Hop-Count Filtering: An Effective Defense Against Spoofed 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) attackers to (1) conceal flooding sources and localities of flooding traffic, and (2) coax uncompromised 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 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. Based on this observation, we propose a novel filtering technique for Internet servers to winnow away spoofed IP packets. By clustering address prefixes based on hop-counts, *Hop-Count Filtering* (HCF) builds an accurate IP to hop-count (IP2HC) mapping table to detect and discard spoofed IP packets. Through analysis using network measurement data, we show that HCF can identify and then discard close to 90% of spoofed IP packets with little collateral damage. We implement and evaluate the HCF in the Linux kernel, demonstrating its benefits with experimental measurements.

1 Introduction

DDoS attacks pose a serious threat to the availability of Internet services [4, 15, 25]. Instead of subverting services, DDoS attacks limit and block legitimate users' access by exhausting victim servers' resources [5], or saturating stub networks' access links to the Internet [16]. To conceal flooding sources and localities of flooding traffic, attackers spoof IP addresses by randomizing the 32-bit source-address field in the IP header [10, 11]. In addition, some known DDoS attacks, such as smurf [6] and more recent DRDoS (Distributed Reflection Denial of Service) attacks [16, 30], are not possible without IP spoofing. They masquerade the source IP address of each spoofed packet with the victim's IP address. The Internet is vulnerable to IP spoofing because of the statelessness of IP protocol and destination-based routing. The IP protocol lacks the control to prevent a sender from hiding its packets' origin. Moreover, destination-based routing does not maintain state information on senders, and delivers each IP packet to its destination without authenticating the packet's source IP address. Overall, IP spoofing [26] makes DDoS attacks much harder to detect and counter.

To thwart DDoS attacks, researchers have taken two distinct approaches. The first approach improves the routing infrastructure, while the second approach enhances the resilience of Internet servers against attacks. The first approach performs either off-line analysis of flooding traffic traces or on-line filtering of spoofed IP packets inside routers. Off-line IP traceback [2, 34, 36, 39] 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 [13, 20, 21, 23, 24, 29] to detect abnormal traffic patterns and foil DDoS attacks. However, these solutions require not only router support, but also coordination among different routers and wide-spread deployment.

The end-system approach protects Internet servers with sophisticated resource management to servers. This approach provides more accurate resource accounting, and fine-grained service isolation and differentiation [1, 3, 32, 37], for example, to shield interactive video traffic from FTP traffic. However, without a mechanism to detect 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 particular service under attack, it cannot sustain the availability of that 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 also be used in conjunction with advanced resource-management schemes.

As discussed earlier, end-system-based filtering that does not require router support is necessary to detect and discard spoofed traffic. 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 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 it 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 winnow away 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 legitimate IP packets from the sources that have been spoofed. *Hop-Count Filtering* (HCF) builds an accurate IP2HC (IP to hop-count) 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.

Two running states, *alert* and *action*, within HCF use this mapping to inspect the IP header of each IP packet. Under normal condition, HCF resides in *alert* state, watching for abnormal TTL behaviors without discarding packets. Upon detection of an attack, HCF switches to *action* state, in which the HCF discards those IP packets with mismatching hop-counts. Besides the IP2HC inspection, several efficient mechanisms [14, 17, 28, 40] are available to detect DDoS attacks. Through analysis using network measurement data, we show that the HCF can recognize close to 90% of spoofed IP packets. Then, since our hop-count-based clustering significantly reduces the percentage of false positives,[1] we can discard spoofed IP packets with little collateral damage. To ensure that the filtering mechanism itself withstands attacks, our design is light-weight and requires only a moderate amount of storage. We implement a test module of HCF in the Linux kernel, at the network layer as the first step of IP-packet processing. Then, we evaluate its benefits with real experiments and show that HCF is indeed effective in countering IP spoofing while producing significant resource savings.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the hop-count inspection algorithm including hop-count computation, which is in the critical path of HCF. Section 4 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 5 demonstrates the effectiveness of the proposed filter in detecting and discarding spoofed packets. Section 6 deals with the construction of IP2HC mapping table, the heart of HCF. Section 7 details the two running states of HCF, the inter-state transitions, and the placement of HCF. Section 8 presents the implementation and experimental evaluation of HCF. The paper concludes with Section 9.

2 Related Work

Researchers have used the distribution of TTL values seen at servers to detect abnormal spikes due to DDoS traffic [31]. However, we are not aware of any scheme that computes hop-counts of incoming packets to filter spoofed DDoS traffic at or near victims. In this section, we highlight some of the recent filtering techniques on lessening the effects of DDoS packets and their propagation in the Internet.

As a proactive solution to DDoS attacks, several filtering schemes [13, 23, 29], which must execute on IP routers, have been proposed to prevent spoofed IP packets from reaching intended victims. The most straightforward scheme is ingress filtering [13], 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 benefit to their customers.

Given the reachability constraints imposed by routing and network topology, route-based distributed packet filtering (DPF) [29] 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 [29] show that a significant fraction of spoofed packets may be filtered out, and those spoofed packets that the 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 [23], 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 verifies whether an IP packet arrives at its expected interface. By matching incoming IP addresses with their expected receiving interfaces, the set of IP source addresses that any attacker can spoof is greatly reduced.

There already exist commercial solutions [19, 27] 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-based mechanism that does not require **any** network support.

3 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, then present the inspection algorithm in detail.

3.1 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 of the IP header. TTL is an 8-bit field in the IP header, originally introduced to specify the maximum lifetime of IP packets in the Internet. During transit, each intermediate router decrements the TTL value of an IP packet by one before forwarding it to the next-hop router. The final TTL value when a packet

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

```
for each packet:
    extract the final TTL $T$ and IP address $S$;
    infer the initial TTL $T_o$;
    compute the hop-count $H_c = T - T_o$;
    index $S$ to get the stored hop-count $H_s$;
    if   $(H_c \neq H_s)$
        packet is spoofed;
    else
        packet is legitimate;
```

Figure 1: Hop-Count inspection algorithm.

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. It would have been simple if all operating systems (OSs) use the same initial TTL, but in practice, there is no consensus on the initial TTL value. Furthermore, Since the OS for a given IP address may change at any time, we cannot assume a single static initial TTL value for each IP address.

Fortunately, however, most modern OSs use only a few selected initial TTL values, 30, 32, 60, 64, 128, and 255, according to [12]. 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 these initial TTL values differ from each other by more than 30, except between 30 and 32, and between 60 and 64. Since it is generally believed that few Internet hosts are apart by more than 30 hops, which is also confirmed by our own observations, 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 and 32, and 60 and 64, we will compute a hop-count value for each of the two possible initial TTL values, and accept the packet if either hop-count matches.

The drawback of limiting the possible initial TTL values is that end-systems that use "odd" initial TTL values, may be incorrectly identified as having spoofing source IP addresses. This may happen if a user switches OS from one that uses a well-known initial TTL value, to one that uses an "odd" value. Note, however, that our filter starts to discard packets only upon detection of a DDoS attack, so such end-systems would suffer only during an actual DDoS attack. The study in [12] shows that the OSs that use "odd" initial TTLs are typically older OSs. We believe that they 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.

3.2 Inspection Algorithm

Assuming that an accurate IP2HC mapping table is present (details of its construction in Section 6), Figure 3.1 outlines the procedure HCF uses 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 it from the final TTL value to obtain the hop-count. Then, 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 classified as spoofed. 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 every spoofed packet. However, we will show in Section 5 that even with a limited range of hop-count values, HCF can be highly effective at identifying spoofed IP addresses.

4 Feasibility of Hop-Count Filtering

Recent Internet experiments [22, 33] have shown that, despite the large number of routing updates, (1) a large fraction of destination prefixes have remarkably stable Border Gateway Protocol (BGP) routes, (2) popular prefixes tend to have stable BGP routes for days or weeks; and (3) a vast majority of BGP instability stems from a small number of unpopular destinations. Moreover, a recent case study of intra-domain routing behavior [35] indicates that the intra-domain topology changes are due mainly to external changes and no network-wide instability is observed. Therefore, it is reasonable to expect hop-counts to be generally stable in the Internet. Moreover, the proposed filter contains a dynamic update procedure to capture hop-count changes as shown in Section 6.2.

In this section, we investigate whether matching the hop-count with the source IP address of each packet suffices to recognize spoofed packets. The valid hop-counts to a server must be diverse enough such that the largest percentage of IP addresses that have a common hop-count value is small. We examine the hop-count distributions of valid IP addresses to 47 Internet Web servers and observe that hop-counts among these IP addresses are indeed diverse enough to support the use of HCF. Furthermore, we show that even an HCF-aware attacker cannot circumvent filtering easily.

4.1 Hop-Count Distribution

The key to effective HCF is that the hop-count distribution of client IP addresses at a server take a range of values. Since HCF cannot recognize forged packets whose source IP addresses have the same hop-count value as that of an attacker, it is important to examine hop-count distributions at various locations in the Internet to ensure that hop-count distributions

[2]In this paper, zombie and fboding source are interchangeable terms.

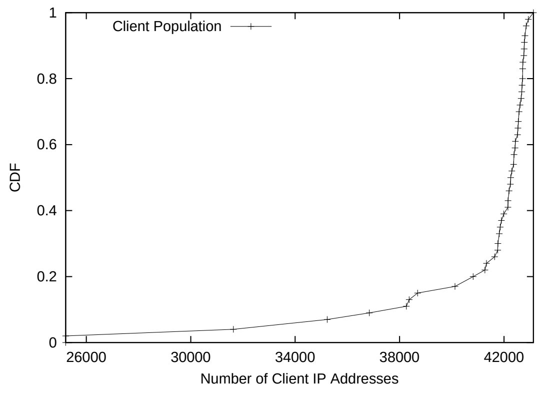

Figure 2: CDF of size of client IP addresses.

are not clustered around a single value. If 90% of client IP addresses are ten hops away from a server, one would not be able to distinguish spoofed packets from legitimate ones using hop-count filtering alone.

To obtain real hop-count distributions, we use the raw `traceroute` data from 50 different `traceroute` gateways in [9]. 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 to more than 40,000 clients.

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.

We examined the hop-count distribution plots of 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:

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

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

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 whether 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-count for each data set. We plot the mean values 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 are generally between 3 and 4 hops. Such distributions allow HCF to work effectively as we will show in our quantitative evaluation of HCF in Section 5.

4.2 Robustness against Evasion

HCF relies on the fact that spoofed IP packets often have mismatching IP addresses and hop-counts to effectively block spoofed packets. Once attackers learn of this technique, they will try to generate spoofed packets with matching source IP addresses and hop-counts. In this subsection, we evaluate whether attackers can easily evade the HCF by constructing "seemingly-legitimate" IP packets.

To evade HCF, an attacker has to 'manufacture' an appropriate initial TTL value for each spoofed packet. Suppose the hop-count from a flooding source to the victim is h_z, and the hop-count from a chosen spoofed IP address to the victim is h_s. Assuming that both the flooding source and the chosen IP address use the same initial TTL value I, the final TTL value of a packet from the flooding source to the victim is $I - h_z$, and the final TTL value of a legitimate packet from the chosen IP address would be $I - h_s$. To evade HCF, the attacker must change the initial TTL value of the spoofed packet to $I' = I - (h_s - h_z)$ so the spoofed packet would have the correct hop-count when it reaches the victim.

An attacker can know h_z, the hop-count from a zombie site to the victim by running `traceroute`. However, due to random selection of the source address of each spoofed IP packet [10, 11], it is extremely difficult, if not impossible, for the attacker to figure out h_s, the hop-count between a randomized IP address and the victim. To figure out h_s in real time, 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. The reason for this is obvious: the attacker cannot observe the final TTL values of normal traffic at the victim, although he can overload the victim. For the attacker to build such an IP2HC mapping table, it must compromise at least one end-host behind any stub network whose IP address is in the random IP address space, and perform `traceroute` to get h_s for the corresponding IP2HC mapping entry. Without such knowledge, even if the attacker knows h_z, he cannot fabricate

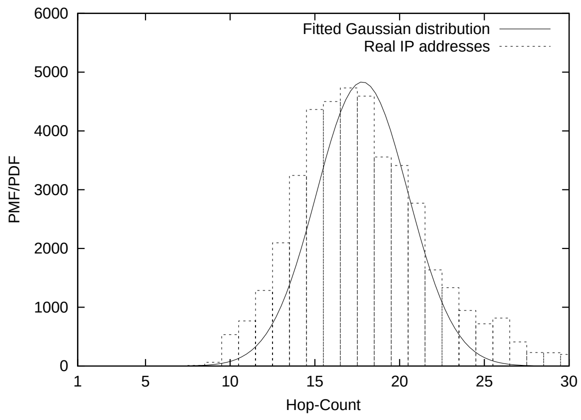

Figure 3: Yahoo.

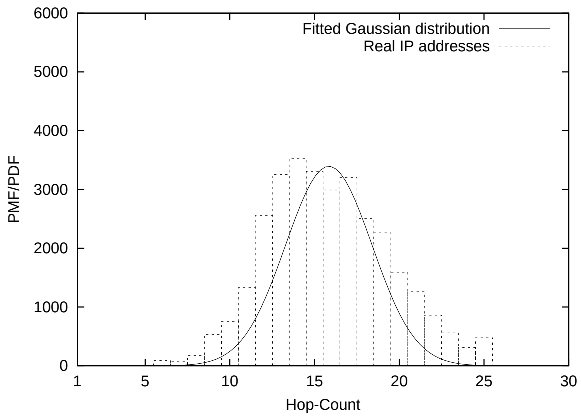

Figure 4: Stanford University.

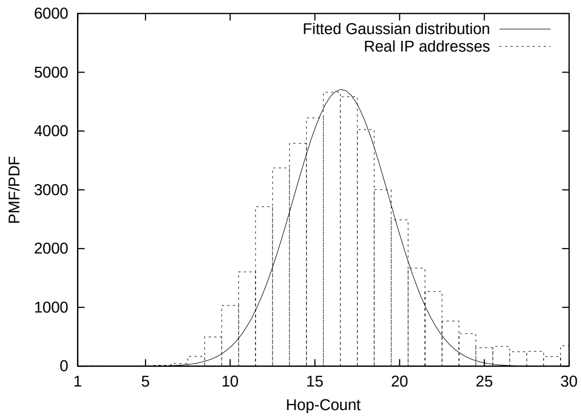

Figure 5: `cpcug.org`.

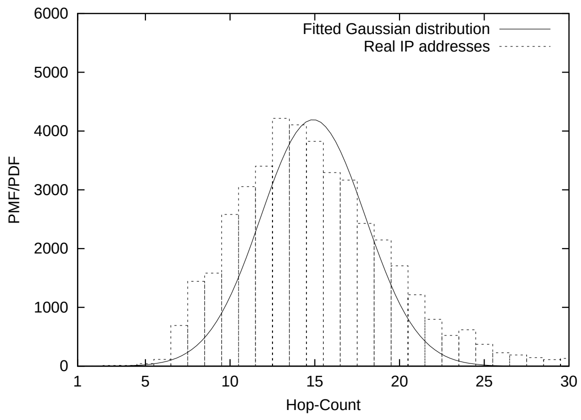

Figure 6: `fenice.it` in Italy.

the desired TTL values to conceal the forgery.

An alternative way to find the IP address to hop-count mapping, without compromising end-hosts, is to figure out the locations of IP addresses, an accurate router-level topology of the Internet, and the underlying routing algorithms and policies. The recent Internet mapping efforts such as Internet Map [7], Mercator [18], Rocketfuel [38], and Skitter [8] projects, may make the approach plausible. However, the current topology mappings put together snap-shots of portions of networks measured at different times. Topology maps thusproduced are generally time-averaged approximation of network connectivity. Moreover, none of topology mappings has the details of a stub network like a campus network, and the access router coverage ranges from fair to poor. Most importantly, inter-domain routing in the Internet is policy-based, and the routing policies are not disclosed to the outside world. Even if an attacker has accurate information on geographic and topology mappings, they cannot obtain hop-count information based on network connectivity alone. The path, and therefore the hop-count, between a source and a destination is determined by routing policies and algorithms that are often unknown. In summary, it is practically impossible to get accurate hop-count information by taking this alternative approach.

Instead of spoofing randomly selecting IP addresses, an attacker may choose to spoof IP addresses from a set of already compromised machines, whose number would be much smaller than 2^{32}, so that he can measure all h_s's and forge the correct hop-counts. However, this weakens the attacker's DDoS attacks in several ways. First, the list of would-be spoofed source IP addresses is greatly reduced,i which makes the detection and blockage 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 probe the victim server somehow to obtain the correct hop-counts. However, we find that network administrators nowadays are extremely alert to unusual access patterns or probing attempts so it would require great effort in coordinating the probing attempts such that it does not 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 [10, 11]. The wide-spread usage of randomness in spoofing IP address has been verified by the "backscatter" study [25], which quanti-

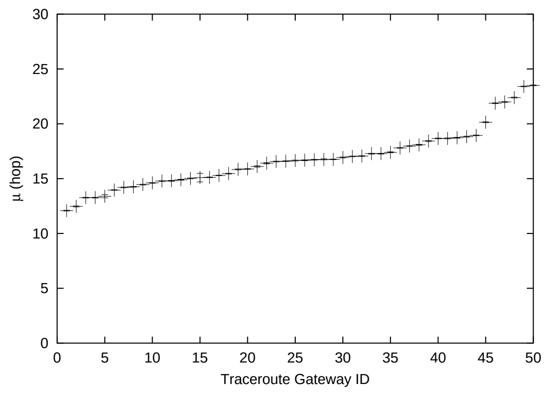

Figure 7: μ parameters for traceroute gateways.

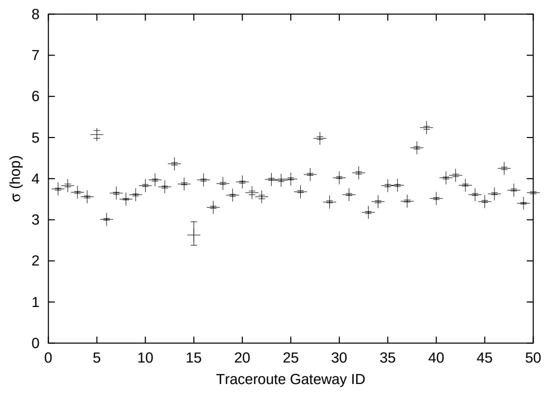

Figure 8: σ parameters for traceroute gateways.

fied DoS activities in the Internet.

5 Effectiveness of HCF

We now assess the effectiveness of HCF from a mathematical standpoint. More specifically, we address the question "what fraction of spoofed IP packets can be detected by the proposed scheme?" We assume potential DDoS victims know the complete mapping between their client IP addresses and hop-counts, and we discuss the construction of such mappings in the next section. We assume that to spoof packets, attacker randomly select source IP addresses from the entire IP address space, and select hop-counts according to some distribution. Without loss of generality, we further assume that attackers evenly divide 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-count values. We will later discuss an update procedure that will capture legitimate hop-count changes.

5.1 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 [10, 11] 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.

5.1.1 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. This observation also follows the Central Limit Theorem. 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 the 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 4.1, even if the attacker floods spoofed IP packets from such a zombie, the 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.

5.1.2 Multiple Sources

DoS attacks usually involve more than a single host, and hence, we need to examine the case of multiple active flooding sources. When there are n sources that flood a total of F packets, each flooding source has a unique hop-count to the victim server, and 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 the flooding source, then the percentage of spoofed packets that the HCF can identify

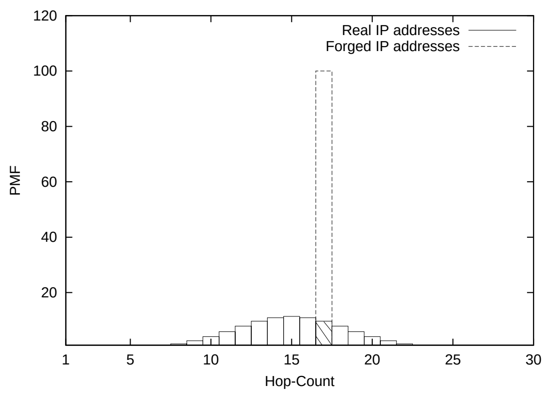

Figure 9: Hop-Count distribution of IP address with a single flooding source.

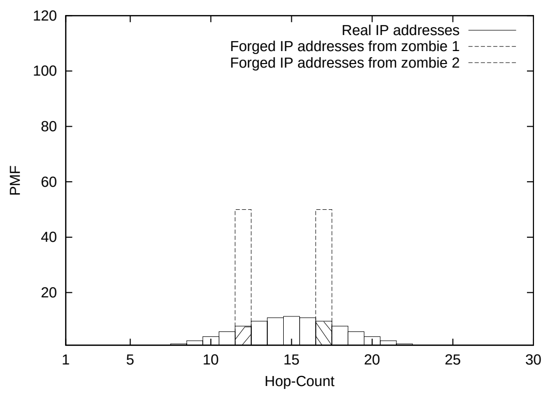

Figure 10: Hop-Count distribution of IP address with two flooding sources.

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.

5.2 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 4.1) 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 the 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 $i\le j$ 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, 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 the HCF.

In general, an attacker could generate initial TTLs within the range $[h_m,h_n]$, based on some known distribution, where the fraction 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 the HCF is:

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

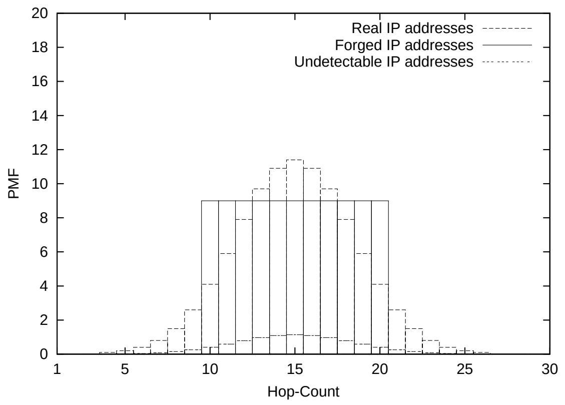

Figure 11: Hop-Count distribution of IP addresses with a single flooding source, randomized TTL values.

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 5.1.1.

6 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 HCF tables. 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.

6.1 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 4.1) 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 the 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 5.2.

6.1.1 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

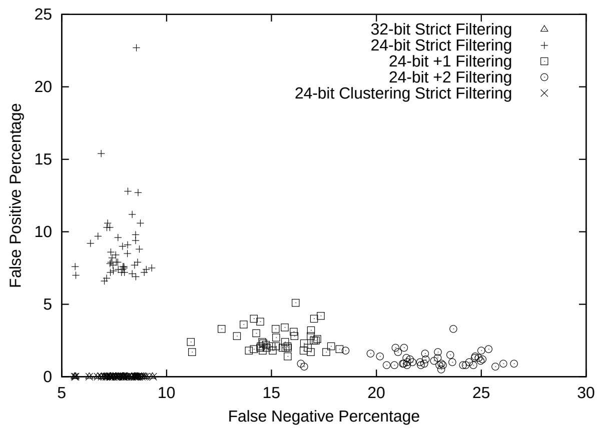

Figure 12: Accuracies of various filters. (Note that the points of 24-bit clustering filtering overlap with those of 32-bit filtering.)

hop-counts differ by greater than two.

We have shown in Section 5.2 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.

6.1.2 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 the leaf nodes, 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 we are able to aggregate them, the parent node will have the same hop-count as the children. We will be able to find the largest possible aggregation for each IP address. Figure 13 shows an example of aggregating a list of IP addresses (with the last octets shown) upward the aggregation tree (showing the first four levels). We can aggregate 11 of 13 IP addresses into four aggregated network prefixes. The remaining IP addresses cannot be aggregated using this scheme.

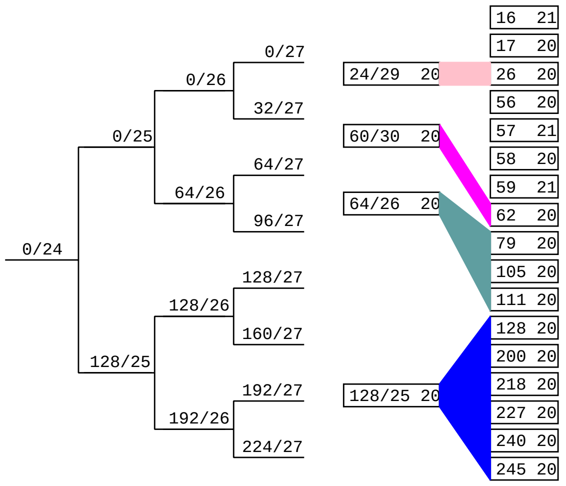

Figure 13: Example of hop-count clustering.

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. The 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

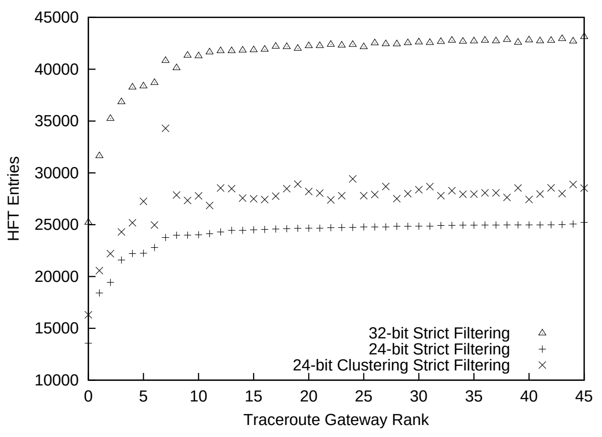

Figure 14: Sizes of various HCF tables.

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%).

6.2 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 [41]. 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.

7 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.

7.1 Tasks in Two States

Figure 7.1 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, the HCF enters the action state. The HCF in alert state will also update the HCF table using the TCP control block of every k-th established TCP connection.

The HCF in action state performs per-packet hop-count inspection and discards spoofed packets, if any. The 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. The HCF stays in action state as long as spoofed IP packets are detected. When the ongoing spoofing ceases, the 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

```
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 ( $t \leq T_2$ )
        Switch_Alert();
```

Figure 15: Operations in two HCF 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, the 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.

7.2 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, the HCF specifically looks for IP spoofing, so it will be able to detect attempts to fool servers into acting as reflectors. Although the HCF is not perfect and some spoofed packets may still slip through the filter, the HCF can detect and intercept enough of the spoofed packets to thwart DRDoS attacks.

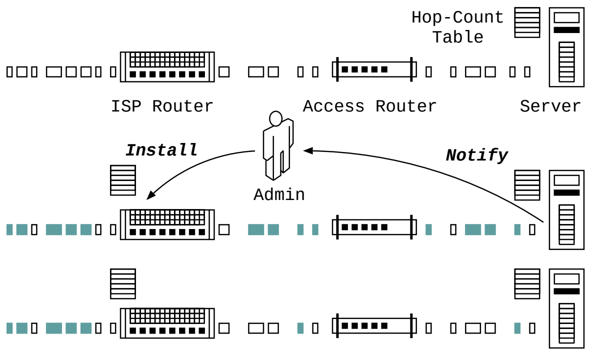

Figure 16: HCF at routers to protect bandwidth.

7.3 Blocking Bandwidth Attacks

To protect server resources such as CPU and memory, the 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.

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 the 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 the 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 its access router and the ISP's edge router. Under the normal network condition, the web server monitors its traffic and builds the HCF table. When the attack traffic arrives at the stub network, the web server or the access router will notice a sudden rise of spoofed traffic. Either of the two will inform the network administrator via an authenticated channel. The administrator can have the ISP install a packet filter using the HCF table in the ISP's edge router. Note that one cannot directly use the HCF table since the hop-counts (distance) 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 update 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.

8 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. Our measurement shows that HCF indeed makes significant resource savings.

8.1 Building the Hop-Count Filter

We implement a test module inside the Linux 2.4.18-3 kernel to examine the per-packet overhead of HCF and the processing time of normal IP packets. To minimize the CPU cycles consumed by spoofed IP packets, the HCF checks the legitimacy of the source IP address of each packet before doing more expensive checksum verification. We locate the IP layer receiving function, `ip_rcv`, and insert the filtering function before the code that verifies the checksum of the IP header. Our test module implements the basic filtering data structure and operations.

The HCF table is based on 24-bit address aggregation discussed in Section 6.1.1, both as a linear lookup table and as a 4096-entry hash table with chaining to resolve collisions. Under the lookup-table implementation, the filtering function determines the initial TTL value of an incoming packet, computes the actual hop-count, and then uses the first 24 bits of the source IP address as the index to retrieve the correct hop-count for comparison. A linear array offers fast lookup and hence, is suitable for popular servers that receive requests from many different client IP addresses. A hash table is more appropriate for servers that have a relatively small number of clients. In the hash table implementation, we compute the hash key by XORing the upper and lower 12-bits of the first 24 bits of each source IP address and search. Given the 24-bit network address space, 4096 is relatively small so collisions are likely to occur. To optimize for performance, entries on each chained list could be arranged according to client access frequencies. To estimate the impact of collision, we hash the client IP addresses from [9] into the 4096-entry hash table to find that, on average, there are 11 entries on a chain, with the maximum being 25.

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 the first-time visitor.

8.2 Experimental Evaluation

For HCF to be useful, the overhead must be much lower than the normal processing of IP packets. 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 packets. We use the built-in Linux macro `rdtscl` to record the execution time in CPU cycles.

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, is the victim server where our filter is installed. A second machine generates various types of IP traffic to emulate the incoming traffic to the victim server. Instead of emulating DDoS attacks, we generate two types of traffic, TCP and ICMP, to emulate the flooding traffic used in DDoS attacks. For TCP flooding, we repeatedly open a TCP connection on the victim machine and close it right away, which includes sending both SYN and FIN packets. However, we only time the `open` system call to emulate SYN flooding. Linux delays the processing of three-way handshake until the final ACK from the active sender is received. Since we include this in our measurement, the measured overhead may be larger than in an actual SYN flooding attack. To emulate ICMP attacks, we ran three experiments of single-stream `ping`s. The first uses default 64-byte packets, and the second uses 1500-byte Ethernet 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 56 bytes and sends packets as fast as the system can transmit.

To understand the HCF's impact on normal IP traffic, we also consider bulk data transfers under both TCP and UDP. We then examine the per-packet overhead without HCF, and the per-packet overhead of the filtering function in per-packet inspection.

To make our tests more realistic under the hash-table implementation, we randomize each hash key to simulate randomized IP addresses in order to hit all possible entries in the

	Lookup		Hash		without HCF	
scenarios	avg	min	avg	min	avg	min
TCP `open`	218	88	1266	136	36342	3700
`ping` 64B	91	88	411	128	20194	3604
`ping` 1500B	89	88	288	108	35925	2436
`ping` fbod	88	88	339	172	20139	3616
TCP bulk	91	88	444	124	6538	3700
UDP bulk	91	88	424	144	6524	3628

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

table. For each hash-table search, we manually traverse a list of 11 entries to emulate the behavior observed from actual measurement. Although we are not able to draw definitive conclusions on the HCF's performance, we can confirm that the HCF makes significant resource savings.

We present the recorded processing times in Table 2. The columns under 'Lookup' and 'Hash' list the execution times of the filtering function under the two implementations. 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 in order to compute average cycles. We present both the minimum and the average number of cycles consumed for each experiment. Note that there is 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 include more lower-level interrupt processing, such as receiving an Ethernet frame, than minimum cycles. We observe that, in general, the filtering functions use much fewer cycles than the emulated attacking traffic, often at least one order of magnitude less. Clearly, the HCF can make significant resource savings by detecting and discarding spoofed traffic. In case of bulk transfers, the differences are also significant. 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 the savings from not processing spoofed traffic.

Table 2 also shows us that the lookup table offers excellent performance with extremely small per-packet processing overhead, while the hash table efficiently utilizes storage space and achieves reasonably good performance.

To illustrate the potential savings in CPU cycles, we would like to know how much CPU resource savings we can achieve, while an attacker launches a spoofed DDoS attack against a web server. Assuming the division between attack and legitimate traffic is aX and bX, with X being the total number of packets per unit time. The CPU cycles consumed without HCF is: $aX \cdot t_D + bX \cdot t_L$, where t_D and t_L are the per-packet processing time of attack and legitimate traffic, respectively. The CPU cycles consumed when the HCF is present is:

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

with t_{DF} and t_{LF} being the filtering overhead for spoofed and legitimate traffic, respectively, and α the percent of spoofed traffic that we cannot filter. Let's assume that the attacker uses 56-byte `ping` traffic to attack the web server that implements the HCF as a hash table. 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 DDoS attacks, or a. The y-axis is the resource savings as the percentage of CPU cycles consumed without HCF. The figures shows a number of plots, each corresponding to an α value. Since the per-packet overhead of DDoS traffic (20194) is much higher than TCP bulk transfer (6538), the percentage of DDoS traffic 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 $(1-\alpha)$.

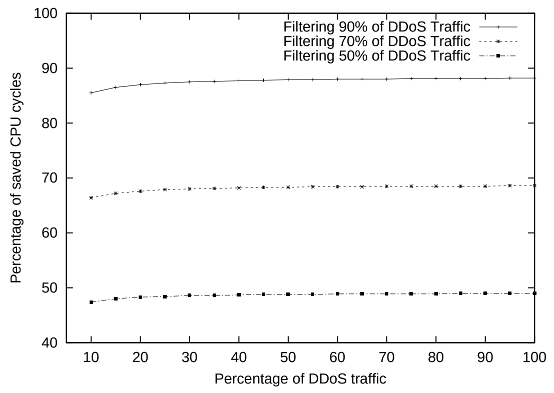

Figure 17: Resource savings by HCF.

9 Conclusion and Future Work

In this paper we presented a hop-count-based filtering scheme that detects and then removes spoofed IP packets to save system resources. Based on the analysis using actual network measurements, we showed that HCF can remove about 90% of spoofed traffic. Moreover, even if an attacker is aware of HCF, he cannot easily circumvent HCF. Our experimental evaluation demonstrates that HCF can be efficiently implemented inside the Linux kernel.

Our analysis and experimental results indicate 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 network support.

There are several issues that warrant further investigation. For example, we need a systematic procedure for setting the parameters of HCF, such as the frequency of dynamic updates. We also need to study the effectiveness of HCF against replicated actual attacks from real attacking traces. These and others are matters of our future inquiry.

References

[1] 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.

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

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

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

[5] CERT Advisory CA-96.21. TCP SYN fbod-ing and IP spoofi ng, November 2000. Available: http://www.cert.org/advisories/CA-96-21.html.

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

[7] 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.

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

[9] 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.

[10] 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.

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

[12] The Swiss Education and Research Network. Default TTL values in TCP/IP, 2002. Available: http://secfr.nerim.net/docs/fi ngerprint/en/ttl default.html.

[13] P. Ferguson and D. Senie. Network ingress fi ltering: Defeating denial of service attacks which employ IP source address spoofi ng. In *RFC 2267*, January 1998.

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

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

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

[17] 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.

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

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

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

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

[22] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental study of internet stability and backbone failures. In *Proccedings of the 29th International Symposium on Fault-Tolerant Computing*, Madison, WI, June 1999.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

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

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

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

[32] 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.

[33] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP routing stability of popular destinations. In *Proceedings of ACM Internet Measurement Workshop'2002*, Marseille, France, November 2002.

[34] 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.

[35] A. Shaikh, C. Isett, A. Greenberg, M. Roughan, and J. Gottlieb. A case study of ospf behavior in a large enterprise network. In *Proceedings of ACM Internet Measurement Workshop'2002*, Marseille, France, November 2002.

[36] 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.

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

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

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

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

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