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

Cheng Jin Haining Wang Kang G. Shin
chengjin@cs.caltech.edu, {hxw,kgshin}@eecs.umich.edu

Abstract

IP spoofing has been exploited by Distributed Denial of Service (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 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. 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. Base 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 Introduction

An Internet host can spoof IP packets by using a raw socket to fill arbitrary source IP addresses into their IP headers [28]. IP spoofing is usually associated with malicious network behaviors, such as Distributed Denial of Service (DDoS) attacks. As one of the most difficult problems in network security, DDoS attacks have posed a serious threat to the availability of Internet services [6, 18, 27]. Instead of subverting services, DDoS attacks limit and block legitimate users' access by exhausting victim servers' resources [7], or saturating stub networks' access links to the Internet [19]. 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 [12, 13]. Moreover, some known DDoS attacks, such as smurf [8] and more recent DRDoS (Distributed Reflection Denial of Service) attacks [19, 33], 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.

To thwart DDoS attacks, researchers have taken two distinct approaches: *router-based* and *victim-based*. The router-based approach makes improvements to the routing infrastructure, while the victim-based approach enhances the resilience of Internet servers against attacks. The router-based approach performs either off-line analysis of flooding traffic or on-line filtering of DDoS traffic inside routers. Off-line IP traceback [4, 36, 37, 38, 41] 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 filtering mechanisms rely on IP router enhancements [15, 23, 24, 25, 26, 31] to detect abnormal traffic patterns and foil DDoS attacks. However, these solutions require not only router support, but also coordination among different routers and networks, and widespread deployment.

Compared to the router-based approach, the victim-based approach has the advantage of being immediately deployable. More importantly, a potential victim has a much stronger incentive to deploy defense mechanisms than network service providers. The current victim-based approach protects Internet servers using sophisticated resource management schemes. These schemes provide more accurate resource accounting, and fine-grained service isolation and differentiation [3, 5, 35, 39], 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. 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. Since filtering spoofed IP packets is orthogonal to resource management, it can be used in conjunction with advanced resource-management schemes.

Therefore, 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 paper, 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 [17, 20, 30, 43] 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.

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

The remainder of the paper is organized as follows. Section 2 presents the TTL-based hop-count computation and the hop-count inspection algorithm, which is in the critical path of HCF. Section 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 demonstrates the effectiveness of the proposed filter in detecting spoofed packets. Section 5 deals with the construction of IP2HC mapping table, the heart of HCF. Section 6 details the two running states of HCF, the inter-state transitions, and the placement of HCF. Section 7 presents the implementation and experimental evaluation of HCF. Section 8 discusses related work. The paper concludes with Section 9.

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.

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 [14], 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 [9, 10], 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,

```
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 ≠ H_s)
           the packet is spoofed;
    else
           the packet is legitimate;
```

Figure 1: Hop-Count inspection algorithm.

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 [14] 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.

2.2 Inspection Algorithm

Assuming that an accurate IP2HC mapping table is present (see Section 5 for details of its construction) Figure 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-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 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 paper, the terms zombie and fboding source are used interchangeably.

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.

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 [32], 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 [16] 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 5.2.

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 1: Diversity of `traceroute` gateway locations.

To obtain actual hop-count distributions, we use the raw `traceroute` data from 50 different `traceroute` gateways in [11]. 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 1. Figure 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.

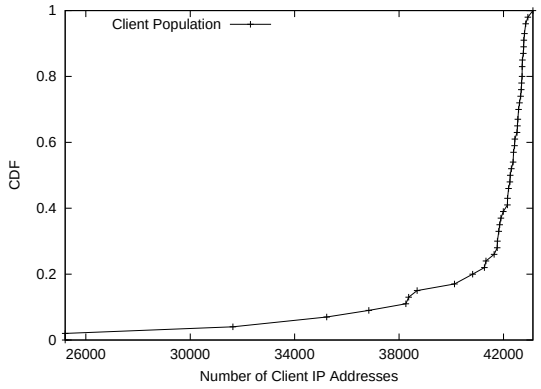

Figure 2: CDF of size of client IP addresses.

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 3–6 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 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 Figures 7 and 8, respectively. We observe that most of the mean values fall between 14 and 19 hops, and the 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.

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

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 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 [12, 13] 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*

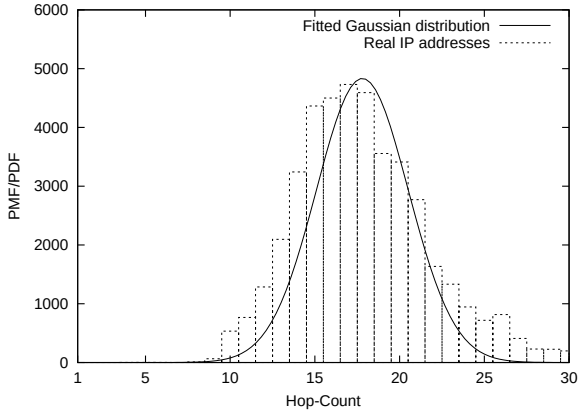

Figure 3: Yahoo.

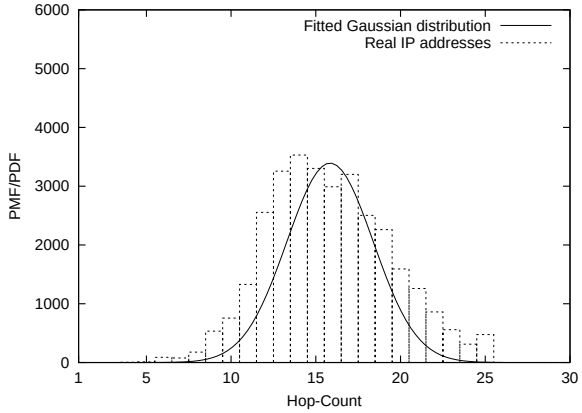

Figure 4: Stanford University.

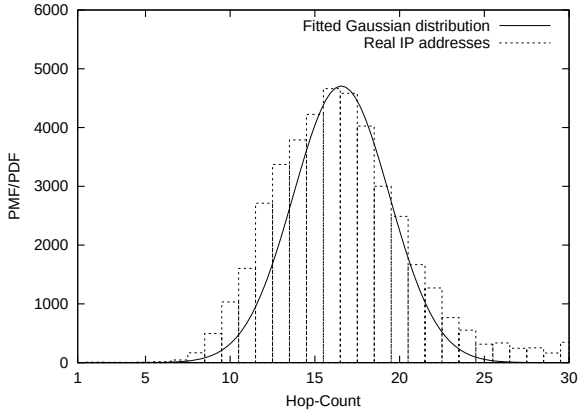

Figure 5: `cpcug.org`.

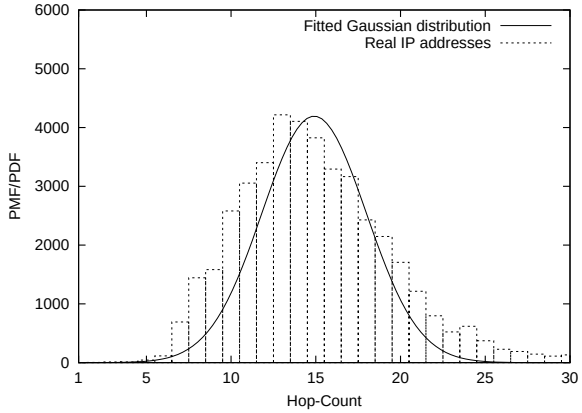

Figure 6: `fenice.it` in Italy.

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 [44]; (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 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 [9], Mercator [21], Rocketfuel [40], and Skitter [10] 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, inter-domain 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-

[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.

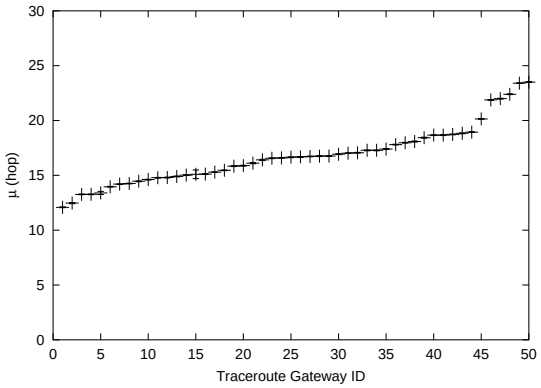

Figure 7: μ parameters for traceroute gateways.

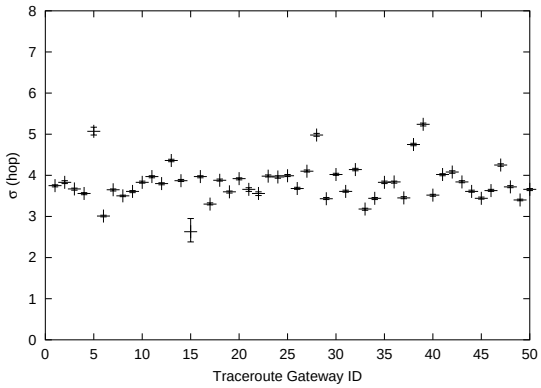

Figure 8: σ parameters for traceroute gateways.

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 [12, 13]. The wide-spread use of randomness in spoofing IP address has been verified by the "backscatter" study [27], which quantified DoS activities in the Internet.

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. 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.

4.1 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 [12, 13] 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.

4.1.1 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 9 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 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%

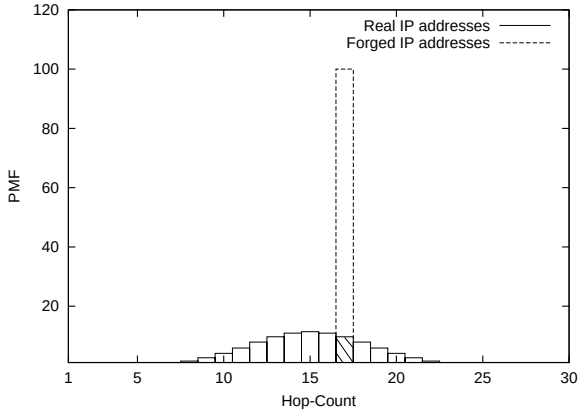

Figure 9: Hop-Count distribution of IP addresses with a single flooding source.

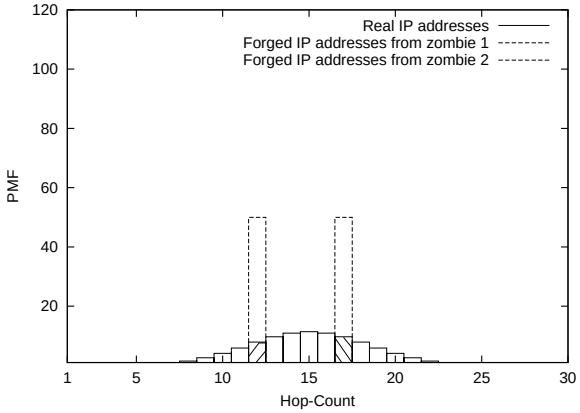

Figure 10: Hop-Count distribution of IP addresses with two flooding sources.

and 8%, respectively. Overall, HCF is very effective against these simple attacks, reducing the attack traffic by one order of magnitude.

4.1.2 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 10 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.

4.2 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. 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 11 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 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

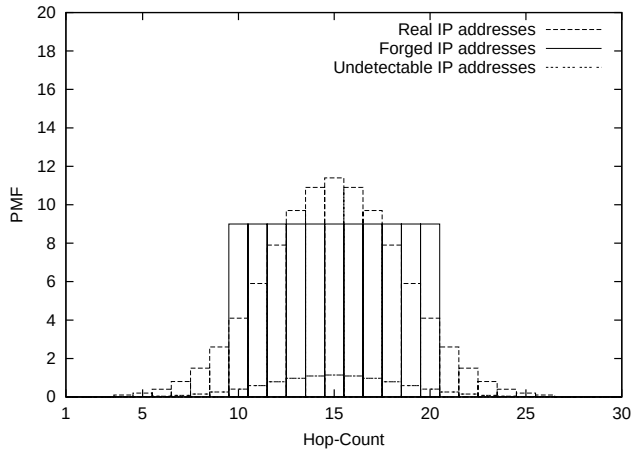

Figure 11: Hop-Count distribution of IP addresses with a single flooding source, randomized TTL values.

Figure 11, 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 11. 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 11, 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 4.1.1.

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.

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 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

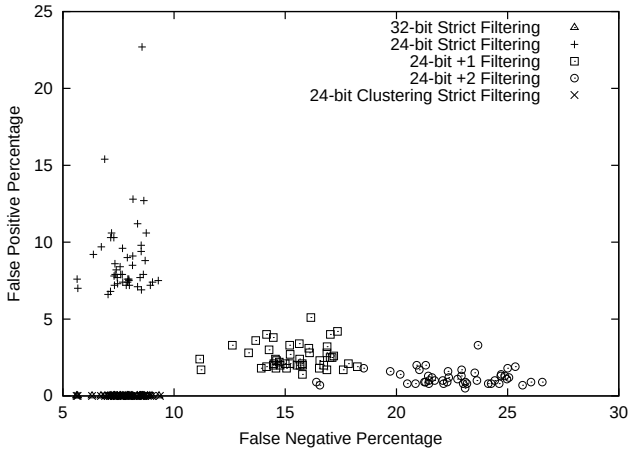

Figure 12: Accuracies of various filters. (Note that the points of 24-bit clustering filtering overlap with those of 32-bit filtering.)

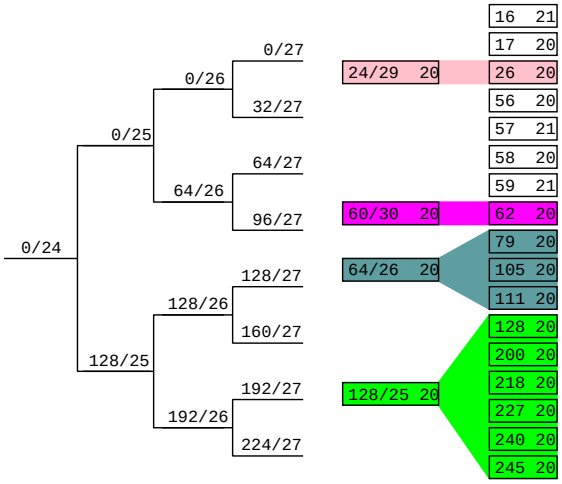

Figure 13: An example of hop-count clustering.

of client IP addresses. We compute the percentage of false negatives according to the calculation in Section 4.2.

5.1.1 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 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 4.2 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 12 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.

5.1.2 Aggregation with Hop-Count Clustering

Under 24-bit aggregation, the percentage of false negatives is still high (≈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 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

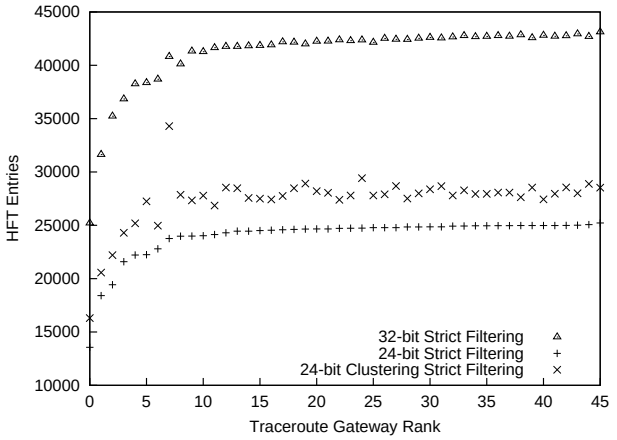

Figure 14: Sizes of various HCF tables.

a given set of IP addresses. Figure 13 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 are known as in Figure 12. 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 12 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 14 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%).

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 [44]. 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 [32] and confirmed by our own observations, the overhead incurred by re-clustering is negligible.

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.

6.1 Tasks in Two States

Figure 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:

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

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.

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.

```
In alert state:
  for each sampled packet p:
    spoof = IP2HC_Inspect(p);
    t = Average(spoof);
    if ( spoof )
            if ( t > T1 )
                    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 ≤ T2 )
            Switch_Alert();
```

Figure 15: Operations in two HCF states.

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 net-

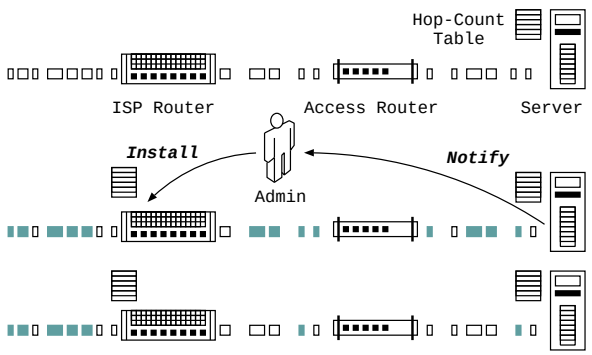

Figure 16: Packet filtering at a router to protect bandwidth.

work's ISP.

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 16 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.

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.

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 [11] 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.

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 ex-

	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` fbod	358	256	20139	3616
TCP bulk	443	168	6538	3700
UDP bulk	490	184	6524	3628

Table 2: CPU overhead of HCF and normal IP processing.

amine 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 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` [1] 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 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 2.

We present the recorded processing times in CPU cycles in Table 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

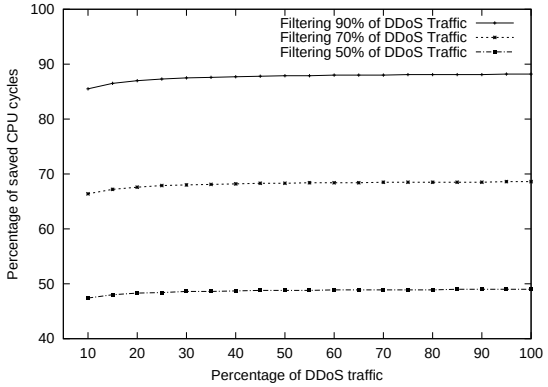

Figure 17: Resource savings by HCF.

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 17. 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 $(1-\alpha)$.

8 Related Work

Researchers have used the distribution of TTL values seen at servers to detect abnormal load spikes due to DDoS traffic [34]. The Razor team at Bindview built Despoof [2], 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 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 [42]. 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 paper, we examine both questions and develop a deployable solution.

There are a number of recent router-based filtering techniques to lessen the effects of DDoS packets or to curb their propagations in the Internet. As a proactive solution to DDoS attacks, these filtering schemes [15, 25, 31, 45], which must execute on IP routers or rely on routers' markings, have been proposed to prevent spoofed IP packets from reaching intended victims. The most straightforward scheme is *ingress filtering* [15], which blocks spoofed packets at edge routers, where address ownership is relatively unambiguous, and traffic load is low. 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 benefits to their customers.

Given the reachability constraints imposed by routing and network topology, route-based distributed packet filtering (DPF) [31] utilizes routing information to determine whether an incoming packet at a router is valid with respect to the packet's inscribed source and destination IP addresses. The experimental results reported in [31] show that a significant fraction of spoofed packets may be filtered out, and those spoofed packets that DPF fails to capture, can be localized into five candidate sites which are easy to trace back.

To validate that an IP packet carries the true source address, SAVE [25], 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 whether 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.

Based on IP traceback marking, Path Identifier (Pi) [45] 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 half of the routers in the Internet participate in packet marking.

There already exist commercial solutions [22, 29] 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.

9 Conclusion and Future Work

In this paper, we present 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 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 show 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.

There are several issues that warrant further research. 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 for practical use, we need a systematic procedure for setting

the parameters of HCF, such as the frequency of dynamic updates. Finally, 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.

References

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

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

[3] 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.

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

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

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

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

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

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

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

[17] M. Fullmer and S. Romig. The osu flow-tools package and cisco netflow logs. In *Proceedings of USENIX LISA'2000*, New Orleans, LA, December 2000.

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

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

[20] 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.

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

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

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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

[33] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *ACM Computer Communication Review*, 31(3), July 2001.

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

[35] 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.

[36] 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.

[37] 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.

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

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

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

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

[42] 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.

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

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

[45] 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.