Defense Against Spoofed IP Traffic Using Hop-Count Filtering

Haining Wang Cheng Jin Kang G. Shin

Abstract— **IP spoofing has often been exploited by Distributed Denial of Service (DDoS) attacks to (1) conceal flooding sources and dilute 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 victim servers is essential to their own protection and prevention 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. More importantly, since the hop-count values are diverse, an attacker cannot *randomly* spoof IP addresses while maintaining consistent hop-counts. On the other hand, an Internet server can easily infer the hop-count information from the Time-to-Live (TTL) field of the IP header. Using a mapping between IP addresses and their hop-counts, the server can distinguish spoofed IP packets from legitimate ones. Based on this observation, we present a novel filtering technique, called *Hop-Count Filtering* (HCF)—which builds an accurate IP-to-hop-count (IP2HC) mapping table—to detect and discard spoofed IP packets. HCF is easy to deploy, as it does not require any support from the underlying network. Through analysis using network measurement data, we show that 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 effectiveness with experimental measurements.**

Keywords: IP spoofing, DDoS attacks, Hop-count, Host-based.

I. Introduction

IP networks are vulnerable to source address spoofing [33]. For example, a compromised Internet host can spoof IP packets by using a raw socket to fill arbitrary source IP addresses into packet headers. IP spoofing is commonly associated with malicious network activities, such as Distributed Denial of Service (DDoS) attacks [21], [27], [32], which block legitimate access by either exhausting victim servers' resources [7] or saturating stub networks' access links to the Internet [18]. Most DDoS attacking tools spoof IP addresses by randomizing the 32-bit source-address field in the IP header [12], [13], which conceals attacking sources and dilutes localities in attacking traffic. The recent "backscatter" study [32], which quantifies DoS activities in the current Internet, has confirmed the wide-spread use of randomness in spoofing IP addresses. Moreover, some known DDoS attacks, such as smurf [8] and more recent DRDoS (Distributed Reflection Denial of Service) attacks [18], [38], are not possible without IP spoofing. Such attacks masquerade the source IP address of each spoofed packet with the victim's IP address. Overall, DDoS attacks with IP spoofing are much more difficult to defend.

To thwart DDoS attacks, researchers have taken two distinct approaches: *router-based* and *host-based*. The router-based approach installs defense mechanisms inside IP routers to trace the source(s) of attack [4], [30], [43], [45], [46], [49], or detect and block attacking traffic [15], [23], [25], [29], [31], [36], [50], [56]. However, these router-based solutions require not only router support, but also coordination among different routers and networks, and wide-spread deployment to reach their potential. In contrast to the router-based approach, the host-based approach can be deployed immediately. Moreover, end systems should have a much stronger incentive to deploy defense mechanisms than network service providers.

The current host-based approaches protect an Internet server either by using sophisticated resource-management schemes [3], [6], [40], [47] or by significantly reducing the resource consumption of each request to withstand the flooding traffic such as SYN cookies [5] and Client Puzzle [24], [53]. Without a mechanism to detect and discard spoofed IP traffic at the very beginning of network processing, spoofed packets will share the same resource principals and code paths as legitimate requests. Under heavy attacks, current approaches are unlikely to be able to sustain service availability due to resource depletion caused by spoofed IP packets. Furthermore, most of existing host-based solutions work at the transport-layer and above, and cannot prevent the victim server from consuming CPU resource in servicing interrupts from spoofed IP traffic. At high speed, incoming IP packets generate many interrupts and can drastically slow down the victim server [42] (also see Section VI). Therefore, the ability to detect and filter spoofed packets at the IP layer without any router support is essential to protection against DDoS attacks. Since filtering spoofed IP packets is orthogonal to the resource-protection mechanisms at higher layers, it can be used in conjunction with advanced resource-protection schemes.

In this paper, we propose a lightweight scheme that validates incoming IP packets at an Internet server without using any cryptographic methodology or router support. Our goal is not to achieve perfect authentication, but to screen out most bogus traffic with little collateral damage. The fundamental idea is to utilize inherent network information—that each packet carries and an attacker cannot easily forge—to distinguish spoofed packets from legitimate ones. The inherent network information we use here is the number of hops a packet takes to reach its destination: 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 a packet to the next hop.

Based on hop-count, we propose a novel filtering technique, called *Hop-Count Filtering* (HCF), to weed out spoofed IP packets at the very beginning of network processing, thus effectively protecting victim servers' resources from abuse. The rationale behind HCF is that most randomly-spoofed IP packets, when arriving at victims, do not carry hop-count values that are consistent with the IP addresses being spoofed. As a receiver, an Internet server can infer the hop-count information and check for consistency of source IP addresses. Exploiting this observation,

The work reported in this paper was supported in part by the NSF under Grants CCR-0329629 & CNS-052392 and by the ONR under Grant No. N00014-04-10726. This paper was previously presented in part at ACM CCS'2003. The authors were with the Department of Electrical Engineering and Computer Science, the University of Michigan, Ann Arbor, MI 48109-2122. Haining Wang (hnw@cs.wm.edu) is now with College of William and Mary, Cheng Jin (chengjin@cs.caltech.edu) with Caltech, and Kang G. Shin (kgshin@umich.edu) with the University of Michigan.

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 attackers. The same pollution-proof method is used for both initializing IP2HC mapping table and inserting additional IP addresses into the table.

To minimize collateral damage, HCF has two running states, *learning* and *filtering*. Under normal conditions, HCF stays in the *learning* state, watching for abnormal TTL behaviors without discarding any packets. Even if a legitimate packet is incorrectly identified as spoofed, it will not be dropped. Therefore, there is *no* collateral damage in the *learning* state. Upon detection of an attack, HCF switches to the *filtering* state, in which HCF discards those IP packets with mismatching hop-counts. 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 number of false positives.[1] Thus, we can discard spoofed IP packets with little collateral damage in the *filtering* state. To ensure that the filtering mechanism itself withstands attacks, our design is lightweight and requires only a moderate amount of storage. We implement HCF in the Linux kernel as the first step of incoming packet processing at the IP layer. 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.

While HCF is simple and effective in thwarting IP spoofing, it is not a complete solution to the generic DDoS problem. Rather, it is only an important piece of the puzzle that weeds out spoofed IP traffic. Like most other schemes in dealing with the DDoS problem, HCF has its own limitations. An attacker may circumvent HCF entirely by not using spoofed traffic, or partially by bombarding a victim with much more attacking traffic than seen before. Also, a "determined" attacker may find a way to build an IP2HC mapping table that is accurate enough for most spoofed IP packets to evade HCF. Moreover, the actual deployment of HCF requires further work in tuning its parameters and handling the IP2HC inaccuracy caused by the Network Address Translator (NAT) boxes and possible hop-count instability. Nevertheless, HCF does greatly enhance the capability to counter DDoS attacks by depriving an attacker of his powerful weapon, random IP spoofing.

The remainder of the paper is organized as follows. Section II presents the TTL-based hop-count computation, the pollution-proof hop-count capturing mechanism, and the hop-count inspection algorithm, which are critical to HCF. Section III demonstrates that the proposed HCF indeed works effectively in detecting spoofed packets, based on a large set of previously-collected `traceroute` data, and also robust against HCF-aware attackers. Section IV presents the construction of the IP2HC mapping table. Section V details the two running states of HCF, the inter-state transitions, and the placement of HCF. Section VI describes our implementation and experimental evaluation of HCF. Section VII discusses related work. Finally, the paper concludes with Section VIII.

[1]Those legitimate packets that are incorrectly identified as spoofed.

II. Basic Principles in HCF

In this section, we describe the basic principles of HCF. Central to HCF is the validation of the source IP address of each packet via hop-count inspection. We first describe the hop-count computation, and then present a safe update mechanism that captures the legitimate mappings between IP addresses and hop-count values. Finally, we summarize HCF in the form of a high-level inspection algorithm.

A. Hop-Count Computation

Since hop-count information is not directly stored in the IP header, one has to compute it based on the final TTL value. 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 decreased 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 (OSes) 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.

According to [14], most modern OSes use only a few selected initial TTL values, 30, 32, 60, 64, 128, and 255. This set of initial TTL values covers most of the popular OSes, 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. To resolve ambiguities in the cases of $\{30, 32\}$, $\{60, 64\}$, and $\{32, 60\}$, we will compute a hop-count value for each of the possible initial TTL values, and accept the packet if there is a match with either 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 spoofed. 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 OSes that use "odd" initial TTLs are typically older OSes. We expect such OSes to constitute a very small percentage of end-hosts in the current Internet. Thus, the benefit of deploying HCF should outweigh the risk of denying service to those end-hosts during attacks.

B. Capturing Legitimate Hop-Count Values

To maintain an accurate IP2HC mapping table, we must capture valid hop-count mappings and legitimate changes in hop-count, while foiling any attempt to slowly pollute the mapping

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

Fig. 1. Hop-count inspection algorithm.

table. We can accomplish this through TCP connection establishment. The IP2HC mapping table should be updated *only* by packets belonging to TCP connections in the `established` state [54]. 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 zombie (or flooding source[2]) that sends the SYN packet with a spoofed IP address will not receive the victim's SYN/ACK packet and thus cannot complete the three-way handshake.[3] Using packets from established TCP connections ensures that an attacker cannot slowly pollute a table by spoofing source IP addresses.

While our pollution-proof mechanism provides safety, it may be too expensive to inspect and update the IP2HC mapping 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 k to adjust the frequency of updates (see Section V-A). Note that the pollution-proof mechanism works to capture legitimate changes in hop-count as well as hop-count values of new IP addresses.

C. Inspection and Validation Algorithm

Assuming that an accurate IP2HC mapping table is present (see Section IV for details of its construction), Figure 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 classified as spoofed. Note that a spoofed IP address may happen to have the same hop-count as the one from a zombie to the victim. In this case, HCF will not be able to identify the spoofed packet. However, as shown in Section III-C.1, even with a limited range of hop-count values, HCF is highly effective in identifying spoofed IP addresses.

III. Does Hop-Count Filtering Really Work?

The feasibility of HCF hinges on four factors: (1) diversity of hop-count values, (2) effectiveness in detecting spoofed packets, (3) robustness against evasions, and (4) stability of hop-counts. In this section, we first assess whether 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 a high probability. Second, we consider the effectiveness of HCF against simple spoofing attacks. Third, we evaluate the robustness of HCF by examining various ways an attacker may circumvent filtering, and by showing that evasion would be very difficult without severely limiting the damage or exposing the attacking sources, which, in turn, makes the detection and blockage of the attacking traffic much easier. Finally, the stability of hop-count values is discussed.

A. Diversity of Hop-Count Distribution

Since hop-count values have a limited range, typically between 1 and 30, multiple IP addresses may have the same hop-count values. Consequently, HCF cannot recognize forged packets whose source IP addresses have the same hop-count value to a destination as that of a zombie. It is prudent to examine hop-count distributions at various locations in the Internet to ensure that the limited range doesn't severely diminish the effectiveness of HCF. A good hop-count distribution should have two properties: being symmetric around the mean value, and reasonably diverse over the entire range. Symmetry is needed to take advantage of the full range of hop-count values, and diversity helps maximize the effectiveness of HCF.

Type	Sample Number
.com sites	11
.edu sites	4
.org sites	2
.net sites	12
foreign sites	18

TABLE I
DIVERSITY OF `traceroute` GATEWAY LOCATIONS.

To obtain actual hop-count distributions, we use the raw `traceroute` data from 47 different `traceroute` gateways in [11]. The locations of `traceroute` gateways are diverse as shown in Table I. 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.

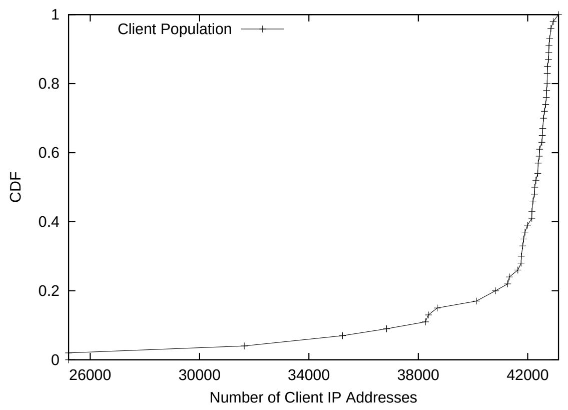

Fig. 2. CDF of the number of client IP addresses.

[2]In this paper, the terms zombie and flooding source are used interchangeably.
[3]There are known vulnerabilities with existing OSes where the initial sequence numbers are fixed or easily predicted. However, this situation can be fixed with a more intelligent selection algorithm.

We examined the hop-count distributions at all `traceroute` gateways and found that the Gaussian distribution (bell-shaped curve) could be a good first-order approximation, but we don't make any claim whether hop-count distributions are indeed Gaussian. Figures 3-6 show the hop-count distributions of four selected sites: a well-connected commercial server, an educational institute, a non-profit organization, and one site outside of the United States.

The mean value μ of a Gaussian distribution specifies the center of the bell-shaped curve. The standard deviation σ describes the girth of the curve–the larger the σ, the more diverse the hop-count distribution, and the more effective HCF will be. For each given hop-count distribution, we use the `normfit` function in Matlab to compute μ and σ. We plot the CDF of the mean and standard deviation of the fitted Gaussian function in Figures 7 and 8. We observe that most of the μ values fall between 14 and 19 hops, and the σ values between 3 and 5 hops. More importantly, in most distributions, the mode accounts for only 10% of the total IP addresses, with the maximum and minimum of the 47 modes being 15% and 8%, respectively. Thus, the hop-count distributions in our data set satisfy both the symmetry and diversity properties to enable very effective filtering.

B. Effectiveness of HCF Against Simple Attacks

We now assess the effectiveness of HCF by asking the question "what fraction of spoofed IP packets can be detected by the proposed HCF?" We assume that potential victim servers know the complete mapping between their client IP addresses and hop-counts (to the victims themselves). In the next section, we will discuss how to construct such mappings. Without loss of generality, we further assume that the attacker evenly divides the flooding traffic among the flooding sources. To make the analysis tractable, we consider only static hop-counts. (The update procedure that captures legitimate hop-count changes has been shown in Section II-B.)

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 between the flooding source and the victim. 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. To assess the performance of HCF against such simple attacks, we consider two scenarios: a single flooding source and multiple flooding sources.

B.1 A Single 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. The shaded area represents those IP addresses — the fraction α_h of total valid IP addresses — that have the same hop-count to the server as the flooding source. Thus, the fraction of spoofed IP addresses that cannot be detected is α_h, and 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. As shown in Section III-A, even if the attacker floods spoofed IP packets from such a zombie, HCF should still identify nearly 90% of spoofed IP addresses. HCF is highly effective against a single attacking source, reducing the attacking traffic by one order of magnitude.

B.2 Multiple Sources

Distributed DoS attacks involve more than a single host. Suppose there are n sources that flood a total of F packets, and each flooding source generates F/n spoofed packets. We assume that each flooding source generates traffic without altering the initial TTL value. If h_i is the hop-count between the victim and flooding source i, then the spoofed packets from source i that HCF can identify is $\frac{F}{n}(1-\alpha_{h_i})$. The fraction, Z, of identifiable spoofed packets generated by n flooding sources is:

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

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 diminish the ability of HCF to identify spoofed IP packets. On the contrary, since hop-count distributions follow a Gaussian distribution, the existence of less effective flooding sources (with small α_h's) reduces the total volume of undetectable attacking traffic.

C. Robustness Against HCF-aware Attackers

Once attackers become aware of HCF, they will attempt to circumvent the hop-count inspection. The robustness of HCF against such HCF-aware attackers is a serious concern to victim servers. In what follows, we first assess the effectiveness of a simple evasion of randomizing initial TTL values. Then, we show that in order to successfully evade HCF, more sophisticated evasion attempts require a large amount of time and resources, and elaborate planning, i.e., casual attackers are unlikely to evade HCF.

C.1 Randomization of Initial TTLs

While the hop-count from a single flooding source to the victim is fixed, randomizing the initial TTL values will create an illusion of attacking packets having many different hop-count values at the victim server. Instead of using the default initial TTL value, an attacker may simply randomize the initial TTL values, hoping that many forged packets may happen to carry matching final TTL values when they reach the victim.

An attacker may generate the full range of hop-counts from 1 to 30 by randomizing initial TTL values from the range $[I_d + h_z - 30,\ I_d + h_z - 1]$, where h_z is the hop-count from the flooding source to the victim and I_d is the default initial TTL value at the flooding source. 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, if I_r follows a certain random distribution $\tilde{R}$, then T_v's follow the same $\tilde{R}$ random distribution. Because the victim derives the hop-count of a received IP packet based on its T_v value, the perceived hop-count of a spoofed source IP address is also $\tilde{R}$ randomly-distributed.

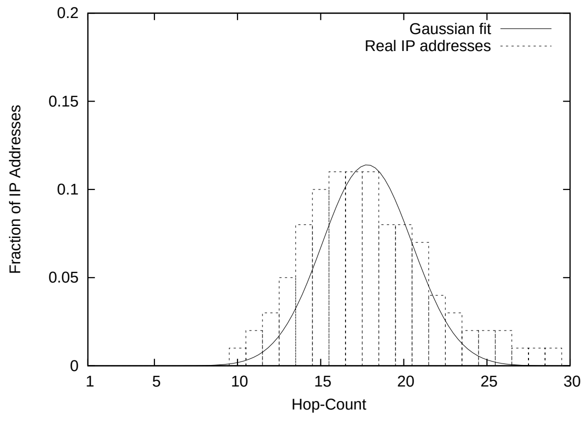

Fig. 3. Commercial.

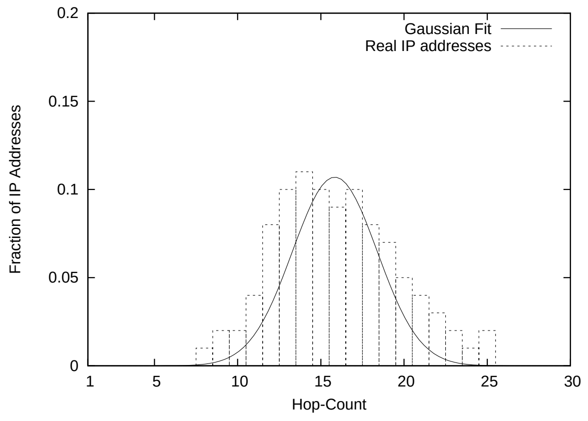

Fig. 4. Educational.

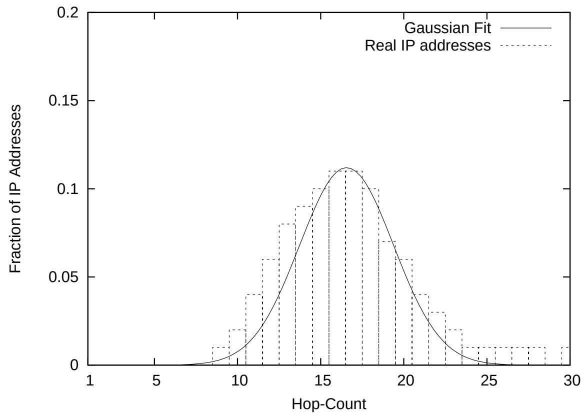

Fig. 5. Non-profit.

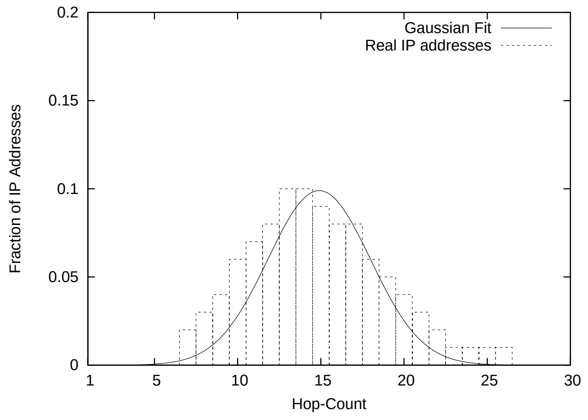

Fig. 6. Foreign.

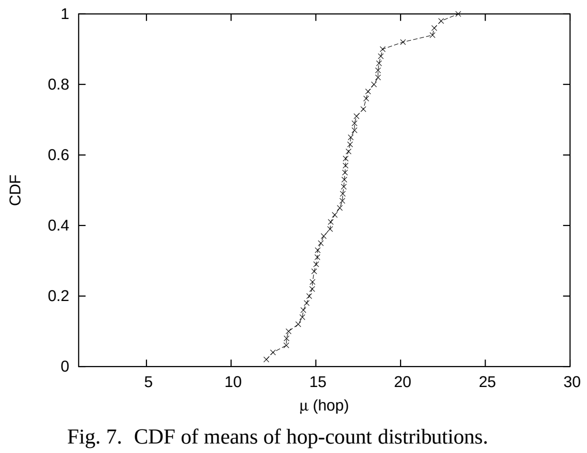

Fig. 7. CDF of means of hop-count distributions.

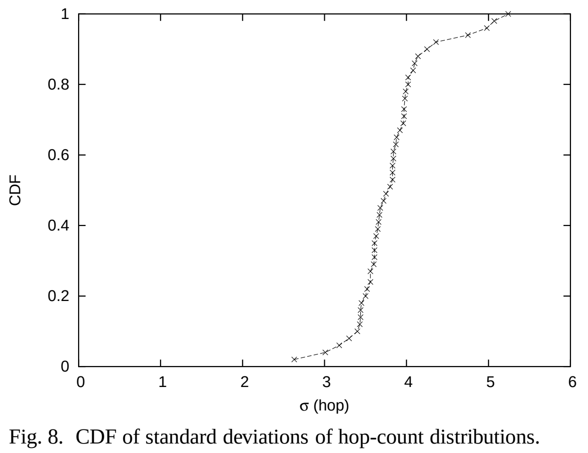

Fig. 8. CDF of standard deviations of hop-count distributions.

As a simple example, we assume that the attacker generates initial TTLs using uniform distribution. Figure 10 illustrates the effect of randomized initial 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 III-A) from real IP addresses to the victim, and the box graph to represent the uniform 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:

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

Here we use $\bar{Z}$ in place of $1 - Z$ to simplify notation. In Fig-

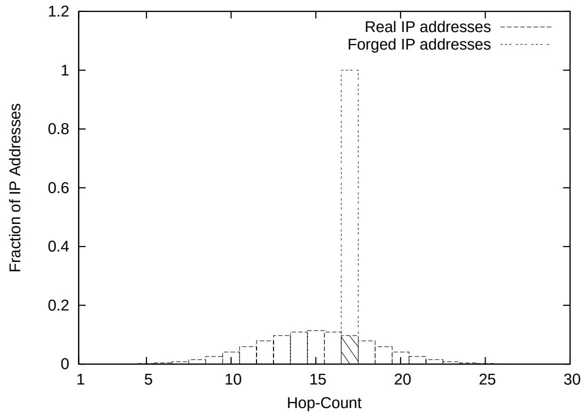

Fig. 9. Hop-count distribution of IP addresses with a single flooding source.

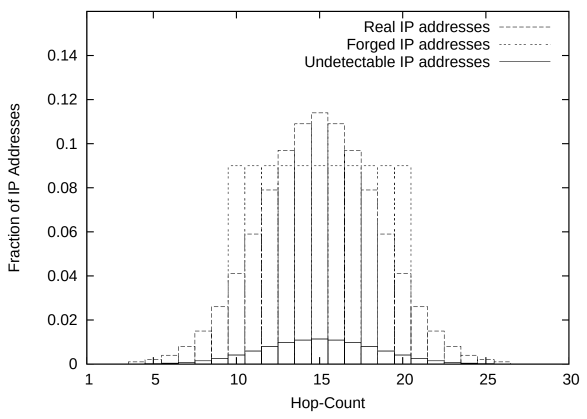

Fig. 10. Hop-count distribution of IP addresses with a single flooding source, randomized TTL values.

ure 10, the range of generated hop-counts is between 10 and 20, so $H = 11$. The summation will have a maximum value of 1, thus $\bar{Z}$ can be at most $1/H = 8.5\%$, which is represented by the area under the shorter Gaussian distribution in Figure 10. 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 a certain $\tilde{R}$ distribution, where the fraction of IP addresses with hop-count h_k is p_{h_k}. If the fraction of the real 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. It is not difficult to see that in order to maximize $\bar{Z}$, an attacker must generate spoofed IP addresses with the most popular hop-count, h_k, for which α_{h_k} is the largest among all αs. Thus, this “more” sophisticated attack is no more threatening than the simple attacks in Section III-B.1.

A more rigorous mathematical analysis of HCF’s robustness against randomized TTL attacks is given in Appendix I.

C.2 Learning of Hop-Count Values

A successful evasion requires that HCF-aware attackers correctly set an appropriate initial TTL value for each spoofed packet. Without loss of generality, we assume 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 initial TTL value of each packet must be set 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 then have the correct final TTL value, $I - (h_s - h_z) - h_z = I - h_s$, when it reaches the victim.

An attacker may easily learn the hop-count, h_z, from a zombie 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 spoofed 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 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. Even if the attacker probes only one host per stub network, with a large IP spoofing space, the probing activity will take considerable amount of time. Moreover, network administrators are alert to unusual access patterns or probing attempts, it would require an excessive amount of effort to coordinate the probing attempts with impunity. Without the correct h_s values, the attacker cannot fabricate the appropriate initial TTL values to conceal forgery.

Without compromising end-hosts, an attacker may compute hop-counts of to-be-spoofed IP addresses based on an existing router-level topology of the Internet, and the underlying routing algorithms and policies. The recent Internet mapping efforts such as Internet Map [9], Mercator [20], Rocketfuel [48], 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 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 advance that can lead to accurate hop-counts based on just Internet topology maps.

Instead of spoofing randomly-selected IP addresses, the attacker may choose to spoof IP addresses from a set of already-compromised machines that are much smaller in number than 2^{32}, so that he can measure all h_s’s and fabricate appropriate initial TTLs. However, this reduces the attacker’s ability to launch a successful attack 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 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]. Thus, the attacker must now modify the available attacking tools, which may be difficult for an unsophisticated attacker.

Overall, although it is not difficult to obtain the appropriate initial TTL for a single IP address, the attacker has to spend a significant amount of time and effort to achieve accurate hop-count information for a large IP spoof space. While HCF cannot eliminate DDoS attacks, it will make it much harder for them to succeed.

D. *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.

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 [37], 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.

Furthermore, recent Internet experiments [28], [41] 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. Within a single domain, a case study of intra-domain routing behavior [44] indicates that the intra-domain topology changes are due mainly to external changes[4] and no network-wide instability is observed.

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 IV-B.

IV. Construction of IP2HC Mapping 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 IP2HC mapping table is critical to detect the maximum number of spoofed IP packets. In this section, we detail our approach to constructing a table. Our objectives in building a 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 the effectiveness of HCF without storing the hop-count for each IP address.

[4]Here the external changes are the routing updates conveyed by external link-state advertisements [44].

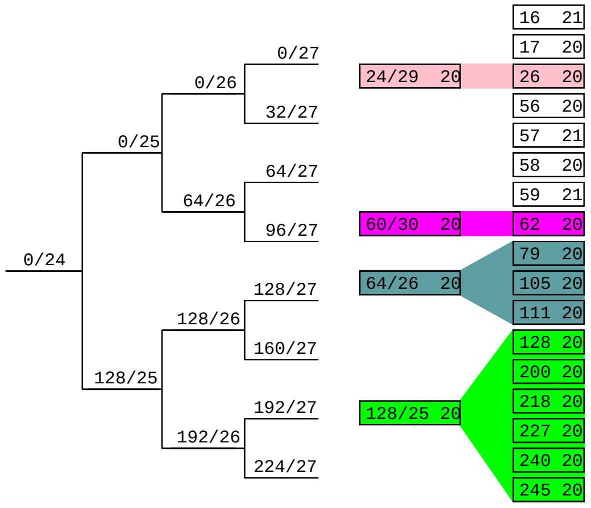

Fig. 11. An example of hop-count clustering.

A. *IP Address Aggregation*

Ideally, the IP2HC mapping table has one entry for each valid IP address. However, this will consume a very large amount of memory, and it is unlikely that an Internet server will receive legitimate requests from all live IP addresses in the Internet. By aggregating IP address, we can reduce the space requirement of IP2HC mapping significantly. More importantly, with IP address aggregation, it is sufficient to capture the hop-count value of one IP address from each subnet in order to build a complete HCF mapping table. In this following, we present and evaluate the techniques for IP address aggregation in constructing IP2HC mapping tables.

A.1 Aggregation Techniques

Aggregating hosts according to address prefix, especially the 24-bit address prefix, is a common method. It is straightforward to implement in practice and can offer fast lookup with an efficient implementation. Assuming an array with one-byte hop-count entry per network prefix, 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 mapping table to find the right hop-count value. For systems with limited memory, the HCF mapping table can be implemented as a hash-table of prefixes of known clients. While 24-bit aggregation may not be the most accurate, it is a certainly a deployable solution.

Since IP addresses within each 24-bit address prefix may be allocated to different physical networks, these hosts are not necessarily co-located and most likely don't have identical hop-counts. To obtain a more accurate IP2HC mapping, one can further divide IP addresses within each 24-bit prefix into smaller clusters based on hop-count. Using a binary tree, we can cluster IP addresses with the same hop-count. The leaves of the tree represent the 254 (excluding the network address and the subnet mask) valid IP addresses inside a 24-bit address prefix. In each iteration, we examine two sibling nodes and aggregate the two nodes as long as they share a common hop-count, or one of them is empty. If aggregation is possible, the parent node will have the same hop-count as the children. We can thus find the largest possible aggregation for a given set of IP addresses. Figure 11 shows an example of clustering a set of IP addresses (with the last octet 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. We are able to aggregate 11 of 17 IP addresses into four network prefixes. The remaining ones must be stored as individual IP addresses.

Based on the BGP routing table information, a network-aware clustering technique [26] has been proposed to identify a group of Web clients that are topologically close to each other and likely to be under a single administration. In contrast, hop-count clustering is self-reliant, and the IP addresses within the same cluster may not be topologically close to each other while they have the same hop-count to the victim server.

To understand whether our clustering method improves HCF over the simpler 24-bit aggregation, we compare the filtering accuracies of mapping tables built under different aggregation techniques.

A.2 Evaluation of Filtering Accuracy

We treat each `traceroute` gateway (Section III-A) as a "web server," and its measured IP addresses as clients to this web server. We build a 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 generates packets by randomly selecting source IP addresses among legitimate clients. We further assume that the attacker knows the general hop-count distribution for each web server and uses it to randomly generate a hop-count for each spoofed packet. This is the most effective DDoS attack that an attacker can launch without learning the exact IP2HC mapping.

We use the percentages of false positives and false negatives to measure filtering accuracy. False positives are those legitimate client IP addresses that are incorrectly identified as spoofed. False negatives are spoofed IP addresses that go undetected by HCF. A good aggregation method should minimize both.

Under each aggregation method, we build an IP2HC mapping table for each web server. Since a 24-bit prefix may contain addresses with different hop-counts, we use the minimum hop-count of all IP addresses inside the 24-bit network address as the hop-count of the network. To filter an IP packet, the source IP address is mapped into the proper table entry through prefix matching, and the hop-count in the packet is checked against the one stored in the table. Since 24-bit aggregation cannot preserve hop-counts for all IP addresses within each address prefix, we examine the performance of three types of filtering policies: 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 more than one hop from those in the table, and +2 Filtering drops packets whose hop-counts differ by more than two hops. 32-bit Strict Filtering is the ideal case where the mapping table has one entry for each valid IP address. Packet filtering based on a table built using hop-count clustering is called *Cluster-based Filtering*.

Figure 12 presents the combined false positive and false negative results for the five filtering schemes: 32-bit Strict, 24-bit Strict, 24-bit +1, 24-bit +2, and Clustering-based Filterings. 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. We observe that the 24-bit strict filtering yields a similar percentage of false negatives as 32-bit Strict Filtering, only 5% of false negatives. This is because the percentage of false negatives is determined by the distribution of hop-counts, and 24-bit aggregation does not alter the hop-count distribution. However, under 24-bit Strict Filtering, most web servers suffer about 10% of false positives, while the percentage of false positives under 32-bit Strict Filtering is zero. As we relax the filtering criterion as in 24-bit +1/+2 Filtering, false positives of 24-bit aggregation are halved while false negatives approximately doubled. If one desires a simpler implementation than Cluster-based Filtering, +1 Filtering under 24-bit aggregation offers a reasonable compromise.

With hop-count-based clustering, we never aggregate IP addresses that do not share the same hop-count. Hence, we can eliminate false positives as long as we update the mapping table when client hop-counts change. As shown in Figure 12, where the points of Clustering-based Filtering overlap with those of 32-bit Strict Filtering, Clustering-based Filtering has nearly identical performance as 32-bit Strict Filtering.

Compared with the 24-bit aggregation, the clustering approach is more accurate but consumes more memory. Figure 13 shows the number of entries in the IP2HC mapping for each web server used in our experiments. The *x*-axis is the web server ID, ranked according to the number of client IP addresses, and the *y*-axis is the number of table entries. The number of entries under Cluster-based Filtering does not include the intermediate nodes used to generate the mapping, e.g., the internal nodes in the clustering trees, because these internal nodes do not need to be stored in the final mapping table. Since the clustering algorithm and the aggregation tree are completely deterministic, we can easily reconstruct the tree on demand to reduce memory consumption. Clustering-based Filtering increases the number of entries by no more than 20% in all but one case, in comparison with the 24-bit Strict Filtering. The 32-bit Strict Filtering, while having slightly higher accuracy, increases the number of entries by at least 67%.

B. Table Initialization and Update

Before running HCF, we need to initialize the IP2HC mapping table and then keep the mapping table updated. The most critical aspect in initializing and updating the IP2HC mapping table is to ensure that only valid IP2HC mappings are stored in the table.

B.1 Initialization and Addition of New Entries

To populate an IP2HC mapping table initially, the administrator of an Internet server should collect traces of its clients to obtain both IP addresses and the corresponding hop-count values.

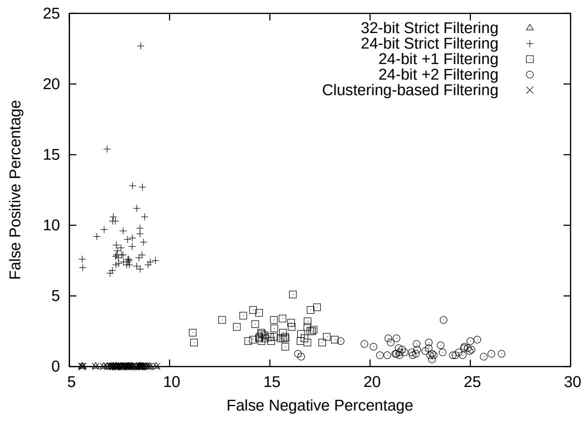

Fig. 12. Accuracy of various filters. (Note that the points of Clustering-based Filtering overlap with those of 32-bit Strict Filtering.)

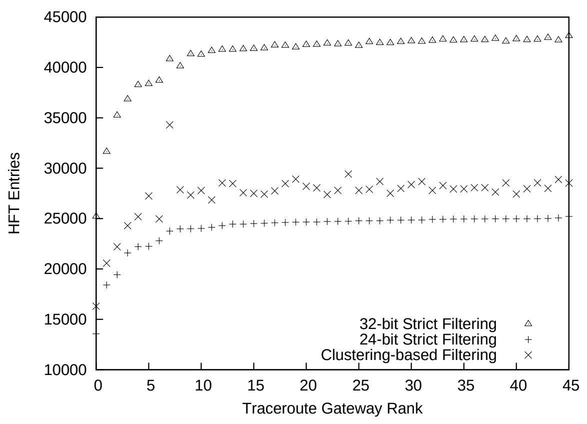

Fig. 13. IP2HC mapping table size.

The initial collection period should be long enough to ensure good filtering accuracy even at the very beginning, and the duration should depend on the amount of daily traffic the server is receiving. For a popular site such as `cnn.com` or `espn.com`, a collection period of a few days could be sufficient, while for a lightly-loaded site, a few weeks might be more appropriate.

After the initial population of the mapping table and activation, HCF will continue adding new entries to the mapping table when requests with previously-unseen legitimate IP addresses are sighted. Thus, over time, the IP2HC mapping table will capture the correct mapping between IP address and hop-count for all clients of a server. This ensures that spoofed IP traffic can be detected, and then discarded with little collateral damage during a DDoS attack.

B.2 Updating Hop-Counts

IP2HC mapping must be kept up-to-date as hop-counts of existing IP addresses change. The hop-count from a client to a server could change as a result of relocation of networks, routing instability, or temporary network failures. Some of these events are transient and therefore can be omitted, but longer-term changes in hop-count must be captured.

Under 24-bit or 32-bit Strict Filtering, a table update involves indexing the table using a given IP address and changing the indexed table entry with a new hop-count. Under hop-count clustering, an update, or adding a new node, may split a node or merge two adjacent nodes on the existing hop-count clustering tree. To carry out the update, one first allocates memory for a new clustering tree. The new clustering tree has a fixed format of depth eight, represented as an array of 511 elements. We populate the array with nodes on the existing clustering tree with updated hop-count(s). We can then repeat the procedure described in Section IV-A.1. In addition, one may need to split an existing node because one of its two immediate child nodes now has a different hop-count as clustering percolates up the tree. When this happens, we replace the parent node with its two child nodes, one having the new hop-count and the other retaining the original hop-count.

Re-clustering should have a relatively small impact on system performance for two reasons. First, each re-clustering instance is a local event on a tree of at most 511 nodes. Second, since hop-count changes are not frequent in the network as reported in [37] and re-affirmed by our own limited observations, re-clustering will probably occur infrequently in practice.

C. *Hop-Count Ambiguity Caused by NATs*

The existence of NAT 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 lower the IP2HC mapping accuracy in the table. However, since a NAT box enforces the assignment of a single source IP address to every outgoing IP packet, this automatically prevents the zombies behind NAT boxes from IP spoofing.

To cope with the hop-count ambiguity caused by NAT boxes, a simple possible solution is to have NAT boxes reset the TTL value of each outgoing IP packet to a default initial TTL. Then, there will be a strict one-to-one mapping between the IP address of a NAT box and a hop-count. While the computed hop-count is the one from the NAT box, instead of the end-hosts behind it, to the victim, this does not affect the filtering accuracy at all as long as the victim maintains the same skewed hop-count as the one computed from IP header. The drawback of this simple solution is the required modification at NAT boxes. However, since NAT boxes must manipulate the IP headers of passing packets anyway, the overhead induced by the proposed TTL reseting is minor.

V. Running States of HCF

Since HCF causes delay in the critical path of packet processing, it should not be active at all times. We therefore introduce two running states inside HCF: the *learning* state captures legitimate changes in hop-count and detects the presence of spoofed packets, and the *filtering* state actively discards spoofed packets. By default, HCF stays in the learning state and monitors the trend of hop-count changes without discarding packets. Upon detection of a flux of spoofed packets, HCF switches to the filtering state to examine each packet and discard spoofed IP packets. In this section, we discuss the details of each state and show that having two states can better protect servers against spoofed IP traffic, while minimizing overhead.

A. *Tasks in Two States*

Figure 14 lists the tasks performed by each state. In the learning 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 legiti-

```
In the learning state:
  for each sampled packet $p$:
    $spoof$ = IP2HC_Inspect($p$);
    $t$ = Average($spoof$);
    if ( $spoof$ )
            if ( $t > T_1$ )
                    Switch to the filtering state;
    Accept($p$);

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

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

    if ( $t \le T_2$ )
            Switch to the learning state;
```

Fig. 14. Operations in two HCF states.

mate 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 filtering state. HCF in the learning state will also update the IP2HC mapping table using the TCP control block of every k-th established TCP connection.

To minimize the overhead of hop-count inspection and dynamic update in the learning state, their execution frequencies are adaptively chosen to be inversely proportional to the server's workload. We measure the 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 k and m, which determine the idle times for table update and inspection, respectively. In contrast, for a heavily-loaded server, both k and m are increased. The two thresholds T_1 and T_2, used for detecting spoofed packets, should also be adjusted based on load. The general guideline for setting execution rates and thresholds with the dynamics of server's workload is given as follows:

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

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

HCF in the filtering state performs a similar set of tasks as in the learning state. The key difference is that HCF in the filtering state must examine every packet (instead of sampling only a subset of packets) and discards spoofed packets, if any. HCF stays in the filtering state as long as a certain number of spoofed IP packets are detected. When the ongoing spoofing ceases, HCF switches back to the learning 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 the learning and filtering states when t fluctuates around T_1. Making the second threshold $T_2 < T_1$ avoids this instability. Note that HCF's filtering accuracy is independent of the settings of T_1 and T_2.

In our filtering accuracy experiments, we have assumed that the IP2HC mapping table holds the complete IP addresses of clients. However, in reality, there are always new requests coming in from unseen address prefixes, regardless of how well a mapping table is initialized or kept up-to-date. To protect against attacking traffic that uses unseen IP addresses, we must drop all packets that have no corresponding entries in the table— the function `IP2HC_Inspect` returns true if p doesn't exist in the table. While undesirable, HCF ensures that legitimate requests from known IP addresses are still served during an attack. Clearly, such collateral damage can be made very low by carefully aggregating IP addresses and diligently populating an HCF mapping table over a long period of time.

B. Blocking Bandwidth Attacks

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

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 alert. 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. 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.

C. Staying "Alert" to DRDoS 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 can easily sustain the availability of its normal service. 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.

VI. 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. For HCF to be useful, the per-packet overhead must be much lower than the normal processing of an IP packet. In addition, since HCF operates at the IP layers, spoofed packets, even when detected, will still consume CPU cycles due to interrupt handling and data link layer processing. We justify the deployment of HCF in practice by measuring the per-packet overhead of HCF and the amount of resource savings when HCF is active.

A. Implementing 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. Since 4096 is relatively small compared to the set of possible 24-bit address prefixes, 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. The clustering overhead has not yet been evaluated.

To implement 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 IP2HC mapping 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.

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

TABLE II

CPU overhead of HCF and normal IP processing.

B. Experimental Evaluation

For HCF to be useful, the per-packet overhead must be much lower than the normal processing of an IP packet. We examine the per-packet overhead of HCF by instrumenting the Linux kernel to time the filtering function as well as the critical path in processing IP 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 traffic 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 traffic 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 two experiments of single-stream `ping`s. The first uses default 64-byte packets at 10 ms intervals, and the second 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 the impact of HCF 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 II.

We present the recorded processing times in CPU cycles in Table II. The column under ’with HCF’ lists the execution times of the filtering function. The column under ‘without HCF’ lists the packet processing times without HCF. Each row in the table represents a single experiment, and each experiment is run with a large number ($\approx$ 40,000) of packets to compute the average number of cycles. We present both the minimum and the average numbers. There exists a difference between average cycles and minimum cycles for two reasons. First, some packets take longer to process than others, e.g., a SYN/ACK packet takes more time than a FIN packet. Second, the average cycles may include lower-level interrupt processing, such as input processing by the Linux Ethernet driver. We observe that, in general, the filtering function uses significantly fewer cycles than the emulated attack traffic, at least an order of magnitude less. Consequently, HCF should provide significant resource savings by detecting and discarding spoofed traffic. Moreover, for both TCP and UDP bulk transfers, the CPU overhead induced by HCF is small (less than 7%). Note that the processing of regular packets takes fewer cycles than the emulated attack traffic. We attribute this to TCP header prediction and the much simpler protocol processing in UDP. 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 15. 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 $(1-\alpha)$.

VII. Related Work

Several efficient mechanisms [17], [19], [35], [52] are available to detect DDoS attacks. In addition, researchers have also used the distribution of TTL values seen at servers to detect abnormal load spikes due to DDoS traffic [39]. 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,

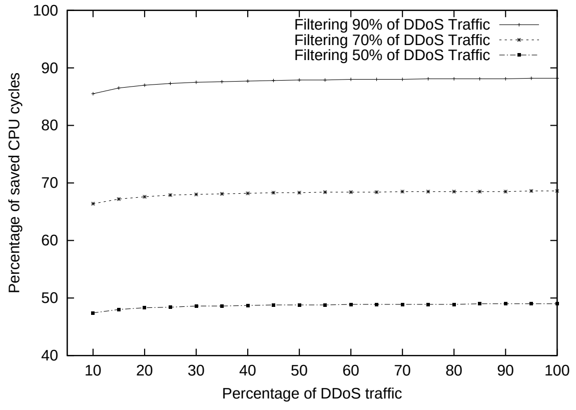

Fig. 15. Resource savings by HCF.

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 independently of, our work, the possibility of using TTL for detecting spoofed packet was discussed in [51]. 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 these 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], [29], [36], [55], 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, router-based distributed packet filtering (DPF) [36] utilizes routing information to determine whether an incoming packet at a router is valid with respect to the inscribed source and destination IP addresses in the packet. The experimental results reported in [36] 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 that are easy to trace back.

To validate that an IP packet carries a true source address, SAVE [29], 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) [55] 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, marking in Pi 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 also exist commercial solutions [22], [34] that block the propagation of DDoS traffic with router support.

However, the main difference between our scheme and these solutions 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. HCF works well because no single entity controls the value of the TTL field, and thus, destinations can use it to fingerprint legitimate IP packets. We speculate that end-host-based filtering can be much more effective if intermediate routers could do more than merely decrementing TTL by one.

VIII. 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 incoming packets to validate their legitimacy. Using only a moderate amount of storage, HCF constructs an accurate IP2HC mapping table via IP address aggregation and hop-count clustering. A pollution-proof mechanism initializes and updates entries in the mapping table. By default, HCF stays in the *learning* state, monitoring abnormal IP2HC mapping behaviors without discarding any packet. Once spoofed DDoS traffic is detected, HCF switches to the *filtering* state and discards most of the spoofed packets.

By analyzing actual network measurements, we have shown that HCF can remove 90% of spoofed traffic. Moreover, even if an attacker is aware of HCF, he 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 Internet servers against spoofed IP packets. Furthermore, HCF is readily deployable in end-systems since it does not require any network support.

There are several issues that warrant further research. First, 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. Second, 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] D. J. Bernstein and Eric Schenk. Linux kernel SYN cookies firewall project. Available: http://www.bronzesoft.org/projects/scfw.

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

[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, December 2000.

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

[19] 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.

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

[21] A. Hussain, J. Heidemann, and C. Papadopoulos. A framework for classifying denial of service attacks. In *Proceedings of ACM SIGCOMM '2003*, Karlsruhe, Germany, August 2003.

[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. Juels and J. Brainard. Client puzzle: A cryptographic defense against connection depletion attacks. In *Proceedings of NDSS'99*, February 1999.

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

[26] B. Krishnamurthy and J. Wang. On network-aware clustering of web clients. In *Proceedings of ACM SIGCOMM '2000*, Stockholm, Sweden, August 2000.

[27] A. Kuzmanovic and E. W. Knightly. Low-rate TCP-targeted denial of service attacks (the shrew vis themice and elephants. In *Proceedings of ACM SIGCOMM '2003*, Karlsruhe, Germany, August 2003.

[28] 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.

[29] 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.

[30] J. Li, M. Sung, J. Xu, and L. Li. Large-scale ip traceback in high-speed internet: Practical techniques and theoretical foundation. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

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

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

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

[40] 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.

[41] 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.

[42] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Proceeding of 5th Annual Linux Showcase & Conference*, pages 165–172, November 2001.

[43] 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.

[44] 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.

[45] 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.

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

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

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

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

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

[51] 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.

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

[53] X. Wang and M. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003.

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

[55] 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.

[56] D. Yau, J. Lui, F. Liang, and Y. Yam. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. *IEEE/ACM Transactions on Networking*, 13(1), February 2005.

APPENDIX

I. ANALYSIS OF RANDOMIZED TTL ATTACKS AGAINST HCF

An attacker may somehow modify the initial TTL values, hoping that the forged packets may carry matching hop-count values when they reach the victim. Assuming that the attacker knows the range of valid hop-count $[1, h_M]$ to a given victim server, an attacker may use initial TTLs in the range $[I_d + h_z - h_M,\ I_d + h_z - 1]$, where h_z is the hop-count from the flooding source to the victim and I_d is the default initial TTL value at the flooding source. Thus, the randomization of initial TTL values is the same as subtracting the right constant between 1 and h_M from $I_d + h_z$ to give a matching IP2HC pair that HCF cannot detect.

Here, we study the scenario where for any given IP address, an attacker is only able to select at random a hop-count value from the range $[1, h_M]$. Without loss of generality, we use the random variable X as the hop-count value the attacker draws from a distribution $P_X(k), k = 1, 2, ..., h_M$.

The ability of an attacker to evade HCF can be measured by the probability that the hop-count value X embedded in a spoofed IP packet would match the correct hop-count value Y, or $P[X = Y]$, using Bayes' formula as follows:

$$P[X = Y] = \sum_{k=1}^{h_M} P[X = Y \mid Y = k] \cdot P[Y = k],$$

where $P[Y = k] = P_Y(k), k = 1, 2, ..., h_M$, is the distribution of hop-count values to the victim server. $\{X = k\}$ and $\{Y = k\}$ are two independent events. Thus, $P[X = Y \mid Y = k] = P[X = k] = P_X(k)$. We thus reduce the above equation to:

$$P[X = Y] = \sum_{k=1}^{h_M} P_X(k) \cdot P_Y(k).$$

The attacker would want to to maximize the summation over all possible P_Xs, or equivalently, use the "best" distribution P_X of hop-count values to fool HCF.

$$\max_{P_X} \sum_{k=1}^{h_M} P_X(k) \cdot P_Y(k).$$

This is a special case of a standard Linear Programming problem, but with a very simple solution. We observe that there exists a k_M, for which $P_X(k) \leq P_X(k_M)$ for all k. Therefore, we can provide an upper-bound for the maximization as follows:

$$\sum_{k=1}^{h_M} P_X(k) \cdot P_Y(k) \leq \sum_{k=1}^{h_M} P_X(k) \cdot P_Y(k_M) = P_Y(k_M) \sum_{k=1}^{h_M} P_X(k).$$

Since the sum $\sum_{k=1}^{h_M} P_X(k) = 1$, it follows that the maximization is bounded by $P_Y(k_M)$:

$$\sum_{k=1}^{h_M} P_X(k) \cdot P_Y(k) \leq P_Y(k_M).$$

One way to achieve this upper-bound is to draw from a distribution, where $P_X(k) = 0$ for all $k \neq k_M$, and $P_X(k_M) = 1$. We have shown that the mode $(P_Y(k_M))$ in our data collection is generally around 10%, so on average, only 10% of spoofed IP addresses can evade HCF.