IP-to-hop-count (IP2HC) mapping table, while using a moderate amount of storage, by clustering address prefixes based on hop-count. To capture hop-count changes under dynamic network conditions, we also devise a safe update procedure for the IP2HC mapping table that prevents pollution by HCF-aware attackers. The same pollution-proof method is used for IP2HC mapping table initialization and adding new IP addresses into the table.

Two running states, *alert* and *action*, within HCF use this mapping to inspect the IP header of each IP packet. Under normal condition, HCF stays in *alert* state, watching for abnormal TTL behaviors without discarding any packet. Even if a legitimate packet is incorrectly identified as a spoofed one, it will not be dropped. Therefore, there is no collateral damage in *alert* state. Upon detection of an attack, HCF switches to *action* state, in which HCF discards those IP packets with mismatching hop-counts. Besides the IP2HC inspection, several efficient mechanisms [43, 46, 79, 116] are available to detect DDoS attacks. Through analysis using network measurement data, we show that HCF can recognize close to 90% of spoofed IP packets. In addition, our hop-count-based clustering significantly reduces the percentage of false positives.[1] Thus, we can discard spoofed IP packets with little collateral damage in *action state*. To ensure that the filtering mechanism itself withstands attacks, our design is light-weight and requires only a moderate amount of storage. We implement HCF in the Linux kernel at the IP layer as the first step of incoming packet processing. We evaluate the benefit of HCF with experimental measurements and show that HCF is indeed effective in countering IP spoofing by providing significant resource savings.

Researchers have used the distribution of TTL values seen at servers to detect abnormal load spikes due to DDoS traffic [89]. The Razor team at Bindview built Despoof [8], which is a command-line anti-spoofing utility. Despoof compares the TTL of a received packet that is considered "suspicious," with the actual TTL of a test packet sent to the source IP

[1] Percentage of the legitimate packets identified as the spoofed.

address, for verification. However, Despoof requires the administrator to determine which packets should be examined, and to manually perform this verification. Thus, the per-packet processing overhead is prohibitively high for weeding out spoofed traffic in real time.

In parallel with, and independent of our work, the possibility of using TTL for detecting spoofed packet was discussed in [107]. Their results have shown that the final TTL values from an IP address were predictable and generally clustered around a single value, which is consistent with our observation of hop-counts being mostly stable. However, the authors did not provide a detailed solution against spoofed DDoS attacks. Neither did they provide any analysis of the effectiveness of using TTL values, nor the construction, update, and deployment of an accurate TTL mapping table. In this chapter, we examine both questions and develop a deployable solution.

There already exist commercial solutions [54, 78] that block the propagation of DDoS traffic with router support. However, the main difference between our scheme and the existing approaches is that HCF is an end-system mechanism that does not require **any** network support. This difference implies that our solution is immediately deployable in the Internet. The remainder of the chapter is organized as follows. Section 4.2 presents the TTL-based hop-count computation and the hop-count inspection algorithm, which is in the critical path of HCF. Section 4.3 studies the feasibility of the proposed filtering mechanism, based on a large set of previously-collected `traceroute` data, and the resilience of our filtering scheme against HCF-aware attackers. Section 4.4 demonstrates the effectiveness of the proposed filter in detecting spoofed packets. Section 4.5 deals with the construction of IP2HC mapping table, the heart of HCF. Section 4.6 details the two running states of HCF, the inter-state transitions, and the placement of HCF. Section 4.7 presents the implementation and experimental evaluation of HCF. Section 4.8 discusses the effect of newly-arrived requests and hop-count changes upon filtering accuracy. The chapter concludes with Section 4.9.

4.2 Hop-Count Inspection

Central to HCF is the validation of the source IP address of each packet via hop-count inspection. In this section, we first discuss the hop-count computation, and then detail the inspection algorithm.

4.2.1 TTL-based Hop-Count Computation

Since hop-count information is not directly stored in the IP header, one has to compute it based on the TTL field. TTL is an 8-bit field in the IP header, originally introduced to specify the maximum lifetime of each packet in the Internet. Each intermediate router decrements the TTL value of an in-transit IP packet by one before forwarding it to the next-hop. The final TTL value when a packet reaches its destination is therefore the initial TTL subtracted by the number of intermediate hops (or simply hop-count). The challenge in hop-count computation is that a destination only sees the final TTL value. It would have been simple had all operating systems (OSs) used the same initial TTL value, but in practice, there is no consensus on the initial TTL value. Furthermore, since the OS for a given IP address may change with time, we cannot assume a single static initial TTL value for each IP address.

Fortunately, however, according to [34], most modern OSs use only a few selected initial TTL values, 30, 32, 60, 64, 128, and 255. This set of initial TTL values cover most of the popular OSs, such as Microsoft Windows, Linux, variants of BSD, and many commercial Unix systems. We observe that most of these initial TTL values are far apart, except between 30 and 32, 60 and 64, and between 32 and 60. Since Internet traces have shown that few Internet hosts are apart by more than 30 hops [26, 27], which is also confirmed by our own observation, one can determine the initial TTL value of a packet by selecting the smallest initial value in the set that is larger than its final TTL. For example, if the final TTL value is 112, the initial TTL value is 128, the smaller of the two possible initial values, 128 and 255. To resolve ambiguities in the cases of $\{30, 32\}$, $\{60, 64\}$, and $\{32, 60\}$, we will compute a hop-count value for each of the possible initial TTL values, and accept the packet if there is a match with one of the possible hop-counts.

The drawback of limiting the possible initial TTL values is that packets from end-systems that use “odd” initial TTL values, may be incorrectly identified as having spoofed source IP addresses. This may happen if a user switches OS from one that uses a “normal” initial TTL value to another that uses an “odd” value. Since our filter starts to discard packets only upon detection of a DDoS attack, such end-systems would suffer only during an actual DDoS attack. The study in [34] shows that the OSs that use “odd” initial TTLs are typically older OSs. We expect such OSs to constitute a very small percentage of end-hosts in the current Internet. Thus, the benefit of deploying HCF should out-weight the risk of denying service to those end-hosts during attacks.

4.2.2 Inspection Algorithm

Assuming that an accurate IP2HC mapping table is present (see Section 4.5 for details of its construction) Figure 4.2.1 outlines the HCF procedure used to identify spoofed packets. The inspection algorithm extracts the source IP address and the final TTL value from each IP packet. The algorithm infers the initial TTL value and subtracts the final TTL value from it to obtain the hop-count. The source IP address serves as the index into the table to retrieve the correct hop-count for this IP address. If the computed hop-count matches the stored hop-count, the packet has been “authenticated;” otherwise, the packet is likely spoofed. We note that a spoofed IP address may happen to have the same hop-for each packet:

extract the final TTL T_f and the IP address S;

infer the initial TTL T_i;

compute the hop-count $H_c = T_i - T_f$;

index S to get the stored hop-count H_s;

if $(H_c \neq H_s)$

the packet is spoofed;

else

the packet is legitimate;

Figure 4.1: Hop-Count inspection algorithm.

count as the one from a zombie (flooding source[2]) to the victim. In this case, HCF will not be able to identify the spoofed packet. However, we will show in Section 4.4 that even with a limited range of hop-count values, HCF is highly effective in identifying spoofed IP addresses.

Occasionally, legitimate packets may be identified as spoofed due to inaccurate IP2HC mapping or delay in hop-count update. Therefore, it is important to minimize collateral damage under HCF. We note that an identified spoofed IP packet is only dropped in the *action* state, while HCF only keeps track of the number of mis-matched IP packets without discarding any packets in the *alert* state. This guarantees *no* collateral damage in the *alert* state, which should be much more common than the *action* state.

[2]In this chapter, the terms zombie and flooding source are used interchangeably.

4.3 Feasibility of Hop-Count Filtering

The feasibility of HCF hinges on three factors: (1) stability of hop-counts, (2) diversity of hop-count distribution, and (3) robustness against possible evasions. In this section, we first examine the stability of hop-counts. Then, we assess if valid hop-counts to a server are diverse enough, so that matching the hop-count with the source IP address of each packet suffices to recognize spoofed packets with high probability. Finally, our discussion will show that it is difficult for an HCF-aware attacker to circumvent filtering.

4.3.1 Hop-Count Stability

The stability in hop-counts between an Internet server and its clients is crucial for HCF to work correctly and effectively. Frequent changes in the hop-count between the server and each of its clients not only lead to excessive mapping updates, but also greatly reduce filtering accuracy when an out-of-date mapping is in use during attacks.

The hop-count stability is dictated by the end-to-end routing behaviors in the Internet. According to the study of end-to-end routing stability in [85], the Internet paths were found to be dominated by a few prevalent routes, and about two thirds of the Internet paths studied were observed to have routes persisting for either days or weeks. To confirm these findings, we use daily `traceroute` measurements taken at ten-minute intervals among 113 sites [42] from January 1st to April 30th, 2003. We observed a total of 10,814 distinct one-way paths, a majority of which had 12,000 traceroute measurements each over the five-month period. In these measurements, most of the paths experienced very few hop-count changes: 95% of the paths had fewer than five observable daily changes. Therefore, it is reasonable to expect hop-counts to be stable in the Internet. Moreover, the proposed filter contains a dynamic update procedure to capture hop-count changes as discussed in Section 4.5.2.

4.3.2 Diversity of Hop-Count Distribution

Because HCF cannot recognize forged packets whose source IP addresses have the same hop-count value as that of a zombie, a diverse hop-count distribution is critical to effective filtering. It is necessary to examine hop-count distributions at various locations in the Internet to ensure that hop-counts are not concentrated around a single value. If 90% of client IP addresses are ten hops away from a server, one would not be able to distinguish many spoofed packets from legitimate ones using HCF alone.

Type	Sample Number
Commercial sites	11
Educational sites	4
Non-profit sites	2
Foreign sites	18
.net sites	12

Table 4.1: Diversity of `traceroute` gateway locations.

To obtain actual hop-count distributions, we use the raw `traceroute` data from 50 different `traceroute` gateways in [29]. We use only 47 of the data sets because three of them contain too few clients compared to the others. The locations of `traceroute` gateways are diverse as shown in Table 4.1. Figure 4.2 shows the distribution of the number of clients measured by each of the 47 `traceroute` gateways. Most of the `traceroute` gateways measured hop-counts to more than 40,000 clients.

We examined the hop-count distributions at all `traceroute` gateways to find that the Gaussian distribution (bell-shaped curve) is a good first-order approximation. Figures 4.3 and 4.4 show the hop-count distributions of four selected sites: a well-connected commercial server `net.yahoo.com`, an educational institute Stanford University, a non-profit

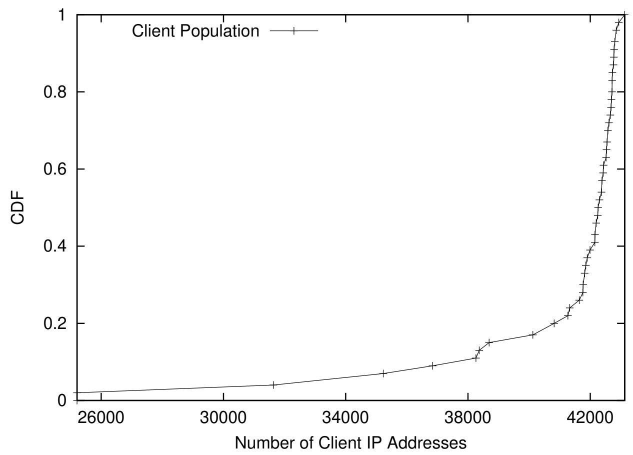


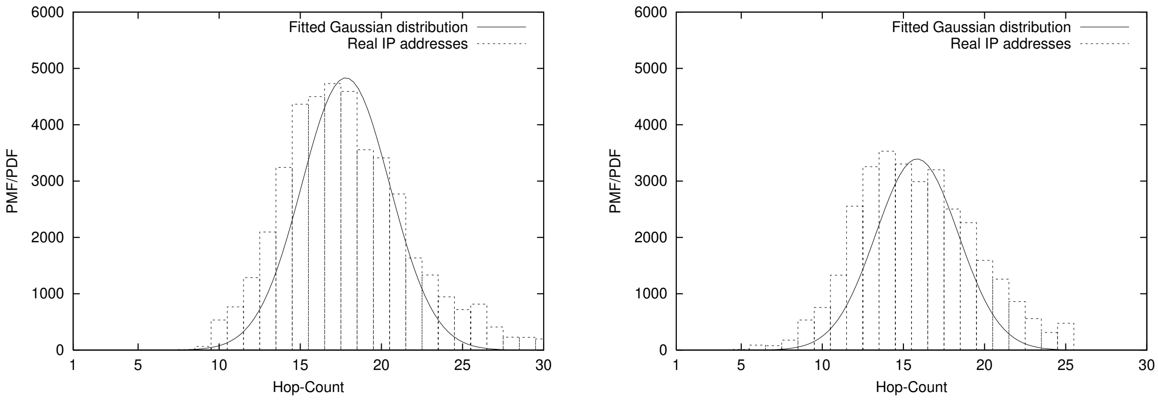
Figure 4.2: CDF of size of client IP addresses.

organization cpcug.org, and one site outside of the United States, fenice.it. We are interested in the girth of a distribution, which can give a qualitative indication of how well HCF works, i.e., the wider the girth, the more effective HCF will be. For Gaussian distributions, the girth is the standard deviation, σ. The Gaussian distribution[3] can be written in the following form:

$$f(h) = C\, e^{-\frac{(h-\mu)^2}{2\sigma^2}}$$

where C is the normalization constant, so the area under the Gaussian distribution sums to the number of IP addresses measured. The mean value of a Gaussian distribution specifies the center of the bell-shaped curve, and the standard deviation specifies the girth of the bell. We are only interested in using the Gaussian distribution to study if hop-count is a suitable measure for HCF. We are not making any definitive claim of whether hop-count distributions are Gaussian or not. For each given hop-count distribution, we use the normfit function in Matlab to fit the distribution of hop-counts for each data set. We plot the means and standard deviations, along with their 95% confidence intervals, in Figure **??**. We observe that most of the mean values fall between 14 and 19 hops, and the

[3]By "distribution," we mean it in a generic sense that is equivalent to histogram.



(a) Yahoo (b) Stanford University

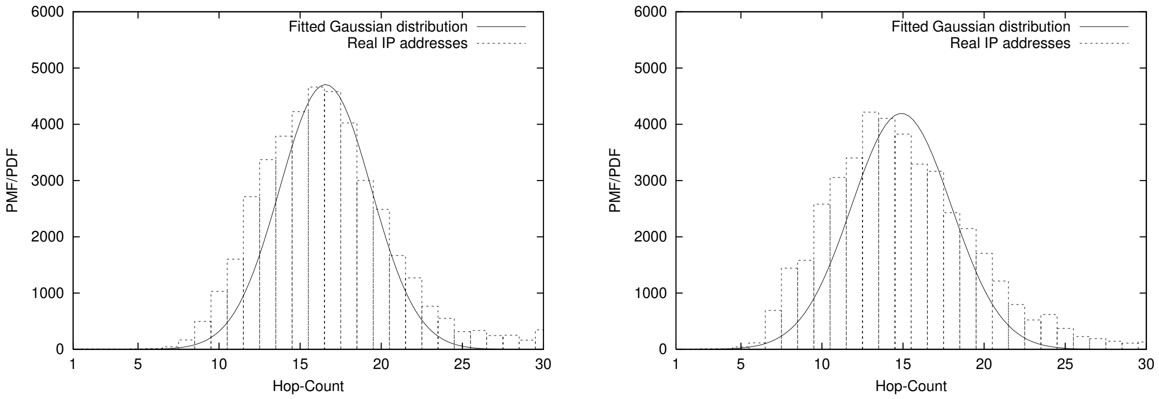
Figure 4.3: Hop-count distribution (part I)

standard deviations between 3 and 5 hops. The largest percentage of IP addresses that have a common hop-count value is only 10%. Such distributions allow HCF to work effectively as we will show in the quantitative evaluation of HCF in Section 4.4.

4.3.3 Robustness against Evasion

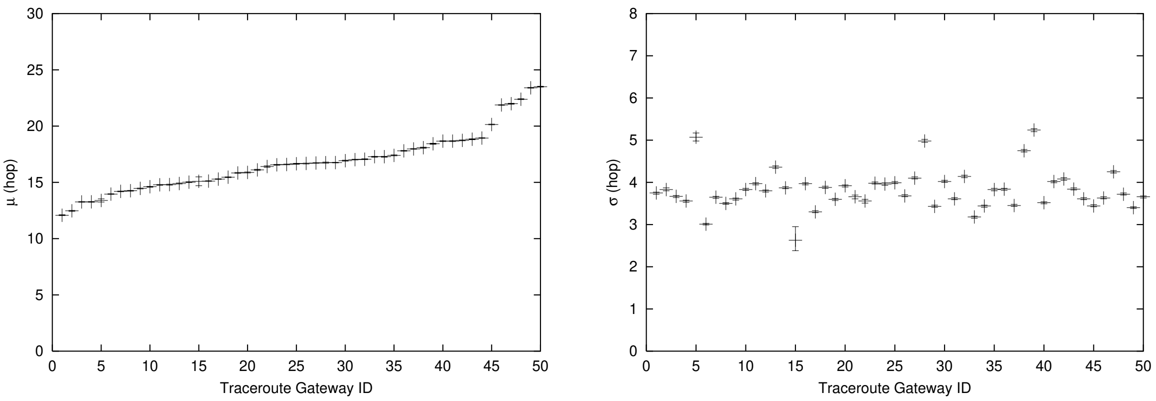
Once attackers learn of HCF, they will try to generate spoofed packets that can dodge hop-count inspections, hence evading HCF. However, such an attempt will either require a large amount of resource or time, and very elaborate planning, i.e., casual attackers are unlikely to be able to evade HCF. In what follows, we assess the various ways attackers may evade HCF.

The key for an attacker to evade HCF is his ability to set an appropriate initial TTL value for each spoofed packet, because the number of hops traversed by an IP packet is determined solely by the routing infrastructure. Assuming the same initial TTL value I for all Internet hosts, a packet from a flooding source, which is h_z hops away from the victim, has a final TTL value of $I - h_z$. In order for the attacker to generate spoofed packets from this flooding source without being detected, the attacker must change the initial TTL



(a) `cpcug.org` (b) `fenice.it` in Italy

Figure 4.4: Hop-count distribution (part II)



(a) μ (b) σ

Figure 4.5: Means and standard deviations for traceroute gateways

value of each packet to $I' = I - (h_s - h_z)$, where h_s is the hop-count from the spoofed IP address to the victim. Each spoofed packet would have the correct final TTL value, $I - (h_s - h_z) - h_z = I - h_s$, when it reaches the victim.

An attacker can easily learn the hop-count, h_z, from a zombie site to the victim by running `traceroute`. However, randomly selecting the source address for each spoofed IP packet [32, 33] makes it extremely difficult, if not impossible, for the attacker to learn h_s. To obtain the correct h_s values for all spoofed packets sent to the victim, the attacker has to build *a priori* an IP2HC mapping table that covers the entire random IP address space. This is much more difficult than building an IP2HC mapping table at the victim, since the attacker cannot observe the final TTL values of normal traffic at the victim. For an attacker to build such an IP2HC mapping table, he or she may have to compromise at least one end-host behind every stub network whose IP addresses are in the random IP address space, and perform `traceroute` to get h_s for the corresponding IP2HC mapping entry. Without correct h_s values, an attacker cannot fabricate the appropriate initial TTL values to conceal forgery.

Without compromising end-hosts, it may be possible for an attacker to probe the h_s value for a given IP address if it is not sending any packets to the network. The probing procedure works as follows: (1) force the victim into the *action* state to actively filter packets by launching a DDoS attack; (2) probe the quiescent host and extract the latest value of its IP identification field of the header [118]; (3) send a spoofed packet containing a legitimate request with the quiescent IP address as the source IP address to the victim with a tentative initial TTL; (4) re-probe the quiescent host and check if its IP ID has increased by more than one. If it has, this indicates that the victim has accepted the spoofed packet and the initial TTL is the desired one.[4] Otherwise, the attacker will change the

[4]If the victim accepts the spoofed packet, a response would be sent to the quiescent host, causing it to generate a response, most likely a RST, and increase the IP ID number by one.

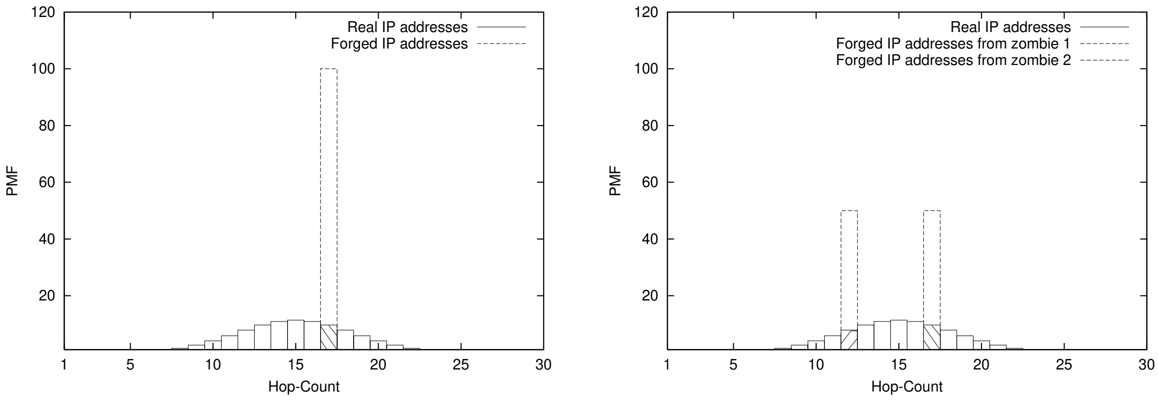
initial TTL value and repeat the above probing procedure. Although it is possible to obtain the appropriate initial TTL for a single IP address, probing the whole random address space requires an excessive amount of time and effort. First, an attacker has to launch a DDoS attack that must last long enough to accommodate a large number of probes, or launch numerous short-lived DDoS attacks to accommodate all probing activities. Even if the attacker probes only one host per stub network, with the Internet containing tens of millions of stub networks, it is difficult to hide during this process of TTL probing. Second, the attacker must ensure an IP address remains quiescent during the probing. Since the attacker cannot prevent the probed IP address from becoming active, he or she can easily misinterpret an increase of the IP ID number as the forged initial TTL being correct.

Without compromising end-hosts, an attacker may compute hop-counts of to-be-spoofed IP addresses based on an accurate router-level topology of the Internet, and the underlying routing algorithms and policies. The recent Internet mapping efforts such as Internet Map [26], Mercator [48], Rocketfuel [102], and Skitter [27] projects, may make the approach plausible. However, the current topology mappings put together snapshots of various networks measured at different times. Thus-produced topology maps are generally time-averaged approximations of actual network connectivity. More importantly, interdomain routing in the Internet is policy-based, and the routing policies are not disclosed to the public. The path, and therefore the hop-count, between a source and a destination is determined by routing policies and algorithms that are often unknown. Even if an attacker has accurate information of the Internet topology, he or she cannot obtain the correct hop-counts based on network connectivity alone. We believe that the quality of network maps will improve with better mapping technology, but we do not anticipate any near-term advances that can lead to accurate hop-counts based on just Internet maps.

Instead of spoofing randomly-selected IP addresses, an attacker may choose to spoof IP addresses from a set of already-compromised machines that are much smaller in number than 2^{32}, so that he or she can measure all h_s's and fabricate appropriate initial TTLs. However, this weakens the attacker's ability in several ways. First, the list of would-be spoofed source IP addresses is greatly reduced, which makes the detection and removal of flooding traffic much easier. Second, source addresses of spoofed IP packets reveal the locations of compromised end-hosts, which makes IP traceback much easier. Third, the attacker must somehow probe the victim server to obtain the correct hop-counts. However, network administrators nowadays are extremely alert to unusual access patterns or probing attempts; so, it would require a great deal of effort to coordinate the probing attempts so as not to raise red flags. Fourth, the attacker must modify the available attacking tools since the most popular distributed attacking tools, including mstream, Shaft, Stacheldraht, TFN, TFN2k, Trinoo and Trinity, generate randomized IP addresses in the space of 2^{32} for spoofing [32, 33]. The wide-spread use of randomness in spoofing IP address has been verified by the "backscatter" study [75], which quantified DoS activities in the Internet.

4.4 Effectiveness of HCF

We now assess the effectiveness of HCF from a statistical standpoint. More specifically, we address the question "what fraction of spoofed IP packets can be detected by the proposed HCF?" We assume that potential DDoS victims know the complete mapping between their client IP addresses and hop-counts (to the victims themselves). In the next section, we will discuss the construction of such mappings. We assume that, to spoof packets, the attacker randomly selects source IP addresses from the entire IP address space, and chooses hop-counts according to some distribution. Without loss of generality, we further assume that the attacker evenly divides the flooding traffic among the flooding sources.



(a) Single flooding source (b) Two flooding sources

Figure 4.6: Hop-Count distribution of IP addresses

This analysis can be easily extended for cases where the flooding traffic is unevenly distributed. To make the analysis tractable, we consider only static hop-counts. We will later discuss an update procedure that will capture legitimate hop-count changes.

4.4.1 Simple Attacks

First, we examine the effectiveness of HCF against simple attackers that spoof source IP addresses while still using the default initial TTL values at the flooding sources. Most of the available DDoS attacking tools [32, 33] do not alter the initial TTL values of packets. Thus, the final TTL value of a spoofed packet will bear the hop-count value between the flooding source and the victim. To assess the performance of HCF against such simple attacks, we consider two scenarios: single flooding source and multiple flooding sources.

A Single Source

Given a single flooding source whose hop-count to the victim is h, let α_h denote the fraction of IP addresses that have the same hop-count to the victim as the flooding source. Figure 4.6 (a) depicts the hop-count distributions seen at a hypothetical server for both real client IP addresses, and spoofed IP addresses generated by a single flooding source. Since spoofed IP addresses come from a single source, they all have an identical hop-count. Hence, the hop-count distribution of spoofed packets is a vertical bar of width one. On the other hand, real client IP addresses have a diverse hop-count distribution that is observed to be close to a Gaussian distribution. The shaded area represents those IP addresses — the fraction α_h of total valid IP addresses — that have the same distance to the server as the flooding source. Thus, the fraction of spoofed IP addresses that cannot be detected is α_h. The remaining fraction $1 - \alpha_h$ will be identified and discarded by HCF.

The attacker may happen to choose a zombie that is 16 or 17 — the most popular hop-count values — hops away from the victim as the flooding source. However, the standard deviations of the fitted Gaussian distributions are still reasonably large such that the percentage of IP addresses with any single hop-count value is small relative to the overall IP address space. As shown in Section 4.3.2, even if the attacker floods spoofed IP packets from such a zombie, HCF can still identify nearly 90% of spoofed IP addresses. In most distributions, the mode accounts for 10% of the total IP addresses, with the maximum and minimum of the 47 modes being 15% and 8%, respectively. Overall, HCF is very effective against these simple attacks, reducing the attack traffic by one order of magnitude.

Multiple Sources

DoS attacks usually involve more than a single host, and hence, we need to examine the case of multiple active flooding sources. Assume that there are n sources that flood a total of F packets, each flooding source generates F/n spoofed packets. Figure 4.6 (b) shows the hop-count distribution of spoofed packets sent from two flooding sources. Each flooding source is seen to generate traffic with a single unique hop-count value. Let h_i be the hop-count between the victim and flooding source i, then the spoofed packets from source i that HCF can identify is $\frac{F}{n}(1 - \alpha_i)$. The fraction, Z, of identifiable spoofed packets generated by n flooding sources is:

$$\begin{aligned} Z &= \frac{\frac{F}{n}(1-\alpha_{h_1}) + \cdots + \frac{F}{n}(1-\alpha_{h_n})}{F} \\ &= 1 - \frac{1}{n}\sum_{i=1}^{n} \alpha_{h_i} \end{aligned}$$

This expression says that the overall effectiveness of having multiple flooding sources is somewhere between that of the most effective source i with the largest α_{h_i} and that of the least effective source j with the smallest α_{h_j}. Adding more flooding sources does not weaken the HCF's ability to identify spoofed IP packets. On the contrary, since the hop-count distribution follows Gaussian, existence of less effective flooding sources (with small α_h's) enables the filter to identify and discard more spoofed IP packets than in the case of a single flooding source.

4.4.2 Sophisticated Attackers

Most attackers will eventually recognize that it is not enough to merely spoof source IP addresses. Instead of using the default initial TTL value, the attacker can easily randomize it for each spoofed IP packet. Although the hop-count from a single flooding source to the victim is fixed, randomizing the initial TTL values will create an illusion of packets having many different hop-count values at the victim server. The range of randomized initial TTL values should be a subset of $[h_z,\ I_d + h_z]$, where h_z is the hop-count from the flooding source to the victim and I_d is the default initial TTL value. The starting point in this range should not be less than h_z. Otherwise, spoofed IP packets bearing TTLs smaller than h_z will be discarded before they reach the victim. The simplest method of generating initial TTLs at a single source is to use a uniform distribution. The final TTL values, T_v's, seen at the victim are $I_r - h_z$, where I_r represents randomly-generated initial TTLs. Since h_z is constant and I_r follows a uniform distribution, T_v's are also uniformly-distributed.

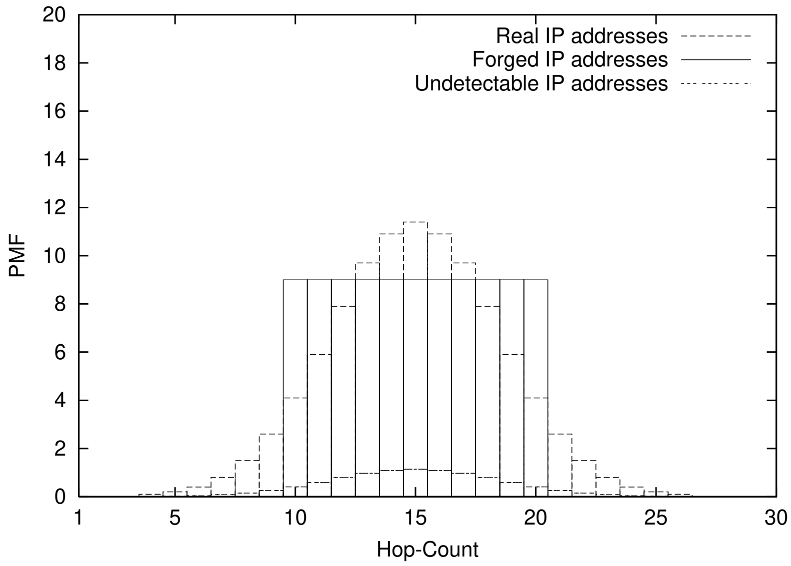


Figure 4.7: Hop-Count distribution of IP addresses with a single flooding source, randomized TTL values

Since the victim derives the hop-count of a received IP packet based on its T_v value, the perceived hop-count distribution of the spoofed source IP address is uniformly-distributed.

Figure 4.7 illustrates the effect of randomized TTLs, where $h_z = 10$. We use a Gaussian curve with $\mu = 15$ and $\sigma = 3$ to represent a typical hop-count distribution (see Section 4.3.2) from real IP addresses to the victim, and the box graph to represent the perceived hop-count distribution of spoofed IP addresses at the victim. The large overlap between the two graphs may appear to indicate that our filtering mechanism is not effective. On the contrary, uniformly-distributed random TTLs actually conceal fewer spoofed IP addresses from HCF. For uniformly-distributed TTLs, each spoofed source IP address has the probability $1/H$ of having the matching TTL value, where H is the number of possible hop-counts. Consequently, for each possible hop-count h, only α_h/H fraction of IP addresses have correct TTL values. Overall, assuming that the range of possible hop-counts is $[h_i,\ h_j]$ where $i \leq j$ and $H = j - i + 1$, the fraction of spoofed source IP addresses that have correct TTL values, is given as:

$$\bar{Z} = \frac{\alpha_{h_i}}{H} + \ldots + \frac{\alpha_{h_j}}{H} = \frac{1}{H} \cdot \sum_{k=i}^{j} \alpha_{h_k}.$$

Note that we use $\bar{Z}$ in place of $1 - Z$ to simplify notation. In Figure 4.7, the range of generated hop-counts is between 10 and 20, so $H = 11$. The summation will have a maximum value of 1 so $\bar{Z}$ can be at most $1/H = 8.5\%$, which is represented by the area under the shorter Gaussian distribution in Figure 4.7. In this case, less than 10% of spoofed packets go undetected by HCF.

In general, an attacker could generate initial TTLs within the range $[h_m, h_n]$, based on some known distribution, where the probability of IP addresses with hop-count h_k is p_{h_k}. If in the actual hop-count distribution at the victim server, the fraction of the IP addresses that have a hop-count of h_k is α_{h_k}, then the fraction of the spoofed IP packets that will not be caught by HCF is:

$$\bar{Z} = \sum_{k=m}^{n} \alpha_{h_k} \cdot p_{h_k}.$$

The term inside the summation simply states that only p_{h_k} fraction of IP addresses with hop-count h_k can be spoofed with matching TTL values. For instance, if an attacker is able to generate initial TTLs based on the hop-count distribution at the victim, p_{h_k} becomes α_{h_k}. In this case, $\bar{Z}$ becomes $\bar{Z} = \sum_{k=m}^{n} \alpha_{h_k}^2$. Based on the hop-count distribution in Figure 4.7, we can again calculate $\bar{Z}$ for $m = 0$ and $n = 30$ to be 9.4%, making this attack slightly more effective than randomly-generating TTLs. Surprisingly, none of these "intelligent" attacks are much more effective than the simple attacks in Section 4.4.1.

4.5 Construction of HCF Table

We have shown that HCF can remove nearly 90% of spoofed traffic with an accurate mapping between IP addresses and hop-counts. Thus, building an accurate HCF table (i.e., IP2HC mapping table) is critical to detecting the maximum number of spoofed IP packets. In this section, we detail our approach to constructing an HCF table. Our objectives in building an HCF table are: (1) accurate IP2HC mapping, (2) up-to-date IP2HC mapping, and (3) moderate storage requirement. By clustering address prefixes based on hop-counts, we can build accurate IP2HC mapping tables and maximize HCF's effectiveness without storing the hop-count for each IP address. Moreover, we design a pollution-proof update procedure that captures legitimate hop-count changes while foiling attackers' attempt to pollute HCF tables.

4.5.1 IP Address Aggregation

It is highly unlikely that an Internet server will receive legitimate requests from all live IP addresses in the Internet. Also, the entire IP address space is not fully utilized in the current Internet. By aggregating IP address, we can reduce the space requirement of IP2HC mapping significantly. More importantly, IP address aggregation covers those unseen IP addresses that are co-located with those IP addresses that are already in an HCF table.

Grouping hosts according to the first 24 bits of IP addresses is a common aggregation method. However, hosts whose network prefixes are longer than 24 bits, may reside in different physical networks in spite of having the same first 24 bits. Thus, these hosts are not necessarily co-located and have identical hop-counts. To obtain an accurate IP2HC mapping, we must refine the 24-bit aggregation. Instead of simply aggregating into 24-bit address prefixes, we further divide IP addresses within each 24-bit prefix into smaller clusters based on hop-counts. To understand whether this refined clustering improves HCF over the simple 24-bit aggregation, we compare the filtering accuracies of HCF tables under both aggregations — the simple 24-bit aggregation (without hop-count clustering) and the 24-bit aggregation with hop-count clustering.

For this accuracy experiment, we treat each traceroute gateway (Section 4.3.2) as a "web server," and its measured IP addresses as clients to this web server. We build an HCF table based on the set of client IP addresses at each web server and evaluate the filtering accuracy under each aggregation method. We assume that the attacker knows the client IP addresses of each web server and generates packets by randomly selecting source IP addresses among legitimate client IP addresses. We further assume that the attacker knows the general hop-count distribution and uses it to generate the hop-count for each spoofed packet. This is the DDoS attack that the most knowledgeable attacker can launch without learning the exact IP2HC mapping, i.e., the best scenario for the attacker.

We define the filtering accuracy of an HCF table to be the percentages of false positives and false negatives. False positives are those legitimate client IP addresses that are incorrectly identified as spoofed. False negatives are spoofed packets that go undetected by HCF. Both should be minimized in order to achieve maximum filtering accuracy. We compute the percentage of false positives as the number of client IP addresses identified as spoofed divided by the total number of client IP addresses. We compute the percentage of false negatives according to the calculation in Section 4.4.2.

Aggregation into 24-bit Address Prefixes

For each web server, we build an HCF table by grouping its IP addresses according to the first 24 bits. We use the minimum hop-count of all IP addresses inside a 24-bit network address as the hop-count of the network. After the table is constructed, each IP address is converted into a 24-bit address prefix, and the actual hop-count of the IP address is compared to the one stored in the aggregate HCF table. Since 24-bit aggregation does not preserve the correct hop-counts for all IP addresses, we examine the performance of three

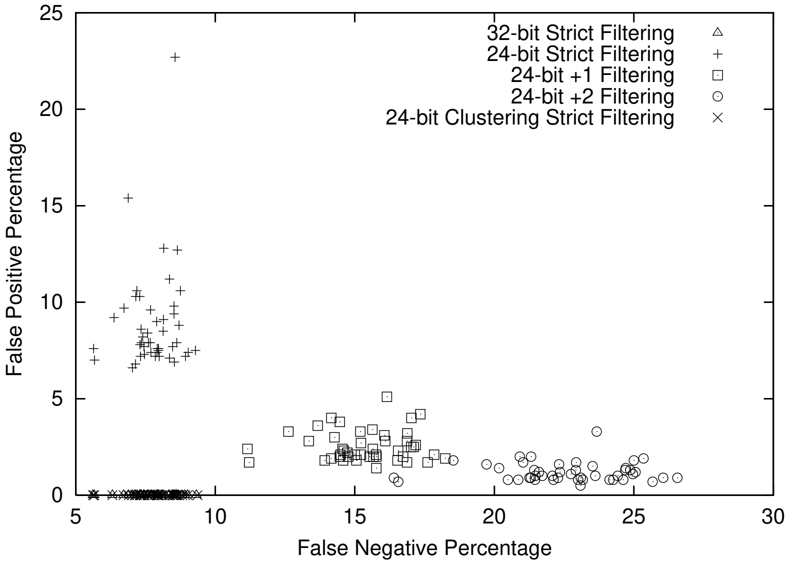


Figure 4.8: Accuracies of various filters. (Note that the points of 24-bit clustering filtering overlap with those of 32-bit filtering.)

types of filters: "Strict Filtering," "+1 Filtering," and "+2 Filtering." "Strict Filtering" drops packets whose hop-counts do not match those stored in the table. "+1 Filtering" drops packets whose hop-counts differ by greater than 1 compared to those in the table, and "+2 Filtering" drops packets whose hop-counts differ by greater than two.

We have shown in Section 4.4.2 that percentage of false negatives is determined by the distribution of hop-counts. Aggregation of IP addresses into 24-bit network addresses does not change the hop-count distribution significantly. Thus, the 24-bit strict filtering yields a similar percentage of false negatives for each web server to the case of storing individual IP addresses (32-bit Strict Filtering in the figure). On the other hand, percentage of false positives is significantly higher in the case of aggregation as expected. Figure 4.8 presents the combined false positive and false negative results for the three filtering schemes. The x-axis is the percentage of false negatives, and the y-axis is the percentage of false positives. Each point in the figure represents the pair of percentages for a single web server.

For example, under "24-bit Strict Filtering," most web servers suffer about 10% of false positives, while only 5% of false negatives. As we relax the filtering criterion, false positives are halved while false negatives approximately doubled. Clearly, tolerating packets with mismatching hop-counts requires to make a trade-off between percentage of false positives and that of false negatives. Overall, +1 Filtering offers a reasonable compromise between false negatives and false positives. Considering the impact of DDoS attacks without HCF, a small percentage of false positives may be an acceptable price to pay.

In practice, 24-bit aggregation is straightforward to implement and can offer fast lookup with an efficient implementation. Assuming a one-byte entry per network prefix for hop-count, the storage requirement is 2^{24} bytes or 16 MB. The memory requirement is modest compared to contemporary servers which are typically equipped with multi-gigabytes of memory. Under this setup, the lookup operation consists of computing a 24-bit address prefix from the source IP address in each packet and indexing it into the HCF table to find the right hop-count value. For systems with limited memory, the aggregation table can be implemented as a much smaller hash-table. While 24-bit aggregation may not be the most accurate, at present it is a good and deployable solution.

Aggregation with Hop-Count Clustering

Under 24-bit aggregation, the percentage of false negatives is still high ($\approx$15%) if false positives are to be kept reasonably small. Based on hop-count, one can further divide IP addresses within each 24-bit prefix into smaller clusters. By building a binary aggregation tree iteratively from individual IP addresses, we cluster IP addresses with same hop-count together. The leaves of the tree represent the 256 (254 to be precise) possible IP addresses inside a 24-bit address prefix. In each iteration, we examine two sibling nodes and determine whether we can aggregate IP addresses behind these two nodes. We will aggregate

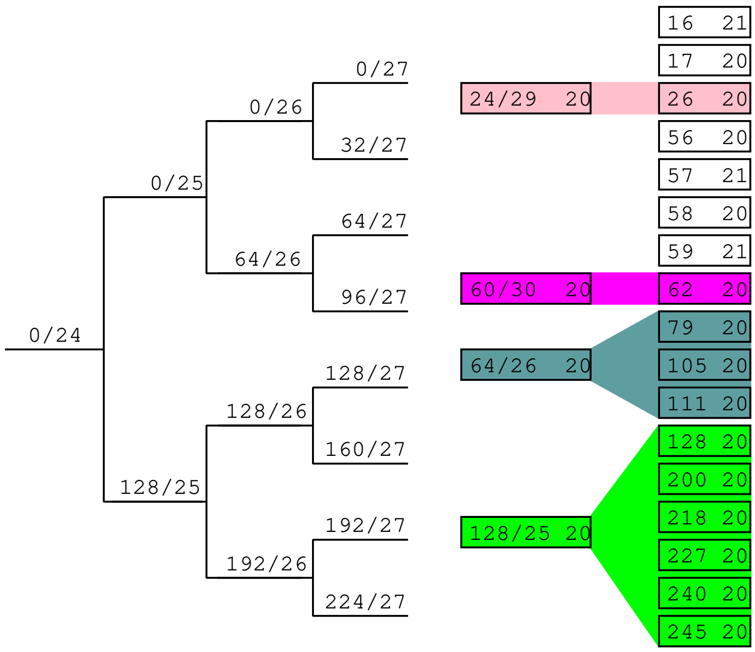


Figure 4.9: An example of hop-count clustering.

the two nodes as long as they share a common hop-count, or one of them is empty. If aggregate is possible, the parent node will have the same hop-count as the children. We can thus find the largest possible aggregation for a given set of IP addresses. Figure 4.9 shows an example of clustering a set of IP addresses (with the last octets shown) by their hop-counts using the aggregation tree (showing the first four levels). For example, the IP address range, 128 to 245, is aggregated into a 128/25 prefix with a hop-count of 20, and the three IP addresses, 79, 105, and 111 are aggregated into a 64/26 prefix with a hop-count of 20. However, we cannot aggregate these two blocks further up the tree due to holes in the address space. We are able to aggregate 11 of 17 IP addresses into four network prefixes. The remaining IP addresses must be stored as individual IP addresses.

With hop-count-based clustering, we never aggregate IP addresses that do not share the same hop-count. Hence, we can eliminate false positives when all clients of a server

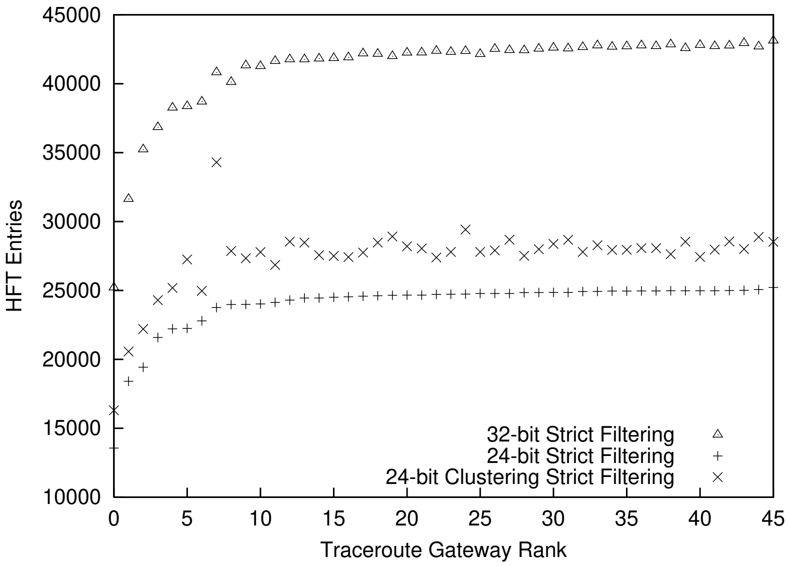


Figure 4.10: Sizes of various HCF tables.

are known as in Figure 4.8. HCF will be free of false positives as long as the table is updated with the correct hop-counts when client hop-counts change. Furthermore, under hop-count clustering, we observe no noticeable increase in false negatives compared to the approach of 32-bit Strict Filtering. Thus, one cannot see the difference in Figure 4.8 due to their having similar numbers of false positives and negatives. Compared to the 24-bit aggregation, the clustering approach is more accurate but consumes more memory. Figure 4.10 shows the number of table entries for all web servers used in our experiments. The x-axis is the ID of the web server ranked by the number of client IP addresses, and the y-axis is the number of table entries. In the case of 32-bit Strict Filtering, the number of table entries for each server is the same as the number of client IP addresses. We observe that the hop-count-based clustering increases the size of HCF table, by no more than 20% in all but one case (36%).

4.5.2 Pollution-Proof Initialization and Update

To populate the HCF table initially, an Internet server should collect traces of its clients that contain both IP addresses and the corresponding TTL values. The initial collection period should be commensurate with the amount of traffic the server is receiving. For a very busy site, a collection period of a few days could be sufficient, while for a lightly-loaded site, a few weeks might be more appropriate.

Keeping the IP2HC mapping up-to-date is necessary for our filter to work in the Internet where hop-counts may change. The hop-count, or distance from a client to a server can change as a result of relocation of networks, routing instability, or temporary network failures. Some of these events are transient, but changes in hop-count due to permanent events need to be captured.

While adding new IP2HC entries or capturing legitimate hop-count changes, we must foil attackers' attempt to slowly pollute HCF tables by dropping spoofed packets. One way to ensure that only legitimate packets are used during initialization and dynamic update is through TCP connection establishment, an HCF table should be updated only by those TCP connections in the `established` state [118]. The three-way TCP handshake for connection setup requires the active-open party to send an ACK (the last packet in the three-way handshake) to acknowledge the passive party's initial sequence number. The host that sends the SYN packet with a spoofed IP address will not receive the server's SYN/ACK packet and thus cannot complete the three-way handshake. Using packets from established TCP connections ensures that an attacker cannot slowly pollute an HCF table by spoofing source IP addresses. While our dynamic update provides safety, it may be too expensive to inspect and update an HCF table with each newly-established TCP connection, since our update function is on the critical path of TCP processing. We provide a user-configurable parameter to adjust the frequency of update. The simplest solution would be to maintain a counter p that records the number of established TCP connections since the last reset of p. We will update the HCF table using packets belonging to every k-th TCP connection and reset p to zero after the update. p can also be a function of system load and hence, updates are made more frequently when the system is lightly-loaded.

We note that mapping updates may require re-clustering which may break up a node or merge two adjacent nodes on a 24-bit tree. Re-clustering is a local activity, which confines itself to a single 24-bit tree. Moreover, since hop-count changes are not a frequent event in the network as reported in [85] and confirmed by our own observations, the overhead incurred by re-clustering is negligible.

4.6 Running States of HCF

Since HCF causes delay in the critical path of packet processing, it should not be active at all time. We therefore introduce two running states inside HCF: the *alert* state to detect the presence of spoofed packets and the *action* state to discard spoofed packets. By default, HCF stays in alert state and monitors the trend of hop-count changes without discarding packets. Upon detection of a flux of spoofed packets, HCF switches to action state to examine each packet and discards spoofed IP packets. In this section, we discuss the details of each state and show that having two states can better protect servers against different forms of DDoS attacks.

4.6.1 Tasks in Two States

Figure 4.6.1 lists the tasks performed by each state. In the alert state, HCF performs the following tasks: sample incoming packets for hop-count inspection, calculate the spoofed packet counter, and update the IP2HC mapping table in case of legitimate hop-count changes. Packets are sampled at exponentially-distributed intervals with mean m in either time or the number of packets. The exponential distribution can be precomputed and made into a lookup table for fast on-line access. For each sampled packet, IP2HC_Inspect() returns a binary number $spoof$, depending on whether the packet is judged as spoofed or not. This is then used by Average() to compute an average spoof counter t per unit time. When t is greater than a threshold T_1, HCF enters the action state. HCF in alert state will also update the HCF table using the TCP control block of every k-th established TCP connection.

HCF in action state performs per-packet hop-count inspection and discards spoofed packets, if any. HCF in action state performs a similar set of tasks as in alert state. The main differences are that HCF must examine every packet (instead of sampling only a subset of packets) and discards spoofed packets. HCF stays in action state as long as spoofed IP packets are detected. When the ongoing spoofing ceases, HCF switches back to alert state. This is accomplished by checking the spoof counter t against another threshold T_2, which should be smaller than T_1 for better stability. HCF should not alternate between alert and action states when t fluctuates around T_1. Making the second threshold $T_2 < T_1$ avoids this instability.

To minimize the overhead of hop-count inspection and dynamic update in alert state, their execution frequencies are adaptively chosen to be inversely proportional to the server's workload. We measure a server's workload by the number of established TCP connections. If the server is lightly-loaded, HCF calls for IP2HC inspection and dynamic update more frequently by reducing user-configurable parameters, k and x. In contrast, for a heavily-loaded server, both k and x are decreased. The two thresholds T_1 and T_2, used for detecting spoofed packets, should also be adjusted based on load. The general guideline for setting execution rates and thresholds with the dynamics of server's workload is given as follows:

$$Load \nearrow \quad \Rightarrow \quad Rates \searrow \quad \Rightarrow \quad Threshold \searrow$$

```
In alert state:
  for each sampled packet p:
      spoof = IP2HC_Inspect(p);
      t = Average(spoof);
      if ( spoof )
              if ( t > T_1 )
                      Switch_Action();
      Accept(p);

  for the k-th TCP control block tcb:
      Update_Table(tcb);
```

```
In action state:
  for each packet p:
      spoof = IP2HC_Inspect(p);
      t = Average(spoof);
      if ( spoof )
              Drop(p);
      else Accept(p);

      if ( t ≤ T_2 )
              Switch_Alert();
```

Figure 4.11: Operations in two HCF states.

Currently, however, we only recommend these parameters to be user-configurable. Their specific values depend on the requirement of individual networks in balancing security and performance.

4.6.2 Staying "Alert" to DRDoS Attacks

Introduction of the alert state not only lowers the overhead of HCF, but also makes it possible to stop other forms of DoS attacks. In DRDoS attacks, an attacker forges IP packets that contain legitimate requests, such as DNS queries, by setting the source IP addresses of these spoofed packets to the actual victim's IP address. The attacker then sends these spoofed packets to a large number of reflectors. Each reflector only receives a moderate flux of spoofed IP packets so that it may easily sustain the availability of its normal service, thus not causing any alert. The usual intrusion detection methods based on the ongoing traffic volume or access patterns may not be sensitive enough to detect the presence of such spoofed traffic. In contrast, HCF specifically looks for IP spoofing, so it will be able to detect attempts to fool servers into acting as reflectors. Although HCF is not perfect and some spoofed packets may still slip through the filter, HCF can detect and intercept enough of the spoofed packets to thwart DRDoS attacks.

4.6.3 Blocking Bandwidth Attacks

To protect server resources such as CPU and memory, HCF can be installed at a server itself or at any network device near the servers, i.e., inside the 'last-mile' region, such as the firewall of an organization. However, this scheme will not be effective against DDoS attacks that target the bandwidth of a network to/from the server. The task of protecting the access link of an entire stub network is more complicated and difficult because the filtering has to be applied at the upstream router of the access link, which must involve the stub network's ISP.

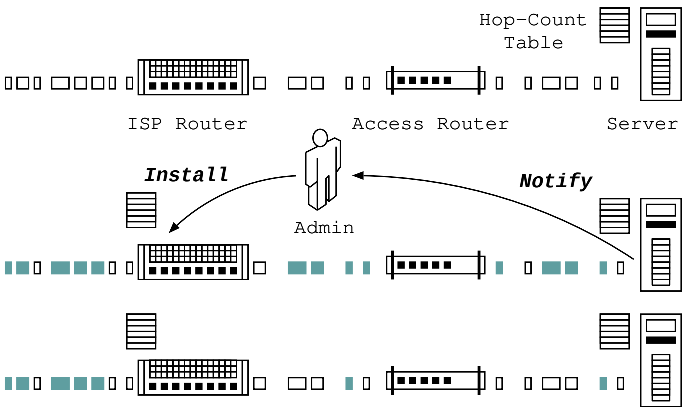


Figure 4.12: Packet filtering at a router to protect bandwidth.

The difficulty in protecting against bandwidth flooding is that packet filtering must be separated from detection of spoofed packets as the filtering has to be done at the ISP's edge router. One or more machines inside the stub network must run HCF and actively watch for traces of IP spoofing by always staying in the alert state. For complete protection, the access router should also run HCF in case attacking traffic terminates at the access router. This can be accomplished by substituting a regular end-host configured as a router. In addition, at least one machine inside the stub network needs to maintain an updated HCF table since only end-hosts can see established TCP connections. Under an attack, this machine should notify the network administrator who then coordinates with the ISP to install a packet filter based on the HCF table on the ISP's edge router.

Our two running-state design makes it natural to separate these two functions — detection and filtering of spoofed packets. Figure 4.12 shows a hypothetical stub network that hosts a web server that runs HCF. The stub network is connected to its upstream ISP via an access router and the ISP's edge router. Under normal network condition, the web server monitors its traffic and builds the HCF table. When attack traffic arrives at the stub network, HCF at the web server will notice this sudden rise of spoofed traffic and inform the network administrator via an authenticated channel. The administrator can have the ISP install a packet filter in the ISP's edge router, based on the HCF table. Note that one cannot directly use the HCF table since the hop-counts from client IP addresses to the web server are different from those to the router. Thus, all hop-counts need to be decremented by a proper offset equal to the hop-count between the router and the web server. Once the HCF table is enabled at the ISP's edge router, most spoofed packets will be intercepted, and only a very small percentage of the spoofed packets that slip through HCF, will consume bandwidth. In this case, having two separable states is crucial since routers usually cannot observe established TCP connections and use the safe update procedure.

4.7 Resource Savings

This section details the implementation of a proof-of-concept HCF inside the Linux kernel and presents its evaluation on a real testbed. The two concerns we addressed are the per-packet overhead of HCF and the amount of resource savings when HCF is active. Our measurements show that HCF only consumes a small amount of CPU time, and indeed makes significant resource savings.

4.7.1 Building the Hop-Count Filter

To validate the efficacy of HCF in a real system, we implement a test module inside the Linux kernel. The test module resides in the IP packet receive function, `ip_rcv`. To minimize the CPU cycles consumed by spoofed IP packets, we insert the filtering function before the code segment that performs the expensive checksum verification. Our test module has the basic data structures and functions to support search and update operations to the hop-count mapping.

The hop-count mapping is organized as a 4096-bucket hash table with chaining to resolve collisions. Each entry in the hash table represents a 24-bit address prefix, and it uses a binary tree to cluster hosts within the single 24-bit address prefix. Searching for the hop-count of an IP address consists of locating the entry for its 24-bit address prefix in the hash table, and then finding the proper cluster that the IP address belongs to in the tree. Given an IP address, HCF computes the hash key by XORing the upper and lower 12-bits of the first 24 bits of the source IP address. 4096 is relatively small compared to the set of possible 24-bit address prefixes so collisions are likely to occur. To estimate the average size of a chained list, we hash the client IP addresses from [29] into the 4096-bucket hash table to find that, on average, there are 11 entries on a chain, with the maximum being 25. Thus, we use fixed 11-entry chained lists. We determine the size of the clustering tree by choosing a minimum clustering unit of four IP addresses so the tree has a depth of six ($2^6 = 64$). This binary tree can then be implemented as a linear array of 127 elements. Each element in this array stores the hop-count value of a particular clustering. We set the array element to be the hop-count if clustering is possible, and zero otherwise.

To implement the HCF-table update, we insert the function call into the kernel TCP code past the point where the three-way handshake of TCP connection is completed. For every k-th established TCP connection, the update function takes the argument of the source IP address and the final TTL value of the ACK packet that completes the handshake. Then, the function searches the HCF table for an entry that corresponds to this source IP address, and will either overwrite the existing entry or create a new entry for a first-time visitor.

4.7.2 Experimental Evaluation

For HCF to be useful, the per-packet overhead must be much lower than the normal processing of an IP packet. We examine the per-packet overhead of HCF by instrumenting the Linux kernel to time the filtering function as well as the critical path in processing IP

	with HCF		without HCF	
scenarios	avg	min	avg	min
TCP `SYN`	388	240	7507	3664
TCP `open+close`	456	264	18002	3700
`ping` 64B	396	240	20194	3604
`ping` 1500B	298	124	35925	2436
`ping` flood	358	256	20139	3616
TCP bulk	443	168	6538	3700
UDP bulk	490	184	6524	3628

Table 4.2: CPU overhead of HCF and normal IP processing.

packets. We use the built-in Linux macro `rdtscl` to record the execution time in CPU cycles. While we cannot generalize our experimental results to predict the performance of HCF under real DDoS attacks, we can confirm whether HCF provides significant resource savings.

We set up a simple testbed of two machines connected to a 100 Mbps Ethernet hub. A Dell Precision workstation with 1.9 GHz Pentium 4 processor and 1 GB of memory, simulates the victim server where HCF is installed. A second machine generates various types of IP traffic to emulate incoming attack traffics to the victim server. To minimize the effect of caches, we randomize each hash key to simulate randomized IP addresses to hit all buckets in the hash table. For each hop-count look-up, we assume the worst case search time. The search of a 24-bit address prefix traverses the entire chained list of 11 entries, and the hop-count lookup within the 24-bit prefix traverses the entire depth of the tree.

We generate two types of traffic, TCP and ICMP, to emulate flooding traffics in DDoS attacks. In the case of flooding TCP traffic, we use a modified version of `tcptraceroute` [7] to generate TCP SYN packets to simulate a SYN flooding attack. In addition, we also repeatedly open a TCP connection on the victim machine and close it right away, which includes sending both SYN and FIN packets. Linux delays most of the processing and the

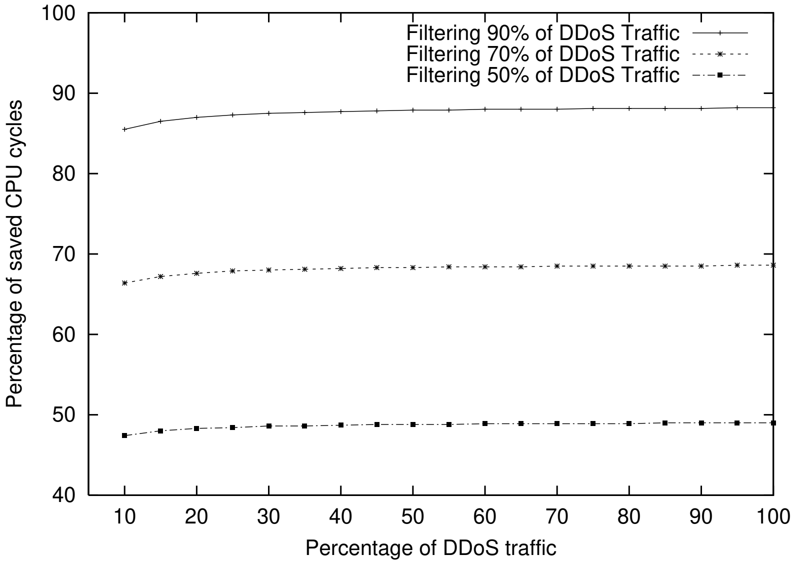


Figure 4.13: Resource savings by HCF.

establishment of the connection control block until receiving the final ACK from the host that does the active `open`. Since the processing to establish a connection is included in our `open + close` experiment, the measured critical path may be longer than that in a SYN flooding attack. To emulate ICMP attacks, we run three experiments of single-stream `ping`s. The first uses default 64-byte packets, and the second uses 1500-byte packets. In both experiments, packets are sent at 10 ms intervals. The third experiment uses `ping` flood (`ping -f`) with the default packet size of 64 bytes and sends packets as fast as the system can transmit. To understand HCF's impact on normal IP traffic, we also consider bulk data transfers under both TCP and UDP. We compare the per-packet overhead without HCF with the per-packet overhead of the filtering function in Table 4.2.

We present the recorded processing times in CPU cycles in Table 4.2. The column under 'with HCF' lists the execution times of the filtering function. The column under 'without HCF' lists the normal packet processing times without HCF. Each row in the table represents a single experiment, and each experiment is run with a large number ($\approx$

40,000) of packets to compute the average number of cycles. We present both the minimum and the average numbers. There exists a difference between average cycles and minimum cycles for two reasons. First, some packets take longer to process than others, e.g., a SYN/ACK packet takes more time than a FIN packet. Second, the average cycles may include lower-level interrupt processing, such as input processing by the Linux Ethernet driver. We observe that, in general, the filtering function uses significantly fewer cycles than the emulated attacking traffic, generally an order of magnitude less. Consequently, HCF should provide significant resource savings by detecting and discarding spoofed traffic. In case of bulk transfers, the differences are also significant. However, the processing of regular packets takes fewer cycles than the emulated attack traffic. We attribute this to TCP header prediction and UDP's much simpler protocol processing. It is fair to say that the filtering function adds only a small overhead to the processing of legitimate IP traffic. However, this is by far more than compensated by not processing spoofed traffic.

To illustrate the potential savings in CPU cycles, we compute the actual resource savings we can achieve, when an attacker launches a spoofed DDoS attack against a server. Given attack and legitimate traffic, a and b, in terms of the fraction of total traffic per unit time, the average number of CPU cycles consumed per packet without HCF is $a \cdot t_D + b \cdot t_L$, where t_D and t_L are the per-packet processing times of attack and legitimate traffic, respectively. The average number of CPU cycles consumed per packet with HCF is:

$$(1 - \alpha) \cdot a \cdot t_{DF} + \alpha \cdot a \cdot t_D + b \cdot (t_L + t_{LF})$$

with t_{DF} and t_{LF} being the filtering overhead for attack and legitimate traffic, respectively, and α the percentage of attack traffic that we cannot filter out. Let's also assume that the attacker uses 64-byte `ping` traffic to attack the server that implements HCF. The results for various a, b, and α parameters are plotted in Figure 4.13. The x-axis is the percentage of total traffic contributed by the DDoS attack, namely a. The y-axis is the number of CPU cycles saved as the percentage of total CPU cycles consumed without HCF. The figure contains a number of curves, each corresponding to an α value. Since the per-packet overhead of the DDoS traffic (20,194) is much higher than TCP bulk transfer (6,538), the percentage of the DDoS traffic that HCF can filter, $(1 - \alpha)$, essentially becomes the sole determining factor in resource savings. As the composition of total traffic varies, the percentage of resource savings remains essentially the same as $(1 - \alpha)$.

4.8 Discussion

Note that in our filtering accuracy experiments, we assume that the IP2HC mapping table holds the complete IP addresses of their clients and the hop-counts are static. However, in reality, there are always new requests coming in and hop-counts may change during a DDoS attack. Since we will not update or add new entries to the mapping table during DDoS attacks (in the action state), these newly-appeared requests and the legitimate ones with changed hop-counts will be dropped by HCF. Therefore, we want to know during a DDoS attack: (1) the percentage of incoming IP addresses that are the first-time visitors; and (2) the probability of a hop-count change that may invalidate an entry of the IP2HC mapping table. Since 90% of DDoS attacks are reported to last less than one hour [75], we set the duration of a DDoS attack to one hour.

The answer to the first question is highly dependent upon the clients' population, clients' access pattern and the duration of initializing the mapping table. Due to the limitation of the available server traces, we conduct a case study based on a one-month Web server trace that was collected at an academic department. In this case study, the initialization of the mapping table lasts a week. We use each individual week of the one-month trace as a different initialization procedure to collect the IP addresses. Then, we employ

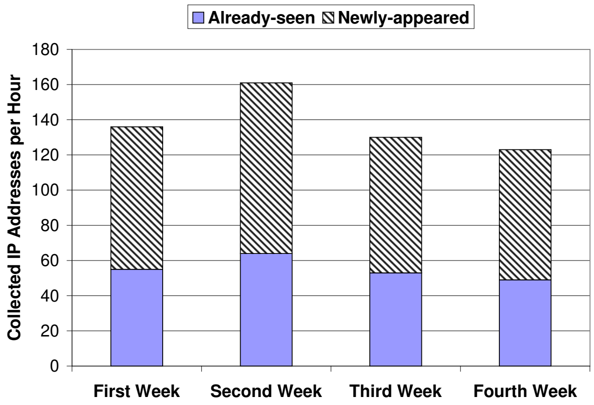


Figure 4.14: The fraction of newly-appeared IP addresses in the legitimate incoming requests

the subsequent day's trace of that week as input to find what fraction of incoming IP addresses are newly-appeared ones, under the one-week initialization. The results of this case study are shown in Figure 4.14. The percentages of newly-appeared IP addresses are consistent among the four different weeks we considered, which are around 60%. Although the 24-bit IP address can reduce the percentage of newly-appeared IP addresses, the reduction is less than 15%. This implies that more than a half of the incoming requests are newly-appeared, and three quarters of them will be dropped by HCF. However, this is only a specific case for a small educational Web server. With a much busier commercial Web server and longer initialization time, we expect that the results would be very different from those in this case study.

Based on the traceroute measurements of our stability study, in which over ten thousands distinct one-way paths were sampled, we elaborate on the distribution of the probability of a hop-count change within ten minutes[5] among our path samples. As shown in Figure 4.15, nearly 90% of the sampled one-way paths have less than 1% probability to

[5]The probe interval at traceroute measurements is ten minutes.

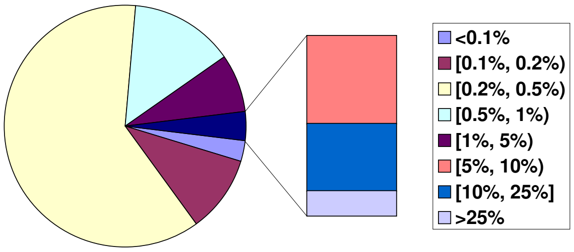


Figure 4.15: The distribution of probability of hop-count changes

change their hop-counts in ten minutes. With respect to this probability distribution, the expected probability of a hop-count change within ten minutes is 1.17%. Since we assume the duration of a DDoS attack is one hour, we need to derive the expected probability of a hop-count change within 60 minutes. As $\bar{P_h} = 1 - (1 - \bar{P_m})^6$, we know that $\bar{P_h}$ is 6.85%. It implies that during a DDoS attack, on average, 6.85% of the mapping table entries will be out-of-date.

Meanwhile, the inter-arrival time between two requests from the same IP address, i.e., the last request that is received before the outbreak of a DDoS attack and the first request after the outbreak of a DDoS attack, determines the fraction of out-of-date hop-counts in the mapping table at the start of a DDoS attack. Based on our Web traces, we find that the inter-arrival time highly depends on our diurnal activities. During working hours, our measured inter-arrival time ranges from a few minutes to over 40 minutes and its average is around 15 minutes. However, during off-working hours, the inter-arrival time ranges from one hour (lunch break) to nine hours (night sleep). So, if the DDoS attack is launched during working hours, the average fraction of out-of-date hop-counts at the start of a DDoS attack will be 1.75%. In contrast, if the DDoS attack is launched during off-working hours, the fraction of out-of-date hop-counts at the start of a DDoS attack will be between 6.85% and 47.1%. Here, we assume that the probability of path changes are uniformly distributed within the 24 hours of a day.

Although the percentage of out-of-date hop-counts is relatively high in the latter case, the out-of-date entries are the ones that have not visited the server for a long time and highly likely will not visit the victim during the DDoS attack; but those clients that visit the server quite often during off-working hours will have their hop-counts updated, and hence, hold accurate hop-counts in the mapping table during the DDoS attack. Therefore, the *average* increase of false positive ratio caused by hop-count changes happened before DDoS attacks will be around 1.75% during any given period.

4.9 Conclusion

In this chapter, we presented a hop-count based filtering scheme that detects and discards spoofed IP packets to conserve system resources. Our scheme inspects the hop-count of each incoming packet to validate the legitimacy of the packet. Using a moderate amount of storage, HCF constructs an accurate IP2HC mapping table via IP address aggregation and hop-count clustering. A pollution-proof mechanism initializes and updates entries in the mapping table. By default, HCF stays in *alert* state, monitoring abnormal IP2HC mapping behaviors without discarding any packet. Once spoofed DDoS traffic is detected, HCF switches to *action* state and discards most of the spoofed packets.

By analyzing actual network measurements, we have shown that HCF can remove 90% of spoofed traffic. Moreover, even if an attacker is aware of HCF, he or she cannot easily circumvent HCF. Our experimental evaluation has shown that HCF can be efficiently implemented inside the Linux kernel. Our analysis and experimental results have indicated that HCF is a simple and effective solution in protecting network services against spoofed IP packets. Furthermore, HCF can be readily deployed in end-systems since it does not require any network support.

However, the HCF cannot protect the newly-appeared requests during a DDoS attack. In our case study, the fraction of newly-appeared IP addresses in the legitimate IP addresses can be as high as 60%. Therefore, for an Internet server that is visited sporadically by most of its clients, the collateral damage caused by HCF may not be acceptable. On the other hand, HCF does guarantee the frequent visitors of an Internet service to be served during a DDoS attack.

CHAPTER V

IP EASY-PASS: EDGE RESOURCE ACCESS CONTROL

5.1 Introduction

To provide real-time communication services to multimedia applications or "subscribed" Internet users, the proposed network QoS (Quality of Service) infrastructure like DiffServ (Differentiated Service) [37] reserves network resources for real-time traffic. Within the network QoS architecture, however, the reserved network resources will become a conspicuous target to potential adversaries, and will be more vulnerable to packet forgery and resource embezzlement. We call such attacks, targeting at reserved network resources and violating network QoS guarantees, as DQoS (*Denial of Quality of Service*) attacks. Basically, there are two distinct DQoS attacks: (1) control flow attacks, e.g., killer reservation in RSVP (Resource ReSerVation Protocol) [121], which directly attack the signaling/control protocol in the control plane for network resource reservation and connection setup; and, (2) data flow attacks (e.g., resource theft in the data plane), in which bogus data packets grab the reserved bandwidth from the "owners," or genuine real-time data flows. Previous research efforts have focused on providing secure communication in the control plane for control flows, such as in ARQoS [1] and Authenticated QoS [2] projects. They proposed a secure RSVP, in which resources are reserved online using strong authentication, and subsequently, compliance with the reservation request parameters is verified.

However, little attention has been paid to defend against DQoS attacks in the data plane, and block the attacking traffic from consuming the reserved network resources. Currently, the usage of reserved network resource hinges on IP addresses and the setting of the ToS (Type of Service) field in the IP header, which can be easily spoofed. Even if we secure the QoS signaling procedure, an adversary within the same stub network or at a neighboring network that is connected with the same ISP, can passively monitor the ongoing traffic towards ISP edge routers. The adversary can then impersonate as legitimate sources by flooding the spoofed data packets that have the same IP header as valid data packets. The packet filters based on packet header information only, cannot screen out these spoofed packets. Moreover, to meet the SLA (Service Level Agreement) [67] between an ISP and its end-users, edge routers perform traffic shaping and policing according to the specified traffic profiles. Without a resource access control mechanism that can efficiently differentiate legitimate real-time traffic from spoofed packets, the traffic conditioning and policing conducted at ISP edge routers cannot protect the reserved network resource from embezzlement. On the contrary, the traffic policing at edge routers aggravates their vulnerability to flooding attacks by blindly dropping packets, since flooding bogus packets can easily cause traffic violation at the edge routers. Even a small amount of attacking traffic can disrupt the loss rate, delay and jitter guarantees, and seriously degrade the promised quality of service.

In this chapter, we propose a fast and lightweighted mechanism for resource access control at the ISP edge routers. It primarily protects real-time IP data flows, for which network resources are reserved, from DQoS attacks. Our resource access control mechanism is a checkpoint at edge routers, and is used for packet-level admission control. We call it an *IP Easy-pass*, because it is similar to the checkpoint at a toll road where only the cars with pre-paid stickers can go through the Easy-pass lane. Note that we apply the IP Easy-pass mechanism in the data plane, and assume that there is a secure channel for QoS signaling in the control plane between the ISP edge router and an end-host. Prior to data transfer, through the secure QoS signaling channel, the ISP edge router and the end-host must communicate shared secrets for generating Easy-passes.

Since IP is stateless, we attach a unique *Easy-pass* to every real-time data packet. Each IP Easy-pass is an encrypted order-sensitive information to warrant the legitimacy of the packet carrying it, and plays a role as an admission ticket which can be used only once and then becomes void. Stale *passes* are invalid. Even if an attacker can sniff the already-used *passes*, he cannot deceive the ISP edge routers by simply copying these void passes into spoofed packets. Thus, the freshness of a *pass* is crucial to the admission of the data packet carrying the pass. A correct sequence of Easy-passes is pre-determined by both sides. It is extremely difficult, if not impossible, for the third party to decrypt the garbled *passes* in a short time, which should be robust against cryptanalysis, and predict the correct sequence. This property ensures that an adversary cannot easily forge a valid new IP Easy-pass. The rule for access control is simple: the ISP edge router knows the correct sequence of passes; it accepts the packet with a *new pass* that is in the right track; otherwise, any packet with a duplicated or out of track pass, is classified as forgery and simply dropped.[1]

Because the generation and verification of *passes* are done on a per-packet basis, it has to be very fast and lightweighted, incurring as little overhead as possible. To generate encrypted *passes*, several parameters, including a symmetric private key and the fundamental elements of producing plain *passes*, are shared between the ISP edge router and the end-host. We attach an IP Easy-pass at the end of each IP packet as its trailer. The RC5 algorithm [93], a fast symmetric block cipher, is used for Easy-pass encryption and

[1]Considering possible packet losses, we admit normal out-of-sequence packets, which are still in the right track, as legal traffic.

decryption. The plaintext and ciphertext of Easy-pass are both 64 bits long. We implement the IP Easy-pass mechanism in the Linux kernel as a kernel module, and analyze its effectiveness against packet forgery and resource embezzlement. Finally, we evaluate its overhead and performance.

The remainder of the chapter is organized as follows. Section 5.2 discusses the motivation of our work. Section 5.3 reveals the vulnerability of the reserved network resource to DQoS attacks. Section 5.4 describes the generation of a plain Easy-pass, the encryption/decryption algorithm, the verification procedure and its embedding as an IP trailer. Section 5.5 describes the implementation of Easy-pass in the Linux kernel, and evaluates the effectiveness of the Easy-pass scheme in a network testbed. Finally, Section 5.6 concludes the chapter with a summary of our work.

5.2 Why Easy-pass

The IPSec [59, 108] protocol is often used to provide end-to-end secure communications at the network layer. It protects each IP packet, supporting address authentication, data integrity and confidentiality. Besides serving secure communications between two end-hosts, IPSec can also be used between two gateways to achieve VPN (Virtual Private Network) services over the public Internet. In the VPN model, the end-hosts in a local area network are trusted entities. However, IPSec is heavy-weight, incurring non-negligible overhead in data transmission [12, 50, 71]. A 10% increase of end-to-end latency overhead introduced by IPSec [50, 71] may not be a serious problem to file transfers or Web sessions. However, such an overhead may violate the delay requirements for delivering real-time data streams. In a study of Voice over IPSec, the effective bandwidth was observed to be reduced up to 50% with respect to VoIP [12].

Although IPSec can achieve host-to-edge authentication by using the tunnel mode, its security model is an overkill for protecting the reserved network resources at edges. We compare the Easy-pass with IPSec AH (Authentication Header) in tunnel mode in three different aspects: (1) bandwidth overhead: the length of Easy-pass is 8 bytes; in contrast, the header overhead incurred by IPSec AH tunnel mode is 44 bytes, which is 5.5 times as much as that of Easy-pass. (2) memory overhead at edges: to counter reply-attack, IPSec maintains a window of 64 sequence numbers for each IP flow. Since the sequence number is 32-bit long, the memory requirement of anti-reply in IPSec is 4096 bits for each IP flow; in contrast, Easy-pass only introduces 32-bit flag and 8-bit base variables to achieve the same goal. Therefore, at an edge router, IPSec consumes over 100 times the memory space of Easy-pass. (3) CPU overhead: we measure the Easy-pass CPU overhead on an off-the-shelf 600 MHz Pentium III PC with 256 RAM as shown in Section 5.5.3, which is 2μs; in contrast, in a similar platform, the CPU overhead of IPSec authentication (HMAC-MD5) ranges from 15 to 30 times more than that of Easy-pass with different packet sizes [88], since IPSec authentication covers the entire IP packet. Note that the addition of IPSec encapsulation overhead will further widen the gap of CPU cycle consumption between IPSec and Easy-pass.

Overall, IPSec is not suitable for protecting reserved network bandwidth, especially in wireless LAN and DSL environments. Instead, we need a lightweight edge-resource access authorization mechanism to protect the end-hosts that subscribe to the premium service, against the flooding attacks from malicious neighbors. For such DQoS problems, Easy-pass is the solution and its overhead does not cause violation of the stringent timing requirements of real-time traffic.

The hop integrity schemes proposed by Gouda *et al.* [47] support router authentication and hop-by-hop message integrity. The hop integrity protocols are executed at all routers in a network, and provide a minimum level of secure communications between two adjacent routers to defend against message modification and message replay. However, hop integrity does not keep track of individual data flows or sessions. It cannot prevent resource theft and message replay attacks between end-hosts and ISP edge routers. Moreover, the accumulation of per-hop overhead may violate the end-to-end QoS requirements.

Session-level admission control in the network QoS infrastructure has been studied for a decade or so. A number of schemes have been proposed [18, 58, 56, 121], such as measured-based admission control [56], distributed admission control [58], and endpoint admission control [18], just to name a few. The fundamental goal of session-level admission control for real-time applications is to accommodate new session requests without compromising the ongoing sessions' QoS guarantees. However, IP Easy-pass is a packet-level resource authorization mechanism, which protects the reserved network resource from attacking traffic.

To protect Internet servers and network resources, various packet filtering techniques have been proposed and used as defensive mechanisms [57, 40, 63, 84, 92, 106, 115, 119]. The packet filters installed at Internet servers or their nearby firewalls [57, 92, 119] only prevent such IP packets from reaching the victim, but cannot protect network resources from theft and abuse (e.g., insider attacks). The intermediate-router-based packet filters [63, 84, 106, 115] can block further propagation of flooding traffic in the network, but cannot protect the upstream network resources. The ingress/egress filtering techniques [40] at ISP edge routers can prevent IP packets from being spoofed as an outsider, but cannot discriminate bogus IP packets with valid IP addresses within the same local ISP.

5.3 Vulnerability of Reserved Network Resource

We will now show that the reserved network resource in the QoS infrastructure like DiffServ [37] is vulnerable to spoofed traffic, and the premium service [36] provided by an ISP is susceptible to flooding attacks. Two entities associated with the reserved network resource are ISP edge routers — the service provider — and end-hosts who have subscribed to the premium service. Note that the IP Easy-pass mechanism, which is independent of any network QoS architecture, is not tied with the DiffServ architecture; and can be applied to any other QoS infrastructure. We discuss Easy-pass in the context of DiffServ, just for the sake of presentation and experimentation. Within the DiffServ infrastructure, EF (Expedite Forwarding) is intended to support premium service for real-time applications that require strict guarantees on bandwidth, delay, jitter and packet loss. In this chapter, the network resource reserved for EF traffic at ISP edge routers, including link bandwidth and router buffers, is the *target* of malicious attacks.

5.3.1 Attacking Model and Assumptions

DQoS attacks in the data plane are made by malicious insiders or unfriendly neighbors, who have not subscribed to the premium service. We assume that the adversary is located in the same stub network as the victim, or at a different stub network but is connected to the same ISP. The adversary can be a cracked machine, an unhappy employee, or a naughty freshman. As the joint that connects the stub networks to the rest of Internet, each ISP edge router performs traffic conditioning and policing for EF traffic: any violation of SLA will be punished by dropping packets. However, such traffic conditioning works only if no packets are spoofed; on the contrary, a blindly dropping policy makes the premium service much more susceptible to the flood of spoofed EF packets. The reason for this is that the network resource reserved for premium service is only a small portion of the link bandwidth and buffer space at ISP edge routers, due mainly to the following usage and financial factors:

- most of traffic will continue to be best-effort (BE), since BE service is free; and
- those users who have subscribed to the premium service will not waste their money by overbooking the resource.

Therefore, flooding even a small amount of spoofed premium service traffic can easily compromise its quality.

There may be several internal routers and switches between an end-host and its ISP edge router. For example, from the EECS Department at the University of Michigan, we use `traceroute` to track the route to `www.cnn.com`, and find that it takes 4 hops to reach `mich.net`, the regional ISP for the University of Michigan. The result is shown as follows. The topology of the campus network at the University of Michigan is shown in Figure 5.1. Within each segmental LAN environment, there is enough bandwidth — e.g., the link speed of CAEN (Computer-Aided Engineering Network) varies between OC-12 (622 Mb/s) and OC-48 (2.4 Gb/s) — to support real-time applications, such as VoIP (Voice over IP). It is also a common practice that no traffic conditioning is performed at internal routers. We further assume that these internal routers and switches are not sabotaged, so that in-flight packets cannot be intercepted and modified. However, the adversary can eavesdrop all the traffic between the end-host and the ISP edge router, and inject any packet at his will.

```
> 1  eecs2s-gw (141.213.10.1)
> 2  CAEN-EECS-GW (141.213.3.4)
> 3  141.213.101.4 (141.213.101.4)
> 4  141.213.127.14 (141.213.127.14)
> 5  atm3-0x1.michnet8.mich.net (192.122.183.13)
```

Suppose `mich.net` provides VoIP service, then between `mich.net` and users at the University of Michigan, there must be SLAs on the provision of network resource for sup-

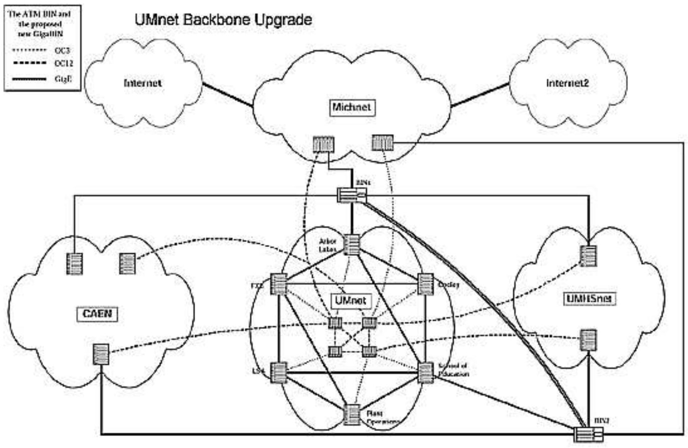


Figure 5.1: The topology of UM campus networks

porting the voice traffic. Typically, the outgoing voice traffic will be policed at the edge router of `mich.net` before traversing its ISP network. Carrying voice traffic does not cost much bandwidth, its provision at the ISP edge router should be a small portion of its link capacity. Under such a scenario, a malicious user, who has not subscribed to the premium service, inside the campus network may monitor the ongoing voice traffic, and then flood duplicate or spoofed packets, simply launching a reply attack. The perceived quality of VoIP service will be degraded significantly. Moreover, during certain peak times of the day (e.g., 10 am – 12 noon and 2 pm – 4 pm when most telephone calls are made), the individual callers may compete with each other for the reserved bandwidth. A rejected caller may disrupt the already-established calls by flooding spoofed packets, thereby degrading the ongoing phone conversations. Once several such conversations have been *forced* to abort, the resource left will allow the ISP edge router to admit such aggressive caller's request even at these peak times.

We prevent such DQoS attacks by deploying the IP Easy-pass mechanism at the sending end-host and the ISP edge router. The sending end-host generates and encrypts an Easy-pass, then attaches it to each outgoing EF packet. The ISP edge router decrypts and verifies the received Easy-pass, then detaches it from the EF packet before forwarding the packet to the next-hop. Before EF data transmission, the end-host must have already set up a secure channel with the ISP edge router for resource reservation signaling. By utilizing this secure channel, the end-host and the ISP edge router share the secret information for constructing Easy-passes. Such an assumption is quite realistic. Besides the ARQoS [1] and Authenticated QoS [2] projects, the IETF NSIS Working Group [110] was recently formed to develop a set of standards for end-to-end resource reservation and QoS signaling between different administrative domains. Our Easy-pass scheme can be incorporated into the end-to-edge (host-to-network) signaling framework of NSIS so that the shared secrets can be exchanged between the end-host and the edge router once a trust relationship has been established between them. Furthermore, The Easy-pass mechanism can also be extended to validate the legitimacy of high-tiered traffic across two adjacent ISP edge routers that belong to two different ISPs.

5.3.2 Flooding Attacks Disrupting Premium Service

To demonstrate the susceptibility of premium service to flooding attacks, we performed a series of experiments on our testbed. The topology of our testbed is shown in Figure 5.2. Our testbed consists of one Linux-based software router `rtcl11` and three end-hosts. One end-host `rtcl13` is used as the receiving host, and the other two as sending hosts — one `rtcl9` is a normal service subscriber and the other `rtcl10` is the adversary. Based on a set of traffic control (tc) APIs in the Linux kernel, we built a router configuration agent and placed it on the router of our testbed in order to configure the traffic control blocks inside the router. At the end-hosts, we built traffic generation agents, which are a modified version of Iperf [35], to generate both TCP and UDP traffic. To facilitate the experimental setup, a fast Ethernet switch is used to connect the end-hosts.

Each machine in the testbed runs on a 600 MHz Prentium III processor with 256M RAM.

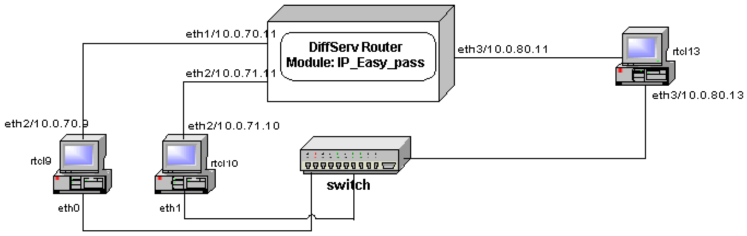


Figure 5.2: The network topology of the DiffServ testbed

In order to support real-time traffic, we deployed the DiffServ EF PHB (Per-Hop Behavior) [36] at the router. In our experiments, 500Kbps of the link bandwidth between the router `rtcl11` and the receiving host `rtcl13` is reserved for the EF traffic originating from the legitimate end-host `rtcl9`. Inside the router, the TBF (Token Bucket Filter) is used for EF traffic conditioning, and the packet scheduling policy is PQ (Priority Queueing). Due to the simple testbed setup, the measured end-to-end delays are not realistic, so we do not include them in our chapter.

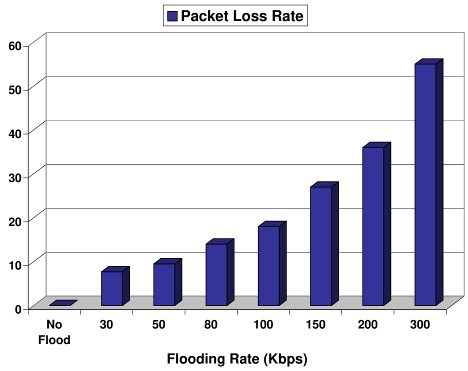


Figure 5.3: Vulnerability of EF traffic to flooding attacks (I) (loss rate)

A well-behaved EF traffic carried by UDP is transmitted from `rtcl9` to `rtcl13`. The packet size is 1000 bytes. If no flooding attacks occur, the reserved network resource

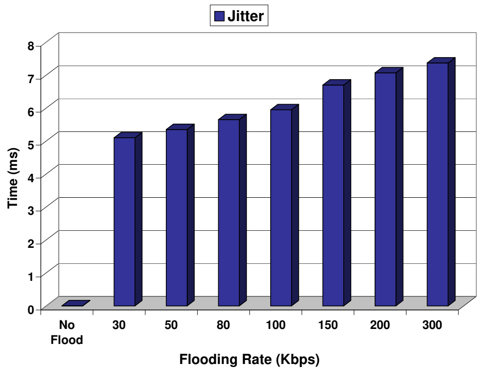


Figure 5.4: Vulnerability of EF traffic to flooding attacks (II) (jitter)

serves the EF traffic well, and achieves the goal of low loss rate, low delay and low jitter. However, under flooding attacks, even a small amount of flooding traffic can seriously degrade the quality of service received by EF traffic. As shown in Figure 5.3, for the flooding rate of 100 Kbps that is only one fifth of the reserved link bandwidth, the loss rate is increased from 0 to 19%. The measured end-to-end delay-jitter is plotted in Figure 5.4. Under the flooding rate of 100 Kbps, the jitter surges from 3μs to 5.95 ms, clearly showing that the flooding traffic increases the jitter by several orders of magnitude. Such serious degradation in packet loss and jitter will make any real-time application like VoIP or video-conferencing infeasible.

5.4 Description of IP Easy-pass

At an end-host subscribed to EF service, we create a unique *pass* and attach it to each outgoing EF packet. A valid *pass* authorizes its carrier packet to access the reserved network resource at downstream routers. In plain-text, a *pass* is just a random number. However, a sequence of *passes* for an EF data flow are generated according to certain rules. Both the sending host and the ISP edge router agree *a priori* on the generation rules. In this section, we will show how to generate a sequence of *passes* for an EF data flow, then discuss the choice of encryption/decryption algorithm, and finally, present the verification and embedding of Easy-passes.

5.4.1 Generation of Easy-Pass

The main parameters in generating a sequence of plain-text *passes* include a *nonce* Λ, a *gradient* Δ, and a *direction* γ. Of the tuple $\{\Lambda, \Delta, \gamma\}$, the first two elements are random numbers, and the last one is a boolean. The nonce is the starting point in generating a sequence of Easy-passes. The gradient is the span between two consecutive Easy-passes. The direction determines if the trend of the Easy-pass sequence is in an increasing or decreasing order, and its value is decided at run-time.

Assuming that the number of bits in an Easy-pass is N, the range space Ω of Easy-pass is 2^N. The chosen space for the initial nonce Λ is $[0, \Omega]$; and that for the gradient is $\{\ \Delta \mid 0 \leq \Delta \leq 2^k\ \cap\ (\Omega\ mod\ \Delta \neq 0),\ \ k << N\}$. The growth direction of Easy-passes is dynamically determined by the chosen Λ. If $\Lambda > \Omega/2$, then the value of Easy-passes decreases; on the other hand, if $\Lambda < \Omega/2$, then the value of Easy-passes increases. Let $d(.)$ be the direction of Easy-passes for EF traffic transmitted between end-host m and the ISP edge router: '0' for increasing order and '1' for decreasing order.

$$d_m(\Lambda) = \gamma = \begin{cases} 0 & \text{if } \Lambda \leq \Omega/2; \\ 1 & \text{if } \Lambda > \Omega/2. \end{cases} \tag{5.1}$$

The plain Easy-pass is the sum of an initial random number Λ and the corresponding gradient Δ. Let $V(.)$ be the value of Easy-pass for the n-th data transmission. The construction of a plain Easy-pass is described as follows:

$$V(n) = \Lambda + (-1)^{\gamma} \cdot (n-1) \cdot \Delta \tag{5.2}$$

where n is the transmission order of a data packet starting from 1. The algorithm for constructing an IP Easy-pass is illustrated in Table 5.1. Step 0 in Table 5.1 shows the selection of two fundamental elements $\{\Lambda, \Delta\}$ during secure QoS signaling phase (e.g. via secure RSVP). The rest of steps in Table 5.1 present the construction algorithm of an Easy-pass. When the value of an Easy-pass reaches the lower_limit, 0, or the upper_limit, Ω, we need to wrap around the value to continue within the range space. The number of packets per round is Ω/Δ.

$$V(n) = \begin{cases} V(n) - \Omega & \text{if } V(n) > \Omega; \\ V(n) + \Omega & \text{if } V(n) < 0. \end{cases} \tag{5.3}$$

To prevent the wrap-around sequencer from coinciding with the previous values, in accordance to number theory, we require that the gradient Δ be a prime number.

To exemplify the working mechanism of IP Easy-pass, a sequence of Easy-passes in plaintext are shown in Figure 5.5. In this simple example, we assume that the sample space for nonce Λ is [0,120], and there are Easy-pass sequences with respect to different initial nonces. The randomly-selected nonce of the first and second sequences are 57 and 67, respectively. Both have the gradient $\Delta = 17$. Since the nonce of the first sequence, 57, is smaller than 60, the median of Ω, the first sequence is increasing. In contrast, the second sequence is decreasing, due to its nonce being larger than 60. Once the plaintext value of Easy-pass is computed, we encrypt it and attach the encrypted Easy-pass to the outgoing EF packet as an IP trailer. We discuss the choice of encryption/decryption algorithm in the next section.

5.4.2 Choice of Encryption/Decryption Algorithm

Since encryption/decryption is performed on each EF data packet, the overhead incurred by the encryption/decryption algorithm should be as low as possible without degrading its security. In the Easy-pass mechanism, we employ RC-5 [93] to encrypt Easy-

Table 5.1: Pseudocode of Constructing an Easy-pass

```
0. Before data transmission:

  nonce ← Λ;     //select a random number for nonce
  gradient ← Δ;  // select a prime number from [0, 2^k]

1. At the very beginning of data transmission:

  n ← 1;
  W_0 ←Λ;
  If (Λ < ⌊Ω/2⌋)
        γ ← 0;
  Else
        γ ← 1;

2. On data transmission, build the n-th Easy-pass W_n as:

  Switch (γ) {
     Case 0 :
              W_n ← W_{n−1} + Δ;
              If (W_n > Ω)
                 W_n ← W_n − Ω;    // wrap around
     Case 1 :
              W_n ← W_{n−1} − Δ;
              If (W_n < 0)
                 W_n ← W_n + Ω;    // wrap around
  }
  n ← n + 1;
```

passes at the end-host, and then decrypt it at the ISP edge router. This is mainly because (1) RC-5 is one of the fastest encryption/decryption algorithms available, and (2) RC-5 is fully parameterized, allowing flexible choices for its parameters. Briefly, RC-5 is a symmetric block cipher, in which the plaintext and ciphertext are fixed-length bit sequences. RC-5 is word-oriented, with a variable number of rounds and a variable-length cryptographic key. It is fast and has low memory requirement. Finally, RC-5 provides high-level security when parameter values are chosen properly.

The parameters in RC-5 that are adjustable include the word size in bits w, the number

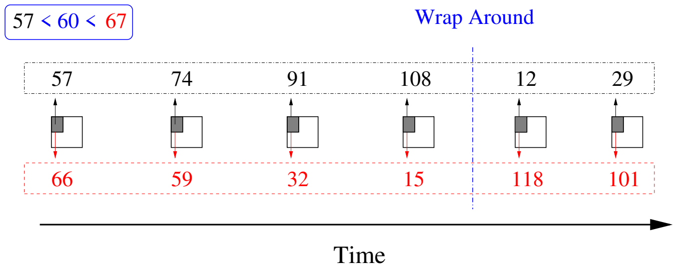


Figure 5.5: The sequence of Easy-passes

of rounds r, and the number of bytes in secret key b. Note that RC-5 uses an expanded key table, S, that is derived from the secret key. The size of table S also depends on the number of rounds, which is equal to $2 \cdot (r + 1)$ words. RC-5 allows a range of parameter values so that one may choose a certain set of parameters that balance the requirement between security and performance. Moreover, applications can adjust these parameters when their own requirements change.

To test the efficiency of RC-5, we choose a secure option of RC5-32/10/16, where $w = 32$, $r = 10$, and $b = 16$, so that the secret key length is 128 bits. We conduct simple experiments on an off-the-shelf 550 MHz Pentium III PC with 256M RAM, running Linux kernel 2.4.7. The CPU time for encryption and decryption on the option of 32/10/16 is only $1\mu s$. Next, we vary the number of rounds and length of the secret key, to find out which one dominates the consumption of CPU cycles. As shown in Figure 5.6, the increase in the number of rounds from 10 to 80 linearly increases the CPU overhead, but increasing the secret key length does not affect the CPU overhead. Therefore, we choose 16 bytes (128 bits) for the secret key length, instead of 32 or 64 bits that are easier to break. On the other hand, we choose the number of rounds to be 10 in order to reduce the resulting CPU overhead.

The input (plaintext) and output (ciphertext) of RC-5 are two-word long. Since we

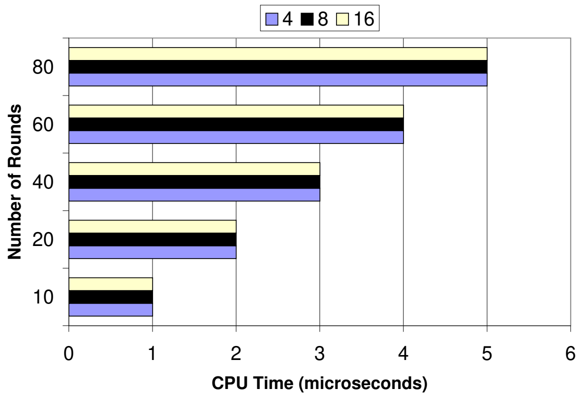


Figure 5.6: CPU overhead for RC-5 encryption/decryption

choose the length of word as 32-bit, the length of Easy-pass is 64 bits. Then, the range space Ω is 2^{64}. Such a large space guarantees avoidance of Easy-pass collision, i.e., no two data packets within the same real-time session will have the same Easy-pass. For example, even if the real-time session reserves a 100 Mbps bandwidth, its average packet size is only 50 bytes, and its duration lasts for four weeks, the required space to avoid any Easy-pass collision is still less than 2^{40}.

5.4.3 Verification of Easy-passes

At the ISP edge router, after decrypting the encrypted Easy-pass in each received EF packet, we verify its legitimacy according to the generation rule of Easy-passes. The first step of the verification procedure is simply checking if $\frac{V_d - \Lambda}{\Delta}$ is an integer, where V_d is the value of the decrypted Easy-pass of the received EF packet. The next step is to make sure that the integer is fresh, i.e., it did not appear before.

If there is no out-of-order transmission between the end-host and its ISP edge router, after decrypting the Easy-pass from each valid EF packet, the ISP edge router will see a sequence of random numbers starting from Λ with an interval of Δ. Assume that the last *checking number*, $\frac{V_d - \Lambda}{\Delta}$, is an integer I, then the correct *checking number* for the one being validated should be $I+1$ if $I>0$ or $I-1$ if $I<0$. The correct sequence of checking numbers should be a series of $\{0, 1, 2, 3, \cdots\}$ or $\{0, -1, -2, -3, \cdots\}$.

However, in case of congestion, packet losses or out-of-order packet arrivals may occur at the edge router. To account for possible holes in the sequence of checking numbers, we introduce a range-window. Given the maximum possible out-of-order value within the first-mile environment is m, we set the range-window size to $2m$. We also introduce two variables: one is *base* to record the *checking number* of the latest received in-order packet; the other is a $2m$-bit long variable, called *flag* as an bit-index-array to record the received out-of-order packets, whose default value is zero. In this chapter, we set m as 16, so the range-window covers 32 packets. Then, $flag$ is a 32-bit word. Note that with the change of the real condition, the value of m is adjustable.

An out-of-order delivery or packet loss is detected, when the difference between the incoming packet's checking number and the *base* is larger than 1. As shown in Figure 5.7, at time t_1, the *base* is 32 but the received packet's checking number is 35. Since $35-32=3>1$, the out-of-order delivery or packet loss is detected. The value of *base* is not updated until the holes in the sequence are filled or the range-window reaches its limit later.

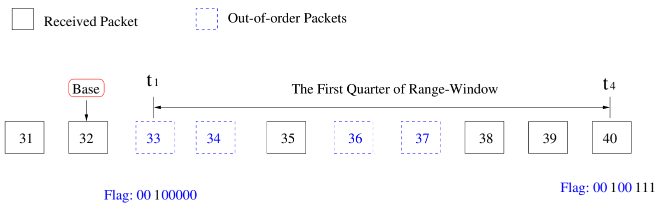


Figure 5.7: Tracking the out-of-order delivery

In Figure 5.7, we only show one quarter of the range-window, and the first byte of *flag* is initially set to "00000000" as default. When the packet with the checking number 35 arrives, the third bit in the *flag* is set to "1", indicating that the packet has been received. Any later arrival of a packet with the same checking number will be treated as a duplicate and discarded. Following this rule, at time t_4, the first byte of flag becomes ”00100111”. The index of the received packet in the *flag* is its *checking number* deducted by *base*. In this manner, the verification module keeps track of the holes in the sequence of checking numbers, and updates the list of holes by changing the corresponding bit in *flag* from ’0’ to ’1’ when one of them is filled.

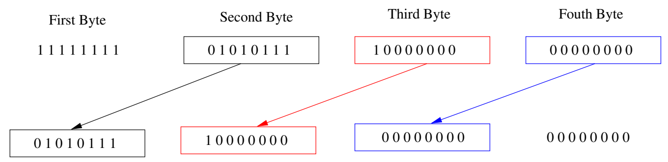


Figure 5.8: Left shift the flag for 8 bits to update the out-of-order state

Once the first byte of *flag* becomes ’0xFF’ or the range-window reaches its right-most end, we shift the *flag* to left for 8 bits and reset the last byte as ’0x00’, as shown in Figure 5.8. Then, we increase the *base* by 8 correspondingly. This shift is very important to seamlessly keep track of the status of the out-of-order delivery using as little memory as possible. The pseudocode of verifying an Easy-pass is shown in Table 5.2.

In summary, the filtering rule for weeding out attacking traffic at the ISP edge router is simple: if the checking number meets any one of the following conditions: (1) it is not an integer; (2) it is smaller than the value of *base*; and (3) its value is larger than the value of *base* but the corresponding bit in *flag* is already set to 1. Then, the packet carrying such a pass will be identified as spoofed and hence discarded without further processing.

5.4.4 Embedding of Easy-Pass

We embed an Easy-pass, which is 64 bits long, as the trailer to an IP packet. The attachment of an Easy-pass is shown in Figure 5.9. Although the occurrence of packet fragmentation between an end-host and its ISP edge routers is rare, the possible fragmentation will detach the *pass* from the fragments, except for the last one. A malicious attacker

Table 5.2: Pseudocode of Verifying an Easy-pass

```
Decrypt the Easy-pass to get V_d

 C_n = |(V_d−Λ)/Δ|;    // derive the absolute checking number
 If (C_n is not integer)
     discard the packet     // attacking packet
 Else
   i = C_n - B;  // get the index
   If (i < 0 )
      discard the packet  // duplicate packet
   Else
      If ( i > 1)  // out-of-order
        Switch (flag[i]) {   // check the bit in flag
          Case '1' :
            discard the packet  // duplicate packet
          Case '0' :
            accept the packet;
             flag[i] = '1';
            If (flag[1:8] == '0xFF' || flag[32] == '1')
                flag << 8;  // left shift 8 bits
                B = B + 8;  // update the base
               // no holes in the first byte or
               // reaches the highest-end
        }

      If ( i == '1')  // in the right track
         accept the packet
          B = C_n;   // update the base
```

can exploit the fragmented packets to steal the reserved resource. To overcome the fragmentation problem, between an end-host and the ISP edge router, we require that an MTU discovery should be performed before data transmission, guaranteeing no fragmentation in the corresponding data flow.

The recent Internet measurement [97] has shown that less than 1% IP packets in the Internet are fragmented. More importantly, it confirmed that IP packet fragmentation is still "considered harmful." Therefore, enforcing no fragmentation not only fills the security holes, but also improves end-to-end performance.

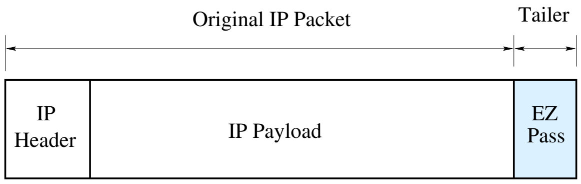


Figure 5.9: Embedding the Easy-pass into an IP packet

One advantage of attaching the Easy-pass value as a trailer to each IP packet is that it is attached by the end-host and subsequently removed upon verification by the edge router before passing the packet on to the ISP network. Therefore, none of the downstream routers and the receiving end-hosts need to be modified to accommodate the proposed scheme. Another advantage is that like MPLS, it allows attachment of the Easy-pass value completely transparent to the transport protocol (TCP, UDP) used by the application-layer.

5.5 Implementation and Evaluation

The IP Easy-pass mechanism is implemented in the Linux kernel, and its effectiveness against flooding attacks is evaluated on the DiffServ testbed. In this section, we first describe the implementation of Easy-pass, and then perform a series of experiments to demonstrate the Easy-pass's protection on real-time traffic. Finally, we measure the overhead incurred by Easy-pass and discuss its impact on end-to-end performance.

5.5.1 Implementation of Easy-pass

We implemented the IP Easy-pass scheme as two loadable Linux kernel modules. One module, *HostModule*, is located at the end-host which calculates and attaches 64-bit Easy-passes to the packets; and the other module, *RouterModule* installed at the edge router, extracts the Easy-pass and verifies the packet before forwarding it to the next-hop.

The *HostModule* uses a hook that has been added to IP layer transmission function (`ip_build_xmit()`) of the Linux kernel version 2.4.7. After preparing the IP packet for transmission on the network, *HostModule* calculates the Easy-pass value, based on the current packet count from the designated data flow. The module then encrypts the field at run-time using RC5 and attaches it after the payload as the trailer of the packet. The whole packet is then placed on the outgoing interface.

On the edge-router side, *RouterModule* uses a hook added to the IP layer receiving function (`ip_rcv()`) to retrieve the encrypted Easy-pass value from the IP packet. The module then decrypts and verifies the pass. If the packet is found to have the correct Easy-pass value, it is allowed to proceed through the protocol stack. Otherwise, the packet is dropped.

Compared with inserting Easy-pass into IP header option fields, attaching the Easy-pass as a trailer has the advantage of being transparent to the protocol checking and processing. Neither header modification nor payload data shifting is required in this case. The only fields that have to be updated are the IP packet total length and the IP checksum, which are done inside our modules. This eases the implementation of our Easy-pass.

5.5.2 Effectiveness Against Resource Theft

To validate the effectiveness of Easy-pass against malicious attacks, we performed a series of experiments on our DiffServ testbed as described in Section 5.3.2. Basically, our experiments can be divided into two groups: one is for protecting low-rate EF traffic, such as audio streams, and the other is for protecting high-rate EF traffic, such as video streams. Since multimedia traffic (real-time audio and Video) is usually transported by UDP [64, 70], all the EF traffic in our experiments are carried by UDP.

Protection of Low-rate (Audio) Traffic

As the maximum bandwidth required for supporting VoIP is 80 Kbps [6], we set the source sending rate and the reserved bandwidth at the router to 80 Kbps, respectively.

	w/o EZ-Pass		w/ EZ-Pass	
Flood Rate	254	502	254	502
20K	22%	21%	0	0
40K	38%	35%	0	0
60K	45%	40%	0	0
80K	54%	54%	0	0
120K	68%	71%	0	0
160K	80%	85%	0	0

Table 5.3: Packet-loss rate for low-rate EF traffic

	w/o EZ-Pass		w/ EZ-Pass	
Flood Rate	254	502	254	502
20K	6.8ms	9.3ms	5.2μs	5.8μs
40K	7.2ms	10.2ms	5.8μs	6.3μs
60K	7.4ms	10.5ms	6.1μs	6.7μs
80K	7.5ms	10.8ms	6.2μs	7.0μs
120K	7.9ms	11.2ms	6.5μs	7.3μs
160K	8.6ms	11.9ms	6.7μs	7.8μs

Table 5.4: End-to-end delay-jitter for low-rate EF traffic

Since the typical packet sizes for real-time audio are 254 and 502 bytes [70] (depending on the encoding rate), we vary the packet size from 254 to 502 bytes in this set of experiments.

Table III illustrates the EF packet-loss rate for different flooding rates ranging from 20 Kbps — one fourth of reserved bandwidth — to 160 Kbps, twice the reserved bandwidth. Without the Easy-pass, the packet-loss rate ranges from 21% to 80%. Under the same flooding rate, the larger packet size incurs a slightly higher packet-loss rate, due to a slightly larger burst size. In contrast, with the Easy-pass, all the attacking packets are identified and discarded. That is, the reserved bandwidth is saved, and hence, no legitimate packets are dropped.

Table IV presents the corresponding results for end-to-end delay-jitter obtained from the same set of experiments. Without Easy-pass the jitter ranges from 6.8 to 11.9 ms. Also, for the same flooding rate, the larger packet size incurs a slightly higher jitter. In contrast,

	w/o EZ-Pass		w/ EZ-Pass	
Flooding rate	1000	800	1000	800
300K	22%	21%	0	0
500K	30%	27%	0	0
800K	37%	33%	0	0
1M	42%	38%	0	0
1.2M	48%	44%	0	0
1.5M	57%	53%	0	0
3M	67%	62%	0	0

Table 5.5: Packet-loss rate for high-rate EF traffic

with Easy-pass, the jitter remains in the same order of magnitude as the one without any flooding attacks.

Protection of High-rate (Video) Traffic

Since MPEG-1, a popular video compression technique, has an encoding rate of 1.5 Mbps, in our second group of experiments, we set the source sending rate to 1.5 Mpbs, and reserve 1.5 Mbps bandwidth at the router. Since the typical packets for real-time video are 800 and 1000 bytes long [64], the EF packet size is varied from 800 to 1000 bytes in these experiments.

Table V presents the EF packet-loss rate for different flooding rates ranging from 300 Kbps, one fifth of reserved bandwidth, to 3 Mbps, twice the reserved bandwidth. Without Easy-pass, the packet loss rate ranges from 21% to 67%. As in the low-rate case, for the same flooding rate, the larger packet size incurs a slightly higher packet-loss rate. With Easy-pass, all the flooding traffic is identified and discarded, hence protecting the reserved bandwidth. None of the legitimate packets were dropped.

Table VI presents the results of end-to-end delay-jitter for the EF traffic. Unsurprisingly, without the Easy-pass, the jitter ranges from 2.1 to 4.3 ms, while the Easy-pass preserves the jitter at the same level of the one without any flooding attacks. In general,

	w/o EZ-Pass		w/ EZ-Pass	
Flooding rate	1000	800	1000	800
300K	2.8ms	2.1ms	2.0μs	2.0μs
500K	3.6ms	2.4ms	2.3μs	2.2μs
800K	3.8ms	2.9ms	2.5μs	2.3μs
1M	4.2.ms	3.3ms	2.8μs	2.6μs
1.2M	4.5ms	3.6ms	3.4μs	3.1μs
1.5M	4.8ms	3.9ms	3.9μs	3.5μs
3M	5.2ms	4.3ms	4.3μs	3.9μs

Table 5.6: End-to-end delay-jitter for high-rate EF traffic

the experimental results of the high-rate case are similar to those of the low-rate case, demonstrating the effectiveness of Easy-pass against flooding attacks.

5.5.3 Overhead of Easy-pass

It is more important to characterize the overhead of Easy-pass introduced at the ISP edge router than that at an end-host, because the end-host can allocate more resources for each outgoing real-time packet than the edge router can. Moreover, the verification of Easy-pass incurs more CPU cycles than the Easy-pass embedding procedure at the end-host.

The per-packet overhead of Easy-pass, which is independent of packet size and the sending rate, is included in the processing overhead of an EF packet at the router. We measure the processing overhead for each EF packet at the router with and without the Easy-pass. In this set of experiments, we vary the reserved bandwidth from 100 Kbps to 2 Mbps, and the packet size from 100 to 1000 bytes. The network resources are well-provisioned and the EF traffic is well-behaved, so there is no queueing delay for the EF traffic. The measured per-packet processing time in the case of an embedded Easy-pass is 8 μs, whereas the same without the Easy-pass is 6 μs. The results are shown in Figure 5.10. The overhead incurred by the verification of Easy-pass is 2 μs. For the purpose of com-

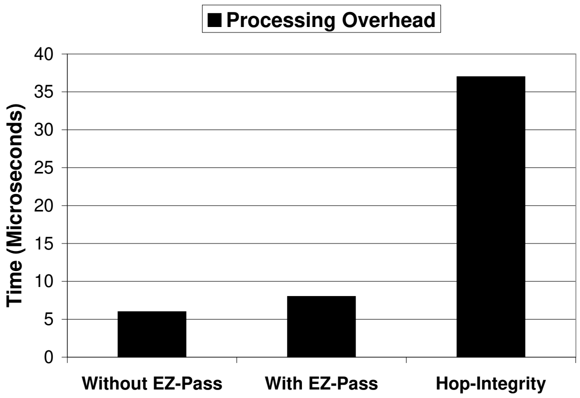


Figure 5.10: Processing overhead with and without Easy-Pass

parison, the authentication overhead of the hop-integrity mechanism [47], which is 37 μs, is also shown in the Figure 5.10. This value was derived in a separate study [47] using a slightly faster machine — a Pentium III 730 MHz running Linux OS. More importantly, the verification of Easy-pass is performed only once at the ISP edge router, not at every downstream router. Since the average end-to-end delay between two Internet-hosts ranges from tens to hundreds of milliseconds, the processing overhead of Easy-pass is negligible.

Each Easy-pass incurs the space overhead of 8 bytes. In the context of video traffic, in which the typical packet size is 800 or 1000 bytes, the space overhead of Easy-pass is 1% or 0.8%. In the context of audio traffic, the space overhead is increased to 3.1% or 1.5%, with respect to the 254 or 500 bytes long packets. Such an increase of space overhead is acceptable to most users. To further reduce the space overhead, we can trade security for performance by decreasing the length of Easy-pass from 64 to 32 bits.

5.6 Conclusion

In this chapter, we proposed a fast and lightweighted IP network-edge resource access control mechanism, called *IP Easy-pass*, to protect reserved network resources at edge devices from theft and abuse. By conducting experiments on a DiffServ testbed, we demonstrated the vulnerability of the reserved network resource to flooding attacks, and the need for IP-layer resource access control. In our scheme, a unique, encrypted, *pass* is attached to each legitimate real-time packet at the end-host. The ISP edge router validates the legitimacy of each incoming real-time packet simply by checking its pass. We described the creation of Easy-pass at the end-host, and its verification procedure at the ISP edge router using the RC5 encryption/decryption algorithm. The Easy-pass mechanism has been implemented as loadable Linux kernel modules.

We conducted a series of experiments on the DiffServ testbed to evaluate the effectiveness of Easy-pass against flooding attacks. The experimental results have shown that the Easy-pass mechanism effectively shields the reserved network resources from spoofed packets — it is shown to protect the legitimate packets from either loss or increased end-to-end delay-jitters for all the flooding rates we considered. Moreover, we measured the computational overhead of Easy-pass, and found it to be a negligible fraction (a few microseconds) of the processing time of each packet at both the end-host as well as at the edge router. Since the verification of Easy-pass is done at the ISP edge router only, not every downstream router, the Easy-pass overhead added to the end-to-end delay of an application traffic stream is negligible when compared to typical delay values of tens of milliseconds. Overall, IP Easy-pass is a very lightweight and effective mechanism for providing network-edge resource access control.

CHAPTER VI

CONCLUSIONS AND FUTURE DIRECTIONS

This dissertation has focused on countering distributed denial of service (DDoS) attacks — one of the hardest network security problems. Our proposed mechanisms demonstrated their protection for Internet services and network resources. We now give an overview of the main contributions of the dissertation and outline some related problems that warrant further investigation.

6.1 Research Contributions

_t_IP Router Architecture:

Based on the concept of layer-4 service differentiation and resource isolation, we developed a transport-aware IP router architecture that can be utilized as a built-in mechanism to counter DDoS attacks. The key components of the *t*IP router architecture are the fine-grained QoS classifier and the adaptive weight-based resource manager. The classifier divides the BAs (Behavior Aggregates) into thinner aggregates. Then, the adaptive resource manager provides fine-grained service differentiation and resource isolation for these thinner aggregates. Our extensive evaluation study has shown that the flooding traffic is significantly throttled and most of flooding packets are dropped in a close proximity to their sources.

SYN-dog:

TCP SYN flooding is the most common DDoS attack, which dominates in the available attacking tools and the number of DoS attacks known to date [75]. We proposed a simple and robust mechanism, called *SYN-dog*, to sniff SYN flooding attacks. The simplicity of SYN-dog lies in its statelessness and low computation overhead. SYN-dog utilizes the distinct protocol behavior of TCP connection establishment and teardown, which consists of SYN–FIN pair and SYN–SYN/ACK pair, for SYN flooding detection. Moreover, SYN-dog is insensitive to sites and access patterns: the non-parametric Cumulative Sum (CUSUM) method [19] is applied, making SYN-dog robust, much more generally applicable and its deployment easier. The efficacy of SYN-dog is evaluated and validated by trace-driven simulations. The evaluation results show that the SYN-dog achieves high detection accuracy and short detection time.

Hop-count Filter (HCF):

While an attacker can forge any field in the IP header of a spoofed packet, he cannot falsify the number of hops the packet takes to reach its destination, which is solely determined by the Internet routing infrastructure. Based on this observation, we presented a novel hop-count-based filter to weed out spoofed IP packets. We built an accurate IP-to-hop-count (IP2HC) mapping table, while using a moderate amount of storage, by clustering address prefixes based on hop-count. To capture hop-count changes under dynamic network conditions, we also devised a safe update procedure for the IP2HC mapping table that prevents pollution by HCF-aware attackers. We implemented HCF in the Linux kernel and evaluated the benefit of HCF with experimental measurements. Our experimental results have shown that HCF is indeed effective in countering IP spoofing by providing significant resource savings.

IP Easy-Pass:

To protect reserved network resources at edge devices from DQoS (Denial of Quality of Service) attacks, we proposed a fast and light-weighted IP network-edge resource access control mechanism, called *IP Easy-pass*. By conducting experiments on our DiffServ testbed, we demonstrated the vulnerability of the reserved network resources to flooding attacks. In our scheme, a unique, encrypted, *pass* is attached to each legitimate real-time packet at the end-host. The ISP edge router validates the legitimacy of each incoming real-time packet simply by checking its pass. The Easy-pass mechanism has been implemented as loadable Linux kernel modules. Our experimental results have shown that the Easy-pass mechanism effectively shields the reserved network resources from spoofed packets. Moreover, we measured the overhead of Easy-pass, and found it to be a negligible impact on end-to-end performance.

6.2 Future Directions

As the Internet has become indispensable to our daily lives, the need for fast and always-on Internet services is increasing. This section presents several issues that stem directly from this dissertation, and a few avenues of future research.

The non-parametric CUSUM methodology [19] employed by the SYN-dog can be easily extended to detect other types of flooding attacks, such as TCP ACK flooding and ICMP flooding attacks. Based on their distinct protocol behaviors, the ACK-dog or ICMP-dog can detect an ongoing flooding attack by observing the violation of normal protocol behaviors: for ACK-dog, the imbalance between TCP ACK packets and the corresponding TCP data packets in the reverse direction; for ICMP-dog, the imbalance between outgoing ICMP Echo requests and incoming ICMP Echo replies. Moreover, based on the CUSUM methodology, we can build a framework for traffic monitoring, in which the traffic monitor agents are installed at different ISP edge routers to detect abnormal traffic patterns between two adjacent ISPs.

There are several issues that warrant further research on HCF (Hop-Count Filtering). First, the existence of NAT (Network Address Translator) boxes, some of which may connect multiple stub networks, could make a single IP address appear to have multiple valid hop-counts at the same time. This may reduce our filtering accuracy. Second, to install the HCF system at a victim site, we need a systematic procedure for setting the parameters of HCF, such as the frequency of dynamic updates. Third, we would like to build and deploy HCF in various high-profile server sites to see how effective it is against real spoofed DDoS traffic.

Note that there is an arms-race between DDoS attacks and the corresponding defense mechanisms. The available DDoS attacking tools are constantly evolving, and new classes of DoS attacks are emerging with the newly-found security weaknesses, e.g., the low-bandwidth denial of service attacks that exploit algorithmic deficiencies in many common applications' data structures [30]. The emergence of new DDoS attacks makes it more challenging to build an effective counter measure. The latest DDoS attacks target root DNS servers and BGP routers [9], instead of high-profile Web servers. Since root DNS servers and BGP routers are the backbone of the Internet infrastructure, knocking off a root DNS server or compromising a BGP router will have much more devastating effects on the whole Internet.

Furthermore, a relatively homogeneous software base coupled with high-speed Internet connections provides an amenity of climate for the widespread of Internet worms. The increasing outbreaks of Internet worms pose an immense risk to the overall security of the Internet. As the most recent Sapphire/Slammer worm began spreading throughout the Internet [73], it doubled in size every 8.5 seconds. It infected more than 90 percent of vulnerable hosts within 10 minutes. Each infected machine is compromised, and could be used as a flooding source in a massive DDoS attacks later. So far, the Internet worms have been mostly nuisances, e.g., the analysis of the Sapphire/Slammer worm revealed no intent to harm its infected hosts [73]. However, in the future the Internet worms coupled with DDoS attacks will be more virulent, and thus, result in a chaos in the Internet. Currently, there is no effective mechanism to defend against the worm threat [74]. How to detect and contain such fast spread of Internet worms in real time, is an open issue.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] The ARQoS project. Available: http://arqos.csc.ncsu.edu/.

[2] The authenticated QoS project. Available: http://www.citi.umich.edu/projects/qos/.

[3] Caida's traffic workload overview. Available: http://www.caida.org/outreach/resources/learn/trafficworkload/tcpudp.xml.

[4] Netscreen 100 firewall appliance. Available: http://www.netscreen.com/.

[5] NLANR network traffic packet header traces. http://pma.nlanr.net/Traces/.

[6] VoIP bandwidth, white paper. Available: http://www.erlang.com/bandwidth.html/.

[7] Dave Andersen. tcptraceroute. Available: http://nms.lcs.mit.edu/software/ron/.

[8] Razor Team at Bindview. Despoof, 2000. Available: http://razor.bindview.com/tools/desc/despoof_readme.html.

[9] D. Atkins and R. Austein. Threat analysis of the domain name system. In *Internet Draft*, November 2002.

[10] H. Balakrishnan, V. Padmanabhan, and R. H. Katz. The effects of asymmetry on TCP performance. In *Proceedings of ACM/IEEE MOBICOM '97*, Budapest, Hungary, September 1997.

[11] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of USENIX OSDI'99*, New Orleans, LA, February 1999.

[12] R. Barbieri, D. Bruschi, and E. Rosti. Voice over IPsec: Analysis and solutions. In *Proceedings of 18th Annual Computer Security Applications Conference*, December 2002.

[13] M. Basseville and I. V. Nikiforov. *Detection of Abrupt Changes : Theory and Application*. Prentice Hall, 1993.

[14] S. M. Bellovin. ICMP traceback messages. In *Internet Draft: draft-bellovin-itrace-00.txt (work in progress)*, March 2000.

[15] D. J. Bernstein and Eric Schenk. Linux kernel SYN cookies firewall project. Available: http://www.bronzesoft.org/projects/scfw.

[16] S. Bhattacharyya, C. Diot, J. Jetcheva, and N. Taft. Pop-level and access-link-level traffic dynamic in a tier-1 pop. In *Proceedings of ACM Internet Measurement Workshop'2001*, San Francisco, CA, November 2001.

[17] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5), September/October 1999.

[18] L. Breslau, E. Knightly, S. Shenker, I. Stoica, and H. Zhang. Endpoint admission control: Architectural issues and performance. In *Proceedings of ACM SIGCOMM'2000*, Stockholm, Sweden, August 2000.

[19] B.E. Brodsky and B.S. Darkhovsky. *Nonparametric Methods in Change-point Problems*. Kluwer Academic Publishers, 1993.

[20] H. Burch and B. Cheswick. Mapping the internet. *IEEE Computer*, 32(4), 1999.

[21] H. Burch and B. Cheswick. Tracing anonymous packets to their approximate source. In *Proceedings of USENIX LISA'2000*, New Orleans, LA, December 2000.

[22] CERT Advisory CA-2000.01. Denial-of-service development, January 2000. Available: http://www.cert.org/advisories/CA-2000-01.html.

[23] CERT Advisory CA-96.21. TCP SYN flooding and IP spoofing, November 2000. Available: http://www.cert.org/advisories/CA-96-21.html.

[24] CERT Advisory CA-98.01. smurf IP denial-of-service attacks, January 1998. Available: http://www.cert.org/advisories/CA-98-01.html.

[25] R. Caceres, P. B. Danzig, S. Jamin, and D. J. Mitzel. Characteristics of wide-area TCP/IP conversations. In *Proceedings of ACM SIGCOMM '91*, Zurich, Switzerland, September 1991.

[26] B. Cheswick, H. Burch, and S. Branigan. Mapping and visualizing the internet. In *Proceedings of USENIX Annual Technical Conference '2000*, San Diego, CA, June 2000.

[27] K. Claffy, T. E. Monk, and D. McRobb. Internet tomography. In *Nature*, January 1999. Available: http://www.caida.org/Tools/Skitter/.

[28] W. S. Cleveland, D. Lin, and D. Sun. IP packet generation: statistical models for TCP start times based on connection-rate superposition. In *Proceedings of ACM SIGMETRICS '2000*, Santa Clara, CA, June 2000.

[29] E. Cronin, S. Jamin, C. Jin, T. Kurc, D. Raz, and Y. Shavitt. Constrained mirror placement on the internet. *IEEE Journal on Selected Areas in Communications*, 36(2), September 2002.

[30] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of USENIX Security Symposium'2003*, Washington D.C, August 2003.

[31] D. Dean, M. Franklin, and A. Stubblefield. An algebraic approach to IP traceback. *ACM Transactions on Information and System Security*, 5(2), May 2002.

[32] S. Dietrich, N. Long, and D. Dittrich. Analyzing distributed denial of service tools: The shaft case. In *Proceedings of USENIX LISA'2000*, New Orleans, LA, December 2000.

[33] D. Dittrich. Distributed Denial of Service (DDoS) attacks/tools page. Available: http://staff.washington.edu/dittrich/misc/ddos/.

[34] The Swiss Education and Research Network. Default TTL values in TCP/IP, 2002. Available: http://secfr.nerim.net/docs/fingerprint/en/ttl_default.html.

[35] M. El-Gendy, A. Bose, H. Wang, and K. G. Shin. Statistical characterization for per-hop QoS. In *Proceedings of IWQoS'2003*, Monterey, CA, June 2003.

[36] B. Davie *et al.* An expedited forwarding PHB (per-hop behavior). In *RFC 3246*, March 2002.

[37] S. Blake *et al.* An architecture for differentiated services. In *RFC 2475*, December 1998.

[38] Y. Bernet *et al.* A framework for differentiated services. In *IETF Internet Draft*, February 1999.

[39] A. Feldmann. Characteristics of TCP connection arrivals. In *ATT Technical Report*, December 1998.

[40] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. In *RFC 2267*, January 1998.

[41] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), August 1995.

[42] National Laboratory for Applied Network Research. Active measurement project (amp), 1998-. Available: http://watt.nlanr.net/.

[43] M. Fullmer and S. Romig. The OSU flow-tools package and cisco netflow logs, December 2000.

[44] L. Garber. Denial-of-service attack rip the internet. *IEEE Computer*, April 2000.

[45] S. Gibson. Distributed reflection denial of service. In *Technical Report, Gibson Research Corporation*, February 2002. Available: http://grc.com/dos/drdos.htm.

[46] T. M. Gil and M. Poletter. MULTOPS: a data-structure for bandwidth attack detection. In *Proceedings of USENIX Security Symposium'2001*, Washington D.C, August 2001.

[47] M. G. Gouda, E. N. Elnozahy, C.-T. Huang, and T. M. McGuire. Hop integrity in computer networks. *IEEE/ACM Transactions on Networking*, 10(3), June 2002.

[48] R. Govinda and H. Tangmunarunkit. Heuristics for internet map discovery. In *Proceedings of IEEE INFOCOM '2000*, Tel Aviv, Israel, March 2000.

[49] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proceedings of ACM SIGCOMM '99*, Cambridge, MA, September 1999.

[50] G. Hadjichristofi, N. Davis IV, and C. Midkiff. IPsec overhead in wireline and wireless networks for web and email applications. In *Proceedings of IEEE IPCCC '2003*, Phoenix, AZ, April 2003.

[51] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of USENIX Security Symposium'2001*, Washington D.C, August 2001.

[52] U. Hengartner, S. Moon, R. Mortier, and C. Diot. Detection and analysis of routing loops in packet traces. In *Proceedings of ACM Internet Measurement Workshop'2002*, Marseille, France, November 2002.

[53] G. Iannaccone, C.-N. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an IP backbone. In *Proceedings of ACM Internet Measurement Workshop'2002*, Marseille, France, November 2002.

[54] Arbor Networks Inc. Peakflow DoS, 2002. Available: http://arbornetworks.com/standard?tid=34&cid=14.

[55] J. Ioannidis and S. M. Bellovin. Implementing pushback: Router-based defense against DDoS attacks. In *Proceedings of NDSS'2002*, San Diego, CA, February 2002.

[56] S. Jamin, P. Danzig, S. Shenker, and L. Zhang. A measurement-based admission control algorithm for integrated services packet networks. *IEEE/ACM Transactions on Networking*, 5(1), February 1997.

[57] C. Jin, H. Wang, and K. G. Shin. Hop-count filtering: An effective defense against spoofed DDoS traffic. In *Proceedings of ACM CCS '2003*, October 2003.

[58] F. Kelly, P. Key, and S. Zachary. Distributed admission control. *IEEE Journal on Selected Areas in Communications*, 18(12), December 2000.

[59] S. Kent and R. Atkinson. Security architecture for the internet prtocotl. In *RFC 2401*, November 1998.

[60] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proceedings of ACM SIGCOMM '2002*, Pittsburgh, PA, August 2002.

[61] T.V. Lakshman and D. Stiliadis. High speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of ACM SIGCOMM '98*, Vancouver, Canada, September 1998.

[62] J. Lemon. Resisting SYN flooding DoS attacks with a SYN cache. In *Proceedings of USENIX BSDCon'2002*, San Francisco, CA, February 2002.

[63] J. Li, J. Mirkovic, M. Wang, P. Reiher, and L. Zhang. SAVE: Source address validity enforcement protocol. In *Proceedings of IEEE INFOCOM '2002*, New York City, NY, June 2002.

[64] M. Li, M. Claypool, and B. Kinicki. MediaPlayer versus RealPlayer — a comparison of network turbulence. In *Proceedings of ACM Internet Measurement Workshop'2002*, Marseille, France, November 2002.

[65] Check Point Software Technologies Ltd. Syndefender. Available: http://www.checkpoint.com/products/firewall-1.

[66] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *ACM Computer Communication Review*, 32(3), July 2002.

[67] J. Martin and A. Nilsson. On service level agreements for ip networks. In *Proceedings of IEEE INFOCOM '2002*, New York City, NY, June 2002.

[68] S. McCreary and K. Claffy. Trends in wide area IP traffic patterns — a view from ames internet exchange. In *Proceedings of ITC '2000*, September 2000.

[69] J. McQuillan. Layer 4 switching. *Data Communications*, October 1997.

[70] A. Mena and J. Heidemann. An empirical study of real audio traffic. In *Proceedings of IEEE INFOCOM '2000*, Tel Aviv, Israel, March 2000.

[71] S. Miltchev, S. Ioannidis, and A. D. Keromytis. A study of the relative costs of network security protocols. In *Proceedings of the USENIX Annual Technical Conference'2002 Freenix Track*, Monterey, CA, June 2002.

[72] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. In *Proceedings of IEEE ICNP'2002*, Paris, France, November 2002.

[73] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the sapphire/slammer worm. In *CAIDA, Tech. Report*, February 2003.

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

[75] D. Moore, G. Voelker, and S. Savage. Inferring internet denial of service activity. In *Proceedings of USENIX Security Symposium'2001*, Washington D.C., August 2001.

[76] Robert T. Morris. A weakness in the 4.2bsd unix TCP/IP software. In *Computing Science Technical Report 117, AT&T Bell Laboratories*, Murray Hill, NJ, February 1985.

[77] S. Murphy. DiffServ additions to ns-2, May 2000. Available: http://www.teltec.duc.ie/ murphys/ns-work/diffserv.

[78] Mazu Networks. Enforcer, 2002. [Online]. Available: http://www.mazunetworks.com/products/.

[79] P. G. Neumann and P. A. Porras. Experience with EMERALD to DATE. In *Proceedings of 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, April 1999.

[80] K. Nichols, V. Jacobson, , and L. Zhang. A two-bit differentiated services architecture for the internet. In *RFC 2638*, July 1999.

[81] R. Oliver. Countering SYN flood denial-of-service attacks. In *Tech Mavens, Inc*, August 2001. http://www.tech-mavens.com/synflood.htm.

[82] K. Papagiannaki, P. Thiran, J. Crowcroft, and C. Diot. Preferential treatment of acknowledgment packets in a differentiated services network. In *Proceedings of IWQoS '2001*, Karlsruhe, Germany, June 2001.

[83] K. Park and H. Lee. On the effectiveness of probabilistic packet marking for IP traceback under denial of service attack. In *Proceedings of IEEE INFOCOM '2001*, Anchorage, Alaska, March 2001.

[84] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets. In *Proceedings of ACM SIGCOMM '2001*, San Diego, CA, August 2001.

[85] V. Paxson. End-to-end routing behavior in the internet. *IEEE/ACM Transactions on Networking*, 5(5), October 1997.

[86] V. Paxson. An analysis of using reflectors for distributed Denial-of-Service Attacks. *ACM Computer Communication Review*, 31(3), July 2001.

[87] V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3), June 1995.

[88] A. Perrig, R. Canetti, J. D. Tygar, and D. Song. Efficient authentication and signing of multicast streams over lossy channels. In *Proceedings of IEEE Symposium on Security and Privacy'2000*, May 2000.

[89] M. Poletto. Practical approaches to dealing with ddos attacks. In *NANOG 22 Agenda*, May 2001. Available: http://www.nanog.org/mtg-0105/poletto.html.

[90] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. In *Proceedings of USENIX OSDI'2002*, Boston, MA, December 2002.

[91] K.K. Ramakrishnan and S. Floyd. A proposal to add explicit congestion notification (ECN) to IP. In *RFC 2481*, January 1999.

[92] J. Reumann, H. Jamjoom, and K. G. Shin. Adaptive packet filters. In *Proceedings of IEEE Globcom '2001*, San Antonio, TX, November 2001.

[93] R. L. Rivest. The RC5 encryption algorithm. *Lecture Notes in Computer Science*, 1008, Springer-Verlag, 1995.

[94] D. Rizzetto and C. Catania. A voice over IP service architecture for integrated communications. *IEEE Network*, 13(3), June 1999.

[95] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proceedings of ACM SIGCOMM '2000*, Stockholm, Sweden, August 2000.

[96] C. L. Schuba, I. V. Krsul, M. G. Kuhn, E. H. Spafford, A. Sundaram, and D. Zamboni. Analysis of a denial of service attack on TCP. In *Proceedings of IEEE Symposium on Security and Privacy'97*, May 1997.

[97] C. Shannon, D. Moore, and K. C. Claffy. Beyond folklore: observations on fragmented traffic. *IEEE/ACM Transactions on Networking*, 10(6), December 2002.

[98] F. D. Smith, F. H. Campos, K. Jeffay, and D. Ott. What TCP/IP protocol header can tell us about the web. In *Proceedings of ACM SIGMETRICS '2001*, Cambridge, MA, June 2001.

[99] A. C. Snoren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based IP traceback. In *Proceedings of ACM SIGCOMM '2001*, San Diego, CA, August 2001.

[100] D. Song and A. Perrig. Advanced and authenticated marking schemes for IP traceback. In *Proceedings of IEEE INFOCOM '2001*, Anchorage, Alaska, March 2001.

[101] O. Spatscheck and L. Peterson. Defending against denial of service attacks in Scout. In *Proceedings of USENIX OSDI'99*, New Orleans, LA, February 1999.

[102] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with rocketfuel. In *Proceedings of ACM SIGCOMM '2002*, Pittsburgh, PA, August 2002.

[103] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of ACM SIGCOMM '98*, Vancouver, Canada, September 1998.

[104] W. R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley Publishing Company, 1994.

[105] R. Stone. Centertrack: An IP overlay network for tracking DoS floods. In *Proceedings of USENIX Security Symposium'2000*, Denver, CO, August 2000.

[106] M. Sung and J. Xu. IP traceback-based intelligent packet filtering: A novel technique for defending against internet DDoS attacks. In *Proceedings of of IEEE ICNP '2002*, Paris, France, November 2002.

[107] S. Templeton and K. Levitt. Detecting spoofed packets. In *Proceedings of The Third DARPA Information Survivability Conference and Exposition (DISCEX III)'2003*, Washington, D.C., April 2003.

[108] R. Thayer, N. Doraswamy, and R. Glenn. IP security document roadmap. In *RFC 2411*, November 1998.

[109] K. Thompson, G. J. Miller, and R. Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network*, 11(6), November/December 1997.

[110] H. Tschofenig and D. Kroeselberg. Security threats for NSIS. In *Internet Draft, draft-ietf-nsis-threats-01.txt*, January 2003.

[111] UCB/LBNL/VINT. Network simulator. In *ns-2, http://www.isi.edu/nsnam/ns/*, 1999.

[112] H. Wang, C. Shen, and K. G. Shin. Adaptive-weighted packet scheduling for premium service. In *Proceedings of IEEE International Conference on Communications'2001*, Helsinki, Finland, June 2001.

[113] H. Wang and K. G. Shin. Robust TCP congestion recovery. In *Proceedings of IEEE ICDCS'2001*, Phoenix, AZ, April 2001.

[114] H. Wang and K. G. Shin. Layer-4 service differentiation and resource isolation. In *Proceedings of IEEE RTAS'2002*, San Jose, CA, September 2002.

[115] H. Wang and K. G. Shin. Transport-aware IP routers: A built-in protection mechanism to counter DDoS attacks. *IEEE Transactions on Parallel and Distributed Systems*, 14(9), September 2003.

[116] H. Wang, D. Zhang, and K. G. Shin. Detecting SYN flooding attacks. In *Proceedings of IEEE INFOCOM '2002*, New York City, NY, June 2002.

[117] P. Wang, Y. Yemini, D. Florissi, J. Zinky, and P. Florissi. Experimental QoS performances of multimedia applications. In *Proceedings of IEEE INFOCOM '2000*, Tel Aviv, Israel, March 2000.

[118] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley Publishing Company, 1994.

[119] A. Yaar, A. Perrig, and D. Song. Pi: A path identification mechanism to defend against DDoS attacks. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003.

[120] D. Yau, J. Lui, and F. Liang. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. In *Proceedings of IWQoS'2002*, Miami Beach, FL, May 2002.

[121] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, 7(5), September 1993.

[122] Y. Zhang and B. Singh. A multi-layer IPsec protocol. In *Proceedings of 9th USENIX Security Symposium*, Denver, Colorado, August 2000.

ABSTRACT

COUNTERING DISTRIBUTED DENIAL OF SERVICE ATTACKS

by

Haining Wang

Chair: Kang G. Shin

The growing number of Distributed Denial of Service (DDoS) attacks impose a significant threat to the availability of Internet services. This dissertation examines the working mechanism of DDoS attacks and develops a number of methodologies and prototypes to counter DDoS attacks. These mechanisms defend against DDoS attacks in different ways: throttling DDoS traffic at IP routers (core or edge), sniffing flooding sources at edge routers, filtering out DDoS traffic at victim severs or their nearby firewalls, and protecting reserved network resources at edge routers.

Based on the concept of layer-4 service differentiation and resource isolation, we propose a transport-aware IP router architecture, in which the flooding traffic is significantly throttled and most of the traffic is dropped in a close proximity to their sources. To sniff SYN flooding attacks, we propose a simple and robust mechanism, called *SYN-dog*. The core of SYN-dog is based on the distinct protocol behavior of TCP connection establishment and teardown, and is an instance of the Sequential Change Point Detection [13]. A non-parametric Cumulative Sum (CUSUM) method [19] is applied, thus making the

SYN-dog insensitive to site and access pattern.

We develop a novel hop-count-based filter to weed out spoofed IP packets at victim sites. *Hop-Count Filtering* (HCF) builds an accurate IP-to-hop-count (IP2HC) mapping table, while using a moderate amount of storage, by clustering address prefixes based on hop-count. To capture hop-count changes under dynamic network conditions, we also devise a safe update procedure for the IP2HC mapping table that prevents pollution by HCF-aware attackers.

Finally, to protect reserved network resources at edge devices from DQoS (Denial of Quality of Service) attacks, we propose a fast and light-weighted IP network-edge resource access control mechanism, called *IP Easy-pass*. we demonstrate the vulnerability of the reserved network resources to flooding attacks. Then, we attach a unique *pass* to each legitimate real-time packet so that an ISP edge router can validate the legitimacy of an incoming IP packet very quickly and simply by checking its pass. Thus, *Easy-pass* shields the reserved network resources from spoofed packets.