NetHCF: Filtering Spoofed IP Traffic With Programmable Switches

Menghao Zhang, Guanyu Li, Xiao Kong, Chang Liu, Mingwei Xu, Guofei Gu, and Jianping Wu, *Fellow, IEEE*

Abstract—In this paper, we identify the opportunity of using programmable switches to improve the state of the art in spoofed IP traffic filtering, and propose NetHCF, a line-rate in-network system to filter spoofed traffic. One key challenge in the design of NetHCF is to handle the restrictions stemmed from the limited computational model and memory resources of programmable switches. We address this by decomposing the HCF scheme into two complementary parts, by aggregating the IP-to-Hop-Count (IP2HC) mapping table for efficient memory usage, and by designing adaptive mechanisms to handle routing changes, IP popularity changes, and network activity dynamics. We implement an open-source prototype of NetHCF, and conduct extensive evaluations. The evaluation results demonstrate that NetHCF is able to process most legitimate traffic in 1 μs, filter spoofed IP traffic effectively under network dynamics, with less than 30% of switch resource occupation.

Index Terms—Hop-count filtering, programmable switches, spoofed IP traffic

1 Introduction

Spoofed IP traffic remains a significant threat to the Internet, which is commonly associated with malicious network activities, especially Distributed Denial of Service (DDoS) attacks [2]. Although many research projects and IETF drafts have been devoted to thwarting spoofed IP traffic, IP address spoofing continues to be a prevalent problems, and the attacks associated with it are still frequently reported in the news [3]. According to the Spoofer Project of CAIDA [4], 24.4% of the Autonomous Systems (ASes) do not deploy any countermeasure to disable spoofed IP traffic, and around 16.1% IP addresses in the Internet can be spoofed. Furthermore, geographical tests show that the networks that allow IP spoofing are distributed worldwide, making this a widespread problem.

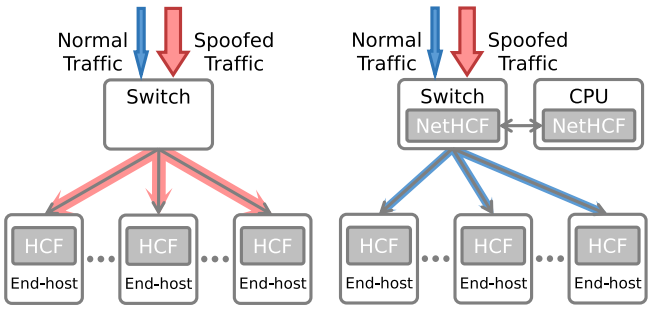
- *Menghao Zhang is with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with Kuaishou Technology, Beijing 100085, China. E-mail: zhangmenghao0503@gmail.com.*
- *Guanyu Li, Xiao Kong, Chang Liu, Mingwei Xu, and Jianping Wu are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, also with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology (BNRist), Beijing 100084, China. E-mail: {ligy18, kx19, l-c19}@mails.tsinghua.edu.cn, xumw@tsinghua.edu.cn, jianping@cernet.edu.cn.*
- *Guofei Gu is with the Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77840 USA. E-mail: guofei@cse.tamu.edu.*

Manuscript received 20 May 2021; revised 10 Jan. 2022; accepted 10 Mar. 2022. Date of publication 22 Mar. 2022; date of current version 14 Mar. 2023. This work was supported in part by the National Key R&D Program of China under Grant 2018YFB1800302, and in part by the National Natural Science Foundation of China under Grants 61832013 and 61872426. A preliminary version of this paper appeared in the conference of ICNP 2019 [1]. (Corresponding authors: Guanyu Li and Mingwei Xu.) Digital Object Identifier no. 10.1109/TDSC.2022.3161015

Although an attacker can forge any field in the IP header, she/he cannot easily falsify the number of hops an IP packet takes to reach its destination, because it depends on the underlying network paths and routing mechanisms. Based on this observation, some previous works [5], [6] propose hop-count based defense mechanisms called *Hop-Count Filtering* (HCF), which can filter spoofed IP traffic (e.g., TCP, UDP, ICMP, etc.) with an IP-to-Hop-Count (IP2HC) mapping table. To guarantee correctness and prevent pollution by attackers, the IP2HC mapping table should only be updated by legitimate packets. One common approach is to monitor the establishment procedures of TCP connections, and update the table only using the connections that have been successfully established. This table can then be used to filter spoofed packets with inconsistent hop counts.

State-of-the-art HCF filtering mechanisms [5], [6] are all located at the end hosts. Indeed, until recently, the conventional wisdom has been that switch hardware must be simple, fixed, and stateless. This necessarily means that the monitoring of TCP establishment procedures has to be performed in the network stacks at end hosts. Actually, they have been partly integrated into the Linux ecosystem. The key downside of these systems is that spoofed packets cannot be filtered until they arrive at the destination servers, which already incurs bandwidth waste, deployment redundancy, and delayed setup of the full mapping table (details in Section 2.2). Even if the packet filtering part of these systems can be separated and installed in edge switches, interactions between the switch and servers are unavoidable, bringing considerable table updating delay and bandwidth resource consumption.

The emergence of programmable switches [7], [8] is shifting the paradigm of simple and fixed switch hardware. Programmable switches enable us to offload intelligence into networks, providing opportunities to filter the spoofed traffic at line rate in switches. Since the programmable



(a) Existing HCF Scheme (b) NETHCF

Fig. 1. NETHCF is a novel re-design of the HCF defense.

switching ASICs (Application-Specific Integrated Circuits) can easily process a few billion packets per second (∼Tbps throughput) [9], [10], one switch could serve tens to hundreds of servers, saving the precious computation and memory resources on end servers. Besides, spoofed IP traffic is filtered before entering the victim network, avoiding unnecessary bandwidth waste and potential collateral damage. We can also deploy such a switch at a strategic location in the network, so that it could set up the IP2HC mapping table faster. A switch-based design can also achieve lower latency and jitter, which are critical to latency-sensitive applications [11], [12], compared to a server-based design. This is because of the notable performance gap between the switching ASICs and general computation resources (e.g., CPU).

Despite these substantial benefits, applying traditional HCF techniques in a switch-based design is non-trivial. In order to design a correct and efficient in-network line-rate HCF, we must carefully address several design challenges. First, for HCF to take effect, the switch must maintain a correct and up-to-date IP2HC mapping table for table lookup. However, the switching ASICs only have limited on-chip memory (i.e., SRAM and TCAM), which is impossible to store the full IP2HC mapping table. Second, as the network is dynamic, the HCF system should also be adapt to network dynamics, e.g., legitimate hop-count changes, IP popularity changes, and network attacks. However, the switching ASICs have a very restrictive computational model, which is challenging to adapt to network dynamics in a timely and effective manner.

To address the challenges above, in this paper, we present NETHCF, an in-network spoofed traffic filtering system. As shown in Fig. 1, different from traditional HCF designs, we decouple the existing HCF scheme into two complementary parts, a data plane *cache* on switching ASICs and a control plane *mirror* with general computation resources. The cache serves the "hot keys," or legitimate packets, at line rate in the data plane. The mirror processes the packets that miss the cache, maintains the IP2HC mapping table, and constantly adjusts the posture of NETHCF to adapt to network dynamics. These two parts rely on each other to overcome their limitations, achieving both correctness and efficiency. We implement a prototype of NETHCF with Barefoot Tofino [7], and make its source code publicly available here [13]. Our prototype and evaluation demonstrate that NETHCF can achieve line-rate and adaptive spoofed traffic filtering with negligible overheads.

In summary, our contributions in this paper include:

- We analyze the limitations of the traditional HCF designs, and identify new opportunities for improving HCF with programmable switching ASICs (Section 2).
- We propose NETHCF, a line-rate in-network spoofed traffic filtering system. We decouple the traditional HCF system into two complementary parts to fit the computational model of switching ASICs, aggregate the IP2HC mapping table to cache more entries in the data plane, and design several mechanisms to make NETHCF adapt to end-to-end routing changes, IP popularity changes, and network activity dynamics (Section 3).
- We implement a prototype of NETHCF, which is publicly available on Github [13]. We also conduct extensive evaluations to show that NETHCF is able to guarantee the line-rate processing for legitimate traffic and filter the spoof traffic effectively, with negligible overheads (Sections 5, and 6).

Finally, we discuss several issues in Section 4, describe related works in Section 7 and conclude this paper in Section 8.

2 Background and Motivation

In this section, we first give some backgrounds on spoofed packet filtering techniques and the HCF scheme, then describe the problems of the existing HCF scheme, and finally show the advantages of programmable switches to resolve the aforementioned headaches.

2.1 Background on HCF

Source address spoofing is among one of the most serious problems that plague the Internet. To thwart this, researchers have proposed two distinct kinds of approaches: *router-based* and *host-based*. The router-based approaches install defense mechanisms inside routers to trace the source(s) of attack [14], [15], [16], [17], [18], or detect and block the attacking traffic with coordinated routers [19], [20], [21], [22], [23], [24], [25]. However, these solutions require not only the router support but also coordination among diverse routers and networks, even wide-spread deployment to achieve their potentials. In contrast to router-based approaches, host-based approaches can be deployed with lower requirements. And the end systems (e.g., the edge customers of the Internet, small ISPs, data centers or enterprises) have a much stronger incentive to deploy defense mechanisms than network service providers. The host-based approaches use sophisticated source-discrimination schemes [26], [27], [28] or significantly reduce the resource consumption of each request [29], [30], [31] to withstand the flooding traffic.

HCF [5], [6] falls into the category of host-based solutions and shines from other approaches because of its simplicity, gracefulness and effectiveness. It can validate incoming IP packets at an Internet server without any cryptographic methodology or router support, making it lightweight and appealing to customers (victims). Besides, HCF effectively mitigates the attack asymmetry between attackers and victims, where victims can easily capture the legitimate IP2HC

TABLE 1
Comparison of HCF, Middlebox-Based HCF and NETHCF. Y for `yes`, N for `no`, U for `uncertainty`

	HCF	Middlebox-based HCF	NETHCF
Bandwidth waste	Y	N	N
Deployment redundancy	Y	N	N
IP2HC slow-setting-up	Y	N	N
Low packet processing capability	U	Y	N
High latency and jitter, poor performance isolation	U	Y	N

mapping to conduct spoofed packet filtering while the attackers are nearly impossible to know the mapping between arbitrary IP address and its hop count to the victims. In particular, from the points of view of victims, they can easily infer the hop-count information by subtracting the final TTL from the initial TTL at the arriving place. According to [32], most modern OSes use only a small set of selected initial TTL values, such as 30, 32, 60, 64, 128 and 255. As almost all the hosts in the Internet are apart by less than 30 hops, one can determine the initial TTL value of a packet by selecting the smallest initial TTL value in the set that is larger than its final TTL. While for attackers, since the hop-count information is deeply rooted in the Internet routing infrastructure and hop-count values is highly diverse, it is nearly impossible for them to maintain consistent hop-count values with random spoofed IP addresses. These basic observations guarantee the effectiveness of the HCF scheme.

2.2 Problem Statement

Existing HCF implementations are all located at the end hosts. Especially, the original paper [5], [6] implements the HCF inside the Linux kernel and integrates it with the network stack. However, under the requirements of modern distributed systems and data centers, there are several practical problems, as we summarize in Table 1.

First, existing HCF suffers from bandwidth waste. Spoofed IP traffic cannot be filtered until it arrives at the targeted end hosts, which still saturates the bandwidth resource of the victim networks and potentially results in collateral damage. In the data centers nowadays, besides the workload traffic from the Internet (i.e., Internet traffic), there is also west-east traffic across different servers (e.g., inter-service traffic, control traffic) [33], [34], [35], [36], [37]. To illustrate this, we select three gateway systems including Tripod [37], StatelessNF [35] and FTMB [36] as representatives. In these gateway systems, each server runs some network functions (NF) and NF states are replicated among different servers via control traffic to maintain high availability. The HCF kernel module is deployed on each server. We use a large number of mice flows with spoofed IP addresses to attack these gateway systems. When the spoofed traffic arrives, as we can see from Fig. 2, although each server can filter most spoofed packets, these packets have entered the network and occupied the network bandwidth already, leading to extra network congestion and additional packet losses, which compromises the availability of these gateway systems significantly.

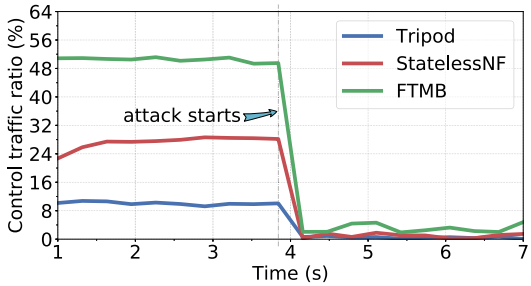


Fig. 2. Bandwidth waste of the original HCF.

Second, existing HCF is deployed on each end host to conduct spoofed IP packet filtering, which not only leads to deployment redundancy but also consumes the precious general computation resources (e.g., CPU, memory) of end hosts. If these resources are saved, more tasks can be conducted. We measure the resource consumption of the HCF kernel module on the server, and show the results in Table 2. The server is equipped with a 40 Gbps NIC and several CPU cores each with a minimum frequency of 1.2 GHz and a maximum of 3.2 GHz. When the NIC load is 50%, the CPU clock frequency has reached the maximum value. When the NIC load is 75%, the HCF kernel module (excluding the normal protocol stack) fills up one CPU core. In addition, the HCF kernel module also requires 1.98 GB memory to store the IP2HC table. Note that each server has to be equipped with the full HCF kernel with same CPU and memory consumption, which is pretty redundant.

Third, with the rapid growth of network bandwidth, modern data centers often use load balancing to distribute the large volume of traffic to a cluster of servers (i.e., from Virtual IP (VIP) to Direct IP (DIP) mapping) [9], [33], [34]. This results in that each end host (i.e., DIP) can only see a portion of incoming traffic, which makes setting up a full IP2HC mapping table extremely slow and postpones the time of HCF to take effect. We replay an Enterprise traffic trace from CAIDA [38] and simulate different DIP numbers to see how this affects the IP2HC mapping table setting up speed. As shown in Fig. 3, the blue dotted line indicates the ideal IP2HC setting-up of the entire network while the remaining solid lines indicate the IP2HC setting-up on each end host when the traffic is load-balanced to different numbers of hosts. Obviously, as the number of hosts increases, the speed of setting up a IP2HC table on the host becomes slower and slower, which postpones the time for HCF to take effect significantly.

An intuitive solution to resolve the headaches above is to implement HCF in servers and make them stand as middleboxes. However, software-based HCF has two fundamental limitations. First, processing packets in software limits capacity, usually at ~10 Gbps and ~M packets per second [39]. We can scale out the packet processing capacity by

TABLE 2
Resource Consuming of the Original HCF

Traffic Load	25%	50%	75%	100%
CPU	46.31%	69.75%	99.64%	139.51%
DRAM	1.98 GB	1.98 GB	1.98 GB	1.98 GB

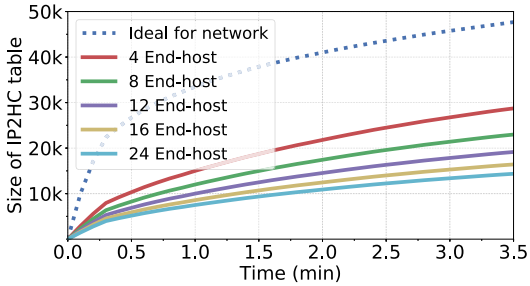
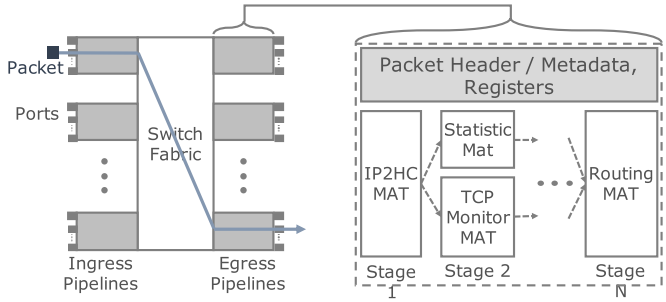


Fig. 3. Slow setting-up of the original HCF.



(a) Switch data plane architecture (b) Pipeline architecture

Match	Action
src IP = 10.0.0.0 / 24	read_hc_with_index (0)
src IP = 192.168.0.5 / 32	read_hc_with_index (1)
...	...
src IP = 192.56.1.0 / 24	read_hc_with_index (N - 1)

(c) Match-Action Table architecture

Fig. 4. Programmable switch data plane.

adding more servers, but doing so raises costs and operational complexity. For example, handling a typical attack traffic volume today (∼Tbps) [40] requires hundreds of servers, which are both costly and difficult to manage. Second, processing packets in software incurs high latency and jitter, with poor performance isolation. Software processing adds a latency of 50us to 1 ms when handling as little as 100 K packets per second [9], [41], which is unacceptable for many latency-sensitive applications [11], [12] in data centers today. When software experiences a flash crowd, legitimate traffic served by the server also experiences increased delay, even unexpected packet drops, which make matters worse.

2.3 Programmable Switches

Recent trends in Software-Defined Networking (SDN) have raised programmable switching ASICs [7], [42], [43] and related domain-specific languages (e.g., P4 [44]) to extend the networking programmability from the control plane to the data plane. Compared to traditional fixed-function switches, the emerging programmable switches offer flexible programmability without sacrificing any performance, even with same power consumption and price [7]. The new hardware provides unprecedented opportunity to overcome the shortcomings of the original HCF design.

There are multiple ingress and egress pipelines in programmable switches, each with multiple ingress and egress ports (Fig. 4(a)). When a packet reaches one of the ingress ports, it is first processed by an ingress pipeline, then queued and switched to one of the egress pipelines to be processed, and finally emitted to a specified egress port. Inside a pipeline, packets are processed sequentially by each stage (Fig. 4(b)), which has its own dedicated resources, including match-action tables and registers. Match-action tables are applied to process packets by matching certain header fields or metadata of the packet and performing actions (e.g., modifying header fields/metadata, read/write registers or drop packets) according to the matching results (Fig. 4(c)). Registers are used to store necessary data or intermediate states to realize stateful packet processing.

With programmable switches and domain-specific languages like P4 [44], developers can customize their data plane. Programmers typically write P4 programs to define packet headers, build packet processing graphs, and specify the match fields and actions of each match-action table. The compiler provided by switch vendors can compile the program to binaries and generate interactive APIs. The binaries are loaded into the data plane with corresponding tools, and the APIs are used by control plane applications to interact with the data plane.

The programmable switching ASICs and P4 language make it straightforward to implement custom terabit packet processing devices, as long as the defined logic can be fitted into the match+action model of switching ASICs. Developers need to carefully design the processing pipelines of their programs to meet the resource and timing requirements of switching ASICs. The major constraints of the current switching ASICs include [10], [45], [46], [47]: 1) the number of pipelines, and the number of stages and ports in each pipeline; 2) the amount of TCAMs (for wildcard and prefix matching of match-action tables) and SRAMs (for prefix matching and registers) that each stage can access; 3) reading and writing to registers must satisfy some restrictions, i.e., a program can only access a register array from tables and actions in the same stage; all registers in a stage must be accessed in parallel; each register array can only be accessed once per packet, with a stateful ALU which conducts simple function, such as simultaneous read and write, conditional update, and basic mathematical operations.

To conclude, while the programmable switching ASICs have their own limitations in the computational model and on-chip resources, they can process packets with high throughput, low latency and jitter, and nearly perfect performance isolation, which provides an unprecedented opportunity to move toward a better HCF. We must get around these limitations and make some clever designs to achieve our goal: serving the legitimate traffic with low latency and filtering the spoofed traffic effectively.

3 Our Approach: NetHCF

As discussed above, recognizing the problems of HCF and identifying the opportunities with programmable switching ASICs, we propose NetHCF, an in-network line-rate spoofed traffic filtering system.

3.1 NetHCF Overview

While the on-chip memory size (TCAM and SRAM) in the switching ASICs (50-100MB [9]) has grown rapidly in recent years, it is still challenging to store the full IP2HC mapping table directly in the switching ASICs. A typical IP2HC

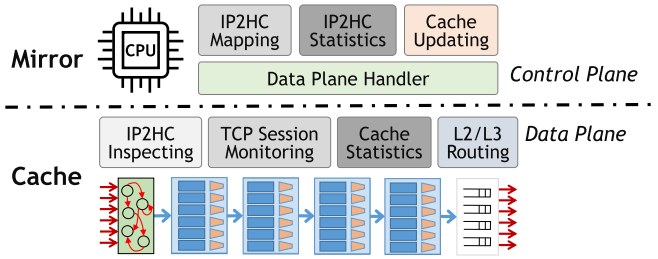


Fig. 5. NETHCF logical architecture.

mapping entry should store the mapping from IP (32 b) to hop-count (5 b), thus simply storing the entire IP2HC mapping table requires at least $2^{32} \times (32 + 5)$ bits (~10 GB) in the switching ASICs. An intuitive approach to resolve it is to store a small hash of the match field (IP). However, on one hand, although we can take various techniques to mitigate the impact of hash collisions (e.g., SilkRoad [9]), it is still impossible to eliminate all the hash collisions thus the disruptions of legitimate flows will occur frequently. And most malicious flows do not strictly obey the TCP state transition map, this collision problem will become much worse. On the other hand, even with hash compression, storing all the IP2HC mapping entries is still difficult[1]. As a result, only a small portion of the IP2HC mapping table can be stored, and dynamic updates have to be taken to adapt to the traffic dynamics [48], which will lead to a dilemma (drop/pass) during attacks when the incoming packet cannot find its IP in the IP2HC mapping table.

Instead of accepting this apparent compromise, we observe network traffic has a *heavy tail* nature that a small portion of IP2HC mapping entries usually serve the vast majority of the legitimate network traffic [49], [50], [51], [52]. Therefore, as shown in Fig. 5, we decouple the existing HCF into two parts logically, an HCF *cache* at the data plane (programmable switching ASICs), and an HCF *mirror* at the control plane (general computation resources, such as server clusters). The data plane cache serves for most active and legitimate IPs at line rate, and the control plane mirror handles the remaining IPs with its complex logic, maintains the IP2HC mapping table, and adjusts the state of NETHCF to adapt to network dynamics. These two parts interact with each other through carefully designed coordination mechanisms to achieve both the advantages of programmable switching ASICs (high performance) and general computation resources (high flexibility) while avoiding the shortcomings of them.

The data plane cache handles the most frequent IPs and reports the packets whose IPs are not in the switching ASICs to the control plane mirror. It mainly consists of three new modules[2], an *IP2HC Inspecting* module which stores the hottest IPs of the whole IP2HC mapping table and inspects the validity of these IPs at line rate, a *TCP Session Monitoring* module which captures the legitimate hop-count changes and updates these IP2HC entries if necessary, and a *Cache Statistics* module which maintains a real-time statistic to count the counters for each cached IP2HC mapping entry and reports hot entries to the control plane for cache updating. As a complementary part to the data plane cache, the control plane mirror maintains a global view for IP2HC mapping and IP2HC statistics. It also plays an important role in aggregating IP2HC entries and handling the packets whose IPs are missed in the cache. More importantly, it is responsible for adjusting the posture of NETHCF to adapt to network dynamics, i.e., routing changes, IP popularity changes, and network activity dynamics.

When NETHCF starts to run for the first time, network operators should collect traces of its clients to obtain both IP addresses and the corresponding hop-count values. This initial collection procedure should be long enough to have a high covering rate for the whole IP space, and the concrete duration depends on the amount of daily traffic the victim is receiving. For example, for popular sites such as `facebook.com`, a few days could be sufficient, while for lightly loaded sites, a few weeks might be more appropriate. After the initial collection procedure, the control plane mirror organizes the IP2HC mapping table in a binary tree and aggregates the entries with an efficient aggregation algorithm (Section 3.2). Meanwhile, the control plane mirror is supposed to constantly insert a number of legitimate IP2HC entries into the data plane cache until the number of entries in the data plane cache becomes relatively stable (the data plane cache is full).

After this initialization, NETHCF would continue adding new entries to the IP2HC mapping table when previously unseen IP addresses are sighted (by sending *packet digests* to the control plane mirror). More importantly, at this relatively stable running state, NETHCF needs to adapt to network dynamics timely and effectively(Section 3.3). First, NETHCF should capture legitimate hop-count changes and update the corresponding IP2HC mapping entries resulted from end-to-end routing updates in the Internet. Second, NETHCF should accommodate to the dynamic incoming traffic to ensure that the hottest IPs are always stored in the cache. Third, to minimize collateral damage and adapt to attack activities, NETHCF has two running states, a *learning* state when packets with wrong hop-count are simply passed and a *filtering* state when these packets are discarded. These two states are switched according to the number of spoofed packets which fail at IP2HC checking in a period of time.

3.2 IP2HC Mapping Table Organization

Although we can store one IP2HC mapping entry for each IP address in the switch ASICs, this would consume a large amount of switch memory and further exacerbate the memory shortage. We observe that many IP addresses with the same prefix share the same hop-count values, especially for a subnet. Therefore, we aggregate the IP2HC mapping table to utilize the limited on-chip memory more efficiently. More importantly, this will help quickly build a complete IP2HC mapping table with the hop-count value of one IP address from each subnet.

In the control plane mirror, to achieve efficient and correct aggregation, we represent the IP2HC mapping table in a binary tree, as shown in Fig. 6. Each leaf in the tree

1. Silkroad [9] compresses both match field and action data of each entry, and they can only store ~10 M entries. This number is still far away from the whole IP space (2^{32}), even with IP address aggregation.

2. In NETHCF, *L2/L3 Routing* module is directly inherited from the traditional switch, and we omit its detail here.

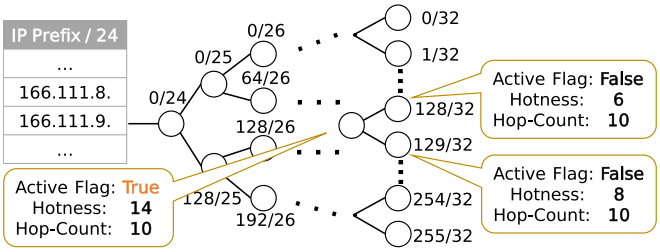


Fig. 6. IP2HC mapping table organization.

represents a valid IP address, while the other nodes represent a specific IP address prefix. Each node in the tree has three attributes, indicating whether it is active, its hotness and its hop-count value. We define the active node as a basic entry for the IP2HC mapping table, and only the active nodes will be stored in the data plane cache. Since it is common that a 24-bit address prefix is allocated to the same physical network, we terminate the IP2HC mapping table aggregation at the 24-bit address prefix, i.e., the depth of the binary tree is limited to 8, and the leaves of the tree represent the 256 valid IP address inside a 24-bit address prefix. Our algorithm runs iteratively across the tree. In each iteration, if two sibling nodes share the common hop-count value, we aggregate them into their parent node with the same hop-count value. To accelerate the building of the IP2HC mapping table, we also aggregate the node whose sibling node is empty. Then the parent node will be selected as the active node for its children node(s), assigned with the same hop-count value. The hotness of a parent node is the sum of all its children nodes' hotness. In this way, we can find the largest possible aggregation for a given set of IP addresses. For example, the IP address range 166.111.8.128 to 166.111.8.255 can be aggregated into 166.111.8.128/25 prefix if all these IP addresses share the same hop-count value. Note that there is no dependency between different active nodes so that we can store some of them in the data plane cache independently. In the data plane cache, to store the IP2HC mapping entries, we design a match-action table where the match field stores the IP/IP prefix and the action field stores the corresponding hop-count value.

3.3 Adapting to Network Dynamics

NetHCF should accommodate to the end-to-end routing changes in the Internet where hop-counts change legitimately, adapt to the traffic dynamics where the popularity of IPs changes over time, and minimize the potential collateral damage where legitimate packets go through the control plane unnecessarily. In the first case, NetHCF should capture the up-to-date legitimate IP2HC changes timely and update the entries at both the data plane cache and the control plane mirror immediately. In the second case, NetHCF should capture the IP access statistics and change the IP2HC mapping entries in the data plane cache to ensure cache hotness. For the last case, we propose two running states to make NetHCF adapt to the attack activities.

3.3.1 Capturing Legitimate Hop-Count Changes

Although hop-count has been proved to be pretty stable [6], [53], [54], there are still cases when hop-count may change, such as routing instability and network reallocation. These changes should be captured as soon as possible to update the IP2HC mapping in both the cache and the mirror.

Algorithm 1. IP2HC Incremental Update Algorithm

```
1: Function aggregate (ip2hc, startNode)
2:   curNode = startNode
3:   while True do
4:     sibNode = getSiblingNode(ip2hc, curNode)
5:     if sibNode == Null or sibNode.hc == curNode.hc
       then
6:       parNode = getParentNode(ip2hc, curNode)
7:       if parNode == Null then
8:         break
9:     parNode.hotness = curNode.hotness + sibNode.hotness
10:    parNode.hc = curNode.hc, parNode.repFlag = True
11:    curNode.repFlag = False, sibNode.repFlag = False
12:    curNode = parNode;
13:  else
14:    break
15:  return
16: Function split (ip2hc, startNode, ipSrc, newHC)
17:  curNode = startNode
18:  while curNode.prefixLen < 32 do
19:      nextBit = getBitOfIP(ipSrc, curNode.prefixLen + 1)
20:      nextNode = getChildNode(ip2hc, curNode, nextBit)
21:          otherNode = getChildNode(ip2hc, curNode,
           nextBit⊕1)
22:       otherNode.hc = curNode.hc, otherNode.repFlag =
          True
23:      nextNode.hc = newHC, nextNode.repFlag = True
24:      curNode.repFlag = False, curNode = nextNode
25:    return
26: Function incrementalUpdate (ip2hc, ipSrc, newHC)
27:    currentNode = indexWithIP(ip2hc, ipSrc)
28:    if currentNode.prefixLen == 32 then
29:      currentNode.hc = newHC
30:      aggregate(ip2hc, currentNode)
31:    else
32:      split(ip2hc, currentNode, ipSrc, newHC)
33:    return
```

Update in the control plane mirror. First, the new IP2HC mapping information should be reported to update the IP2HC mapping table in the control plane mirror immediately. Cache-missed packets are transferred to the control plane mirror in the filtering state directly. While in the learning state, to reduce the communication overheads, we only deliver the *packet digests*, i.e., IP address, IP TTL, and TCP flag, instead of the whole packets. For cache-hit packets, the updated legitimate hop-count values are generated from the *TCP Session Monitor* module and are delivered to the control plane mirror directly. Then the control plane mirror would conduct incremental update for the IP2HC mapping table. As shown in Algorithm 1, if the new IP2HC mapping corresponds to an active node (32 b prefix), update it directly and aggregate it with its sibling node iteratively when possible (**Function** `aggregation()`). Else, split the tree and re-select the new active nodes (**Function** `split()`).

Update in the data plane cache. Second, it is also essential to timely update the hop-count of the IP2HC mapping entries in the data plane cache. A strawman solution is to report the TCP handshake packets to the mirror and to update the

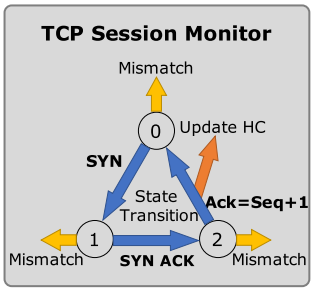


Fig. 7. TCP Session Monitor (learning state).

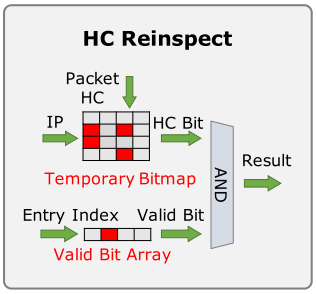


Fig. 8. HC Reinspect table.

hop-count of the IP2HC mapping entry with issued control messages, just as the way to insert it initially. However, entry insertion with the control plane is not atomic, and it takes a few milliseconds [9] (even tens of milliseconds when the control plane is at another server). This indicates that an IP address may have many packets arrived before the control plane completes entry insertion into the cache, and these packets would be classified as spoofed packets. If at the filtering state, NETHCF would discard these packets expectedly, which would cause a significant performance penalty.

To solve this problem, NETHCF introduce the *TCP Session Monitoring* module in the switching ASICs to capture the legitimate hop-count changes. Under normal circumstances (learning state), TCP Session Monitor module monitors the cache-hit TCP handshake packets with inconsistent hop-count values. Only packets that follow the strict TCP state transition map (*Transition State Array*) and correct Seq/Ack number transitions (*Seq_ack_number Array*) are accepted as legitimate TCP connections (Fig. 7). When attacks happen (filtering state), this becomes much more complicated. If we simply let pass all the hop-count inconsistent TCP SYN packets, then all TCP SYN flood packets [55] will also pass the HC checking. In contrast, if we drop these TCP SYN packets, these packets will not get responses (TCP SYN-ACK packets) from the servers to finish their three handshakes, which would stop us from monitoring new TCP connection establishment procedures and capturing legitimate hop-count changes. To resolve this, TCP Session Monitor module changes the posture of itself to a *SYN Cookie* based session monitoring mechanism. As shown in Fig. 9, for a hop-count inconsistent TCP SYN packet, TCP Session Monitor module responses a TCP SYN-ACK packet to wait for the TCP ACK packet. If the replied TCP ACK packet has a correct Seq/Ack number, the module would reply a TCP RST packet to force the client to re-establish the connection and mark this session's *Transition State* flag (*Transition State Array*). Then this connection can be monitored as in the normal circumstances.

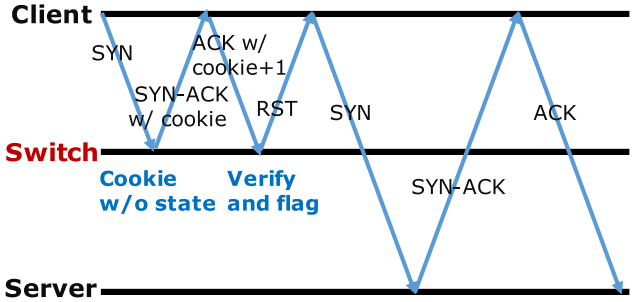


Fig. 9. Switch-based TCP SYN Cookie workflow.

After the TCP Session Monitor module, as shown in Fig. 8, we also introduce an HC Reinspect table, which has a *Valid* flag for each IP2HC mapping entry (*Valid Array*) and a *Temporary Bitmap* to store the temporary legitimate hop-count values. The TCP Session Monitoring module upstream generates an updated legitimate hop-count value and encapsulate it into metadata to deliver to the subsequent stage. The subsequent stage at the HC Reinspect table would update the corresponding IP2HC mapping entry's valid flag as invalid and record the new hop-count value v in the Temporary Bitmap immediately. As shown in Fig. 8, the Temporary Bitmap has N rows and 32 columns. To reduce hash collision, inspired by *bloom filter*, we use k hash functions to map the IP address to different rows, $r_1, r_2, \ldots, r_k$. Then the $r_1, r_2, \ldots, r_k$-th row and v-th column of the Temporary Bitmap is set as 1. For a packet whose matching entry is invalid, if its hop-count value matches the Temporary Bitmap, i.e., the bits in all the hashed rows and hop-count value's column are 1, NETHCF regards it as valid, or vice verse. In this way, we achieve the line-rate hop-count update in the switching ASICs, avoiding the potential race conditions and ensuring the per-IP packet-processing consistency. Note that the hop-count is pretty stable, these hop-count changing cases are rare, which indicates that the row number N in the Temporary Bitmap and the size of the TCP Session Monitor module could be very small. And all the invalid IP2HC entries in the data plane would be updated by the control plane in the upcoming update period.

3.3.2 Capturing IP Popularity Changes

To cope with traffic dynamics where IP popularity changes, the mirror of NETHCF should periodically update the cache with the hottest IP2HC mapping entries. Netcache [10] provides an insightful approach to realize a similar goal. The data plane selects and reports the hot keys from the uncached entries with a heavy-hitter detector, and the control plane compares the hits of the heavy-hitter detectors with the counters of the cached entries to evict less popular keys and insert more popular keys. However, this methodology does not apply to NETHCF, since NETHCF faces a more adversarial scenario. Simply adopting this approach would lead to *fake hot phenomenon*, and the legitimate hot IPs in the data plane would be replaced by the aggressive fake IPs accessed deliberately by the attackers, which would degrade the performance of the legitimate IPs and packets.

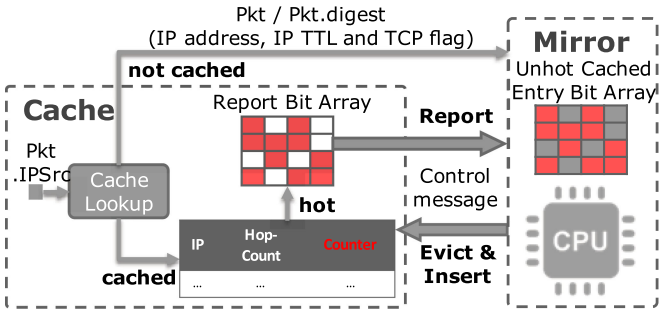


Fig. 10. Cache updating mechanism.

We observe that the main reason for this fake hot phenomenon is that the cache does not have a full view for the uncached IP2HC mapping and cannot determine whether the uncached IP access is legitimate or not, which make it prone to be tricked. As a result, as shown in Fig. 10, rather than providing a heavy-hitter detector for uncached IPs/IP prefixes in the data plane cache, we deliver the packets whose IPs/IP prefixes are uncached to the control plane mirror for processing. Since the control plane mirror has the full views for the whole IP2HC mapping, it can easily distinguish the fake ones from uncached legitimate IP accesses. In particular, in both cache and mirror, we maintain a hit counter (i.e., hotness) for each legitimate IP (*Counter Array*) and a total unhit counter for all the packets that fails IP2HC checking (*Spoofed-packet Counter*). To reduce the communication cost between the data plane cache and the control plane mirror, only digest (IP address, IP TTL, and TCP flag) instead of the whole packet is delivered to the control plane mirror at the learning state. While at the filtering state, the cache send the whole packet for further IP2HC checking and pass/drop decision. For the control plane mirror, to identify the hottest IPs and update the cache accordingly, it must obtain the statistics of the cache. A straightforward approach is to adopt the classic poll mode of SDN to fetch all these data plane counters directly. As there are hundreds of thousands of entries in the cache, it is too expensive. To reduce this overhead, we amortize the overhead across a period: when the counter of an entry exceeds a preset threshold, the cache reports the IP/IP prefix of the entry to the mirror. This can transform the original one-time poll cost into a series of small push operations. In each period, to remove duplicate hot entry reports to the mirror, a *Report Bit Array* is attached behind in the data plane, which guarantees that each hot entry is reported at most once. After the control plane obtains all the hot entries from the data plane cache, it adopts the thought of complementary set and screens out the unreported entries of the cache, which are essentially the cold entries that have not been visited in the last period. Then the control plane mirror select the hottest entries from all uncached legitimate IPs/IP prefixes, and update these data plane cold entries with the selected IPs/IP prefixes. Note that the number of cold entries is relatively small, as the cache is updated periodically. Besides, based on our experiments on the Barefoot Tofino switch, the table entry update capability can reach as high as 160 K entries per second. The small cold entry number and high entry update capability jointly demonstrate that the cache update is not a heavy burden under our circumstances. With all the techniques above, the control plane achieves the correct popular-IP capturing and resilient hot-entry updating.

3.3.3 Running States of NetHCF

Even though we are trying hard to offload as many IP2HC mapping entries as possible into the switching ASICs, it is still impossible to store all of them. As a result, cache-missed packets must be directed to the mirror for decision (pass/drop), which would cause the additional delay for these packets. We observe that the attack scenarios only occupy a small portion of all network activities, thus NetHCF should not be active at all times. Therefore, we introduce two running states for NetHCF to make it adapt to network activity dynamics: the *learning* state which captures the legitimate changes in hop-count and detects the number of spoofed packets, and the *filtering* state which actively discards the spoofed packets with wrong hop-counts. Besides, these two running states can also reduce the unnecessary SYN Cookie reset for legitimate flows in the normal scenarios and also give operators freedoms to choose the running state of NetHCF as they desire. By default, NetHCF stays at the learning state and monitors the changes of hop-count without dropping packets. Upon detecting a large number of spoofed packets in a specific period (larger than threshold T_1), NetHCF switches to the filtering state and discards the spoofed packets. NetHCF stays at the filtering state as long as a certain number of spoofed packets are detected. When the number of spoofed packets decreases and is less than another threshold T_2, NetHCF switches back to the learning state. Note that T_2 should be smaller than T_1 for better stability, which can avoid the unnecessary fluctuates between two states. The filtering accuracy of NetHCF depends on the setting of T_1 and T_2.

In the filtering state, we assume NetHCF has the whole IP2HC mapping table for the complete IP addresses in the mirror. However, this assumption may not stand in reality. There are always new requests from unseen IP addresses, regardless of how well the IP2HC mapping table is initialized or kept up-to-date. To defend against malicious traffic using unseen IP addresses in the filtering state, we must discard these requests that have no corresponding entries in the IP2HC mapping table under attacks. While undesirable, only in this way could NetHCF ensure legitimate packets from known IP addresses are still served during an attack. Certainly, such collateral damage could be made extremely low if the IP2HC mapping table becomes more and more complete (i.e., NetHCF takes a long period of time at the learning state).

3.4 Putting All Together: NetHCF Structure

Data plane cache pipeline. The data plane cache is the core component of NetHCF. It mainly realizes (1) an *IP2HC Inspecting* module to inspect the validity of packets, (2) a *TCP Session Monitoring* module to capture the changes of legitimate hop-count and update the hop-count values at line rate, (3) a *Cache Statistic* module to provide essential legitimate IP hit and spoofed IP hit statistics for cache update and running state switching. The overall structure of the cache is shown in Fig. 11. The *IP2HC Inspecting*

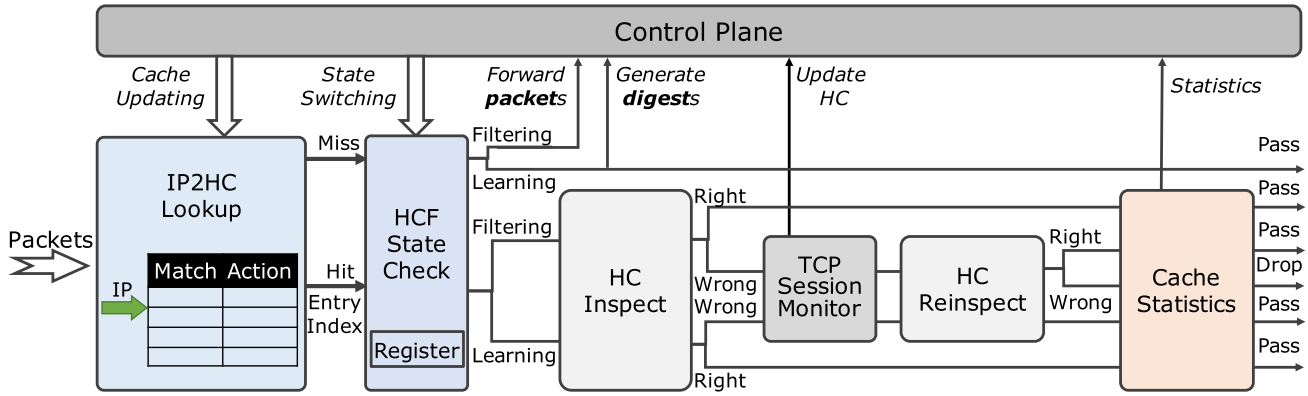


Fig. 11. The workflow of the data plane cache.

module is located at three separated stages of the pipeline. The *IP2HC Lookup* table first matches on the IP address and gives the corresponding hop-count value and entry index in the action data. The *HC Inspect* table calculates the packet's hop-count value and compare it with the pre-fetched hop-count value from the IP2HC Lookup table. In the *HC Reinspect* table, the *Valid Array* (with the same size of IP2HC lookup table) is attached behind to indicate whether the corresponding entry is still valid. Along with the Valid Array, there is a *Temporary Bitmap*, which is used to record the valid hop-count values for updated IPs/IP prefixes (Fig. 8). The *TCP Session Monitoring* module consists of two register arrays, one for TCP transition state (*Transition State Array*) and the other for TCP seq/ack number (*Seq_ack_number Array*) (Fig. 7). Only flows which strictly conform with TCP state transition map are allowed to update the Valid Array. The *Cache Statistics* module is composed of a *Counter Array*, a *Report Bit Array* and a *Spoofed-packet Counter*. When a packet hits the cache and passes the hop-count checking, the Counter Array increases the value in the corresponding entry index location by one. If this value is above the threshold configured by the control plane, this index will keep being marked as hot before the counters are refreshed by the control plane. The Report Bit Array is used to remove the duplicate reports. If the hop-count checking fails, the Spoofed-packet Counter increases by one. The Spoofed-packet Counter is reported to the control plane periodically.

Control plane mirror. The control plane mirror serves as a complementary part for the data plane cache, thus it realizes all the packet processing logic as the cache does, except that this view is global. More importantly, it maintains the binary tree based data structure to record the aggregated IP2HC mapping table. Besides, it maintains a unique index for each IP2HC mapping entry, which is used to index the valid array, the counter array and the report bit array in the data plane cache. It also maintains an IP2HC mapping entry management table, which records whether each active node is in the data plane, its counter, and a heap pointer. We maintain a heap to quickly find the hottest entries in the uncached active nodes. There is also a spoofed-packet counter in the control plane mirror, which records the number of packets failing at the hop-count checking to adjust the state (learning/filtering) of NETHCF.

4 DISCUSSION

NETHCF is used to filter spoofed IP packets with inconsistent hop counts, which shares a similar threat model as the origin HCF scheme [6]. From the algorithm level, it also shares the similar limitations as the original HCF scheme, such as it is difficult to cope with Network Address Translator (NAT) scenarios, where multiple hop-count values correspond to the same IP address. This is becoming much better as IPv6 is being deployed more and more widely recently [56] (The *TTL* field in IPv4 is renamed as *Hop Limit* field in IPv6), and our cache-based NETHCF will become more and more necessary as the address space of IPv6 is much larger than that of IPv4. We also do not talk much about the robustness of NETHCF against various evasion techniques, since they have been broadly discussed in the original paper. NETHCF mainly improves the performance of the original end host based HCF scheme with new system-level designs, so we mainly discuss the unique system-level attributes, which have not been covered before.

Deployment mode. In real-world deployment scenarios, the spoofed traffic is usually dispersed and high-volume, which may exceed the capability of one server and suffer from denial-of-service attacks. As a result, the control plane mirror is usually deployed as a cluster of servers [57], which communicate with the switch control plane agent to update the entries in the data plane cache. Each server in the cluster would undertake a portion of uncached spoofed IP traffic via the data plane switch ports, further achieving high scalability. Besides the control plane scalability, the data plane cache can also be deployed in a set of programmable switches. When single switch cannot provide sufficient storage ability, we can employ multiple switches in a pipeline to accommodate more legitimate IP2HC entries. On the other hand, if NETHCF wants to support higher bandwidth, we can deploy several switches in a parallel way and distribute the incoming traffic evenly to them. Furthermore, to be more general, pipeline extension and parallel extension can be combined together to provide the desired scalability.

Resource constraints. During our engineering with NETHCF, we find the main bottleneck does not come from the throughput. Instead, the memory space is easy to reach the upper limit. Fortunately, the recent trends in programmable switches are leveraging external DRAM in servers to alleviate the resource shortage [58], which may help us

TABLE 3
Replayed Traffic Workloads

#	Traffic	Avg. Flow Length	Avg. Packet Size	Size
1	BigFlow	19.0 packets/flow	451B/packet	2.50 GB
2	SmallFlow	3.3 packets/flow	291B/packet	1.60 GB
3	Enterprise	9.5 packets/flow	622B/packet	1.74 GB

cache more (even complete) IP2HC mapping entries in the data plane, and mitigate this potential attack greatly.

Relationship with SYN Cookie. SYN Cookie is an effective approach for end hosts to prevent TCP SYN flooding attacks initially, which has also evolved into various network-based SYN Proxy mechanisms [59], [60]. Comparing with SYN Cookie, NetHCF can filter all kinds of spoofed IP traffic effectively, not limited to TCP protocol. Although NetHCF leverages SYN Cookie to prevent itself to suffer from TCP SYN flooding, it extend its defense capability to other kinds of spoofed IP traffic, especially ICMP and UDP protocols.

5 Implementation

We have implemented an open source prototype of NetHCF, including all components of the cache and the mirror described in Section 3. The source code is publicly available here [13].

The cache is implemented with ~1 K lines of P4 [44] code and is compiled to Barefoot Tofino ASIC [7] with Barefoot Capilano software suite [61]. The IP2HC Lookup table has 256 K entries totally, with a sub-table to store aggregated entries and another sub-table for 32-bit un-aggregated IP address. Correspondingly, the size of the Valid Array is also 256 K. Since legitimate hop-count changes happen rarely, we set the row number of Temporary Bitmap as 1024 and the hash function number k as 2. For the Cache Statistic module, the Counter (8-bit) Array and the Report Bit Array is also 256 K. The TCP Session Monitoring table contains two register arrays, one for Transition State (2-bit) and another for TCP seq/ack number (4-byte), each with 65536 slots. All of these above result in only a small portion of TCAM and SRAM occupation (Section 6.4), leaving enough space for traditional network processing. For TCP Session Monitoring module, we use built-in hash functions in P4 on both 5-tuple and reverse 5-tuple,[3] and perform XORing on two hash values to represent a bidirectional connection. Furthermore, we reuse the L2/L3 Routing module from traditional switches and just add pass/drop decision function for it.

For ease of prototype, the mirror is written with ~3 K lines of Python code. It can be deployed on one or more servers with lightweight protocols (i.e., ZeroMQ [62]) to communicate with the switch control plane agent. In future, we plan to re-implement the control plane mirror with C-based `DPDK` [63] to further improve the performance.

3. The 5-tuple means (ipSrc, ipDst, protocol, portSrc, portDst), and we denote (ipDst, ipSrc, protocol, portDst, portSrc) as the reverse 5-tuple.

Fig. 12. Bandwidth saving with NetHCF.

6 Evaluation

Our evaluation seeks to answer the following key questions:

- How does NetHCF perform compared with the original host-based HCF [6] (Section 6.2)?
- How effective are the techniques and optimization we have designed in NetHCF (Section 6.3)?
- What are the micro-benchmark metrics of NetHCF (Section 6.4)?

6.1 Experimental Setup

Our testbed is composed of one 3.3 Tb/s Barefoot Tofino switch (Wedge 100BF-32X) and three servers, each of which is equipped with 12 Intel(R) Xeon(R) E5-2698 v4 CPUs and 128 GB memory. These three servers are connected to the switch via 40 Gbps Intel XL710 NICs. In particular, one server serves as the control plane mirror, one server runs an Apache HTTP server with default settings, and one server serves as a client or traffic generator, running `wget`, `iperf` or `tcpreplay` tools. In our experiments, we reset bitmap and all counters in the cache statistics module every three seconds. Our workload traffic is collected from CAIDA [38], including an *Enterprise* traffic trace (about 300 K flows and 200 K IPs), a *BigFlow* traffic trace (around 300 K flows and 200 K IPs), and a *SmallFlow* traffic trace (about 1500 K flows and 250 K IPs), for an extensive evaluation on traces with different characteristics (Table 3). Considering values of hop-count follow a Gaussian distribution ($\mu = 16.5$, $\sigma = 4$) in reality [6], we generate spoofed traffic with hop-count of 16 to mimic a clever attacker, so that more attack packets may happen to have the right hop-counts.

6.2 Performance Improvement

First, to show the effectiveness of NetHCF in filtering spoofed traffic, we run `wget` to generate legitimate TCP traffic and replay the spoofed traffic on the client side simultaneously. Fig. 12 shows the throughput at the client side (top subfigure) and the server side (bottle subfigure). As we can see from this figure, although the client side launches a large number of spoofed packets, only very few of them arrive at the server side. This is because NetHCF detects the attack and switches to filtering state to adapt to network attacks immediately. Compared with the original HCF scheme, by pushing intelligence into the network, NetHCF is able to prevent attack traffic from entering the host network and preserve bandwidth for legitimate traffic.

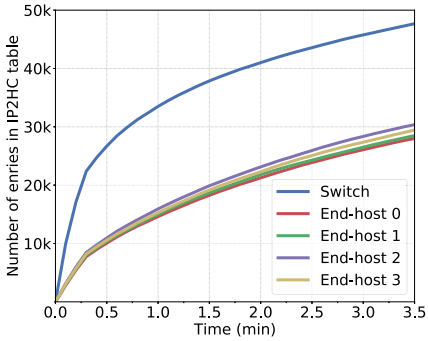


Fig. 13. Setting-up speed of the IP2HC mapping table.

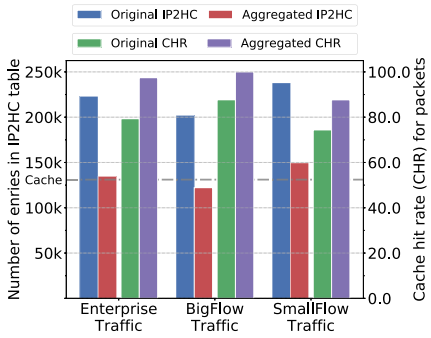


Fig. 14. Effectiveness of the IP2HC aggregation mechanism.

Then, to show the setting-up speed of the IP2HC mapping table, we add another three servers loaded with original HCF scheme and make the client replay the workload traffic traces. We employ ECMP to distribute the traffic to these four servers. The results for the three traces are very similar, so we only show the result for the *Enterprise* traffic. As shown in Fig. 13, the IP2HC mapping table of NETHCF is set up faster than that of HCF, since NETHCF can view the full traffic space. Beside, the total size of the IP2HC table on four hosts is much larger than that of NETHCF, and plenty of identical entries exist in the IP2HC mapping tables on these four end hosts, which also indicates that NETHCF is more efficient in resource utilization.

6.3 Optimization Effectiveness

HCF scheme decoupling. To demonstrate the benefit of decoupling the existing HCF scheme, we compare our cache-based NETHCF with an intuitive approach, adopting a hash-based compression to store a portion of IP2HC mapping table in the data plane (hash-based NETHCF) [48]. We set the same switch memory for these two NETHCF schemes and select the false negative as the metric, i.e., the proportion of spoofed packets identified as normal ones. Table 4 shows that cached-based NETHCF stands out with much lower false negative, even though multiple hash functions are employed to reduce the collisions for hash-based approach. Although we can use more hash functions to further reduce false negative, this would result in unacceptable switch resource occupation, since each hash function has to occupy one stage in the switch pipeline. Notice that ~10% of spoofed packets happen to have the same hop-count with the packets from real IP addresses, which is the inherent limitation of the HCF scheme itself. Besides the high false negative problem, hash-based NETHCF also suffers from a dilemma (drop/pass) during attacks when the incoming packet cannot find its IP in the IP2HC mapping table. This dilemma does not happen in our cache-based NETHCF, since the control plane mirror has the full IP2HC view. In conclusion, NETHCF can guarantee to filter the maximum number of spoofed packets as the original HCF scheme.

TABLE 4
False Negative Comparison

	Approach	False Negative
Hash-based NETHCF	One Hash Function	31.03%
	Two Hash Functions	27.87%
	Three Hash Functions	24.70%
Cache-based NETHCF		9.89%

IP2HC mapping table aggregation. We demonstrate the effectiveness of the IP2HC aggregation technique in Fig. 14. For the control plane mirror, aggregation significantly reduces the number of IP2HC entries that need to be maintained, saving considerable memory resources. More importantly, for the data plane cache, aggregation allows the cache to store hop-count values for more IP addresses and improves the cache hit rate, leading to fewer packets transferred to the mirror. Especially for the *BigFlow* traffic, with aggregation, the size of IP2HC table becomes smaller than the capacity of the cache, so the entire IP2HC table can be put into the cache.

Adapting to legitimate hop-count changes. To show the effectiveness of NETHCF in handling hop-count changes, we analyze and compute the ratio of dropped packets at the server side. As shown in Fig. 15, those solid lines represent the control plane hop-count update mechanism with different traffic traces, and the dotted line represents NETHCF which updates the hop-count values at line rate. As we can see from this figure, when the attack starts (filtering state), the control plane update strategy would cause 0.2% packet losses because of temporary inconsistent hop-counts (Section 3.3.1), while line-rate hop-count update of NETHCF avoids this and keeps no packets dropped. In comparison with our ICNP version [1], we redesign the Temporary Bitmap from a 32-bit bit array into a N-row and 32-column bitmap, and use a bloom filter based mechanism to index the corresponding hop-count values. Based on our simulation, for packets with invalid IP2HC mapping entries, under the current settings (k = 2, N = 1024), the false negative reduces from 20% ~40% to less than 1%.

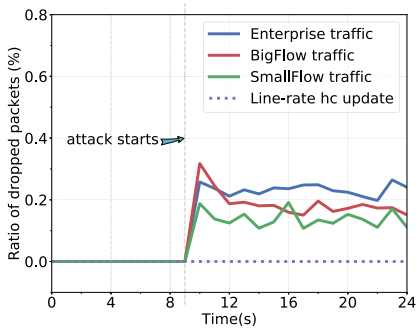


Fig. 15. Effectiveness of the line-rate hop-count updating.

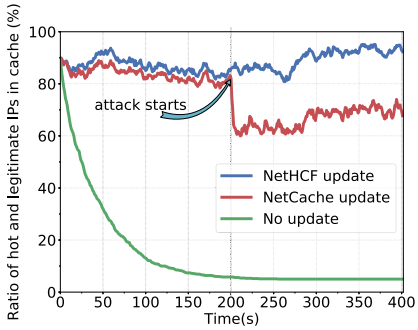


Fig. 16. Effectiveness of the NetHCF cache update mechanism.

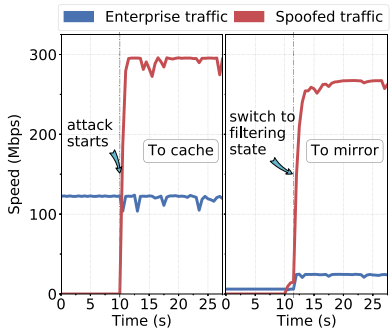


Fig. 17. Communication overhead between cache and mirror.

Adapting to IP popularity changes. To evaluate the effectiveness of the NetHCF cache update mechanism, we replay the spoofed traffic and the *Enterprise* workload traffic simultaneously under these three different cache updating mechanisms, and select the percentage of legitimate and hot entries in the cache as the metric. Fig. 16 demonstrates that the update mechanism of NetHCF performs well all the time, while that of NetCache [10] falls short because of *fake hot phenomenon* (Section 3.3.2) when the attack happens. This indicates that NetHCF is adaptive to IP popularity changes even in adversarial scenarios. If we do not update the cache, just few entries are continuously hot for long connections while most entries may not produce a match any more.

6.4 Micro Benchmarks

Resource utilization. Table 5 displays the resource usage of NetHCF in our Tofino switch. As we can see, NetHCF occupies less than 20% of computational resources, and less than 30% of TCAM and SRAM. We have achieved significant memory saving compared with our ICNP version [1], which occupies one-third of TCAM and half of the SRAM. This memory saving comes from the design of the IP2HC Lookup table, which replaces the IP2Index Lookup table and the Hop-Count Register Array previously. Note that even with all the data plane components, NetHCF still leaves enough space for traditional network processing, and this can even be further optimized by more tuning.

Communication between mirror and cache. To show communication between two parts of NetHCF, we replay the *Enterprise* traffic as workload and initiate the attack at 10 s. We use the bytes of traffic transferred from the cache to the mirror as the metric. As shown in Fig. 17, in the learning state (0∼10 s), only minor workload traffic is steered from the cache to the mirror. When the attack starts and NetHCF switches to the filtering state (after 10 s), a large portion of spoofed traffic needs to be processed in the mirror. This is because the IPs for most spoofed traffic are not stored in the cache, so the spoofed traffic requires the involvement of the mirror to conduct the filtering. Nevertheless, most legitimate traffic is still processed in the data plane cache, guaranteeing the low latency and line-rate for legitimate traffic.

Process latency. Table 6 shows that NetHCF adds negligible latency for most legitimate packets under both states. In particular, the *extra* delay of NetHCF for processing packets in the cache is just tens of nanoseconds, while for the mirror it needs hundreds of microseconds. Actually, in a typical gateway (or ToR switch), 256 K items are sufficient to serve for almost all the concurrent legitimate IPs [9], and the IPs forwarded to the mirror would most be unfamiliar or malicious. Besides, this overhead only happens when NetHCF is in the filtering state, i.e., under attacks, which only occurs infrequently. In conclusion, the average latency for legitimate traffic is far smaller than that of the existing HCF scheme, which benefit latency-sensitive applications in today's data centers significantly.

TABLE 5
Resource Utilization

	IP2HC Inspecting	*TCP Session Monitoring*	*Cache Statistic*	**Total**
Computing				
Tables	6.77%	6.25%	3.13%	**16.15%**
sALUs	2.08%	12.50%	4.17%	**18.75%**
HashBits	4.63%	3.41%	0.56%	**8.59%**
VLIWs	2.73%	2.73%	0.26%	**5.73%**
Memory				
SRAM	15.57%	5.83%	1.98%	**23.39%**
TCAM	22.92%	0.70%	0.00%	**23.61%**

7 Related Work

Besides the most relevant anti-spoofing works discussed in Section Section 2.1, our work is also inspired by the following topics.

Programmable switch acceleration. Researchers have explored various approaches to leverage programmable hardware switches to accelerate various applications in networking [9], [64], [65], [66], distributed systems [10], [67] and security [68], [69]. These applications achieve much better performance with lower costs than counterparts implemented on commodity servers. Different from these works, NetHCF uses switching ASICs to achieve a different goal,

TABLE 6
Latency Overhead

Processing Path	**Processing Latency**
L2/L3 Routing	0.256 μs
L2/L3 Routing + NetHCF *cache*	0.347 μs
L2/L3 Routing + NetHCF *mirror*	272.983 μs
Original HCF	271.579 μs

spoofed IP traffic filtering, and adopts distinct techniques and optimization to achieve our goal.

Hash-based data structures. Hash-based data structures are usually used to fast access the data with low memory costs. Examples include bloom filter [70], counting bloom filter [71], invertible bloom filter [72], count-min sketch [73], cuckoo hashing [74], d-left hashing [75], and etc. These data structures are becoming more and more popular under the programmable hardware switch scenarios [66], [76], as both the memory and time budget for packet processing are limited. NETHCF uses some of these data structures to save the memory, and also designs plenty of other unique techniques and optimizations to achieve our goal.

IP route caching. There are earlier works on traditional IP route caching [77], [78], [79], [80], which propose to store a subset of forwarding rules in the forwarding table and store the rest in inexpensive slow memory. Most of these works share the observation that IP traffic exhibits both temporal and spatial locality for route caching, which also partly inspires our NETHCF design. However, different from these existing works, NETHCF has a main focus on adversary scenarios to mitigate fake hot phenomenon, which has not been covered in these previous works.

FIB aggregation. To overcome the routing scalability problem, a long body of researches have been devoted to FIB aggregation [81], [82], [83], [84], [85], [86]. In particular, there are fast algorithms for optimal FIB aggregation, e.g., ORTC [81], EAR [85], and there are also online algorithms, e.g., SMALTA [84], FIFA [86], to support fast incremental updates with minor sacrifices on compression effectiveness. Inspired by these works, we adopt the IP2HC mapping table aggregation techniques in NETHCF to store more entries in the cache. However, these solutions can not adapt to legitimate hop-count changes and IP popularity changes in low cost, which is a main contribution of our paper.

8 CONCLUSION

In this paper, we identify the new opportunity to improve the current spoofed packet filtering practice using programmable switching ASICs, and propose NETHCF, a line-rate in-network spoofed packet filtering system. We decouple the existing HCF into two parts, aggregate the IP2HC mapping table as much as possible to cache more entries in the data plane cache, and design several effective mechanisms to make NETHCF adapt to end-to-end routing changes, IP popularity changes, and network activity dynamics. We implement a prototype of NETHCF in the state-of-the-art Barefoot Tofino switch and conduct extensive experiments. Evaluations demonstrate that NETHCF can achieve line-rate and adaptive spoofed IP packet filtering with only minimal overheads.

REFERENCES

[1] G. Li *et al.*, "NETHCF: Enabling line-rate and adaptive spoofed IP traffic filtering," in *Proc. IEEE 27th Int. Conf. Netw. Protocols*, 2019, pp. 1–12.

[2] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 39–53, 2004.

[3] Cloudflare, "The real cause of large DDoS - IP spoofing," 2018. Accessed: Oct. 11, 2018. [Online]. Available: https://blog.cloudflare.com/the-root-cause-of-large-ddos-ip-spoofing/

[4] CAIDA, "State of IP spoofing," 2018. Accessed: Oct. 16, 2018. [Online]. Available: https://spoofer.caida.org/summary.php

[5] C. Jin, H. Wang, and K. G. Shin, "Hop-count filtering: An effective defense against spoofed DDoS traffic," in *Proc. 10th ACM Conf. Comput. Commun. Secur.*, 2003, pp. 30–41.

[6] H. Wang, C. Jin, and K. G. Shin, "Defense against spoofed IP traffic using hop-count filtering," *IEEE/ACM Trans. Netw.*, vol. 15, no. 1, pp. 40–53, Feb. 2007.

[7] B. Networks, "Tofino: World's fastest P4-programmable ethernet switch ASICs," 2018. Accessed: Oct. 13, 2018. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

[8] XPliant, "XPliant ethernet switch product family," 2018, Accessed: Oct. 19, 2018. [Online]. Available: https://www.cavium.com/xpliant-ethernet-switch-product-family.html

[9] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 15–28.

[10] X. Jin *et al.*, "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 121–136.

[11] P. X. Gao *et al.*, "Network requirements for resource disaggregation," *Proc. 12th USENIX Symp. Oper. Syst. Des. Implementation*, vol. 16, pp. 249–264, 2016.

[12] Y. Zhu *et al.*, "Congestion control for large-scale RDMA deployments," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 523–536, 2015.

[13] N. Github, "NetHCF," 2019. Accessed: Aug. 19, 2019. [Online]. Available: https://github.com/NetHCF/NetHCF

[14] J. Li, M. Sung, J. Xu, and L. Li, "Large-scale IP traceback in high-speed internet: Practical techniques and theoretical foundation," in *Proc. IEEE Symp. Secur. Privacy*, 2004, pp. 115–129.

[15] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical network support for IP traceback," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 4, pp. 295–306, 2000.

[16] A. C. Snoeren *et al.*, "Hash-based IP traceback," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 3–14, 2001.

[17] D. X. Song and A. Perrig, "Advanced and authenticated marking schemes for IP traceback," *Proc. 20th Annu. Joint Conf. IEEE Comput. Commun. Societies*, vol. 2, pp. 878–886, 2001.

[18] R. Stone *et al.*, "CenterTrack: An IP overlay network for tracking DoS floods," *Proc. USENIX Secur. Symp.*, vol. 21, p. 114, 2000.

[19] J. Ioannidis and S. M. Bellovin, "Implementing pushback: Router-based defense against DDoS attacks," *Proc. Netw. Distrib. Syst.*, vol. 2, 2002.

[20] A. D. Keromytis, V. Misra, and D. Rubenstein, "SOS: Secure overlay services," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 61–72, 2002.

[21] J. Li, J. Mirkovic, M. Wang, P. Reiher, and L. Zhang, "SAVE: Source address validity enforcement protocol," *Proc. 21st Annu. Joint Conf. IEEE Comput. Commun. Societies*, vol. 3, pp. 1557–1566, 2002.

[22] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling high bandwidth aggregates in the network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 3, pp. 62–73, 2002.

[23] K. Park and H. Lee, "On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 15–26, 2001.

[24] D. K. 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 Trans. Netw.*, vol. 13, no. 1, pp. 29–42, Feb. 2005.

[25] X. Liu, A. Li, X. Yang, and D. Wetherall, "Passport: Secure and adoptable source authentication," in *Proc. 5th USENIX Symp. Netw. Syst. Des. Implementation*, 2008, pp. 365–378.

[26] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," *Proc. Symp. Oper. Syst. Des. Implementation*, vol. 99, pp. 45–58, 1999.

[27] X. Qie, R. Pang, and L. Peterson, "Defensive programming: Using an annotation toolkit to build DoS-resistant software," *ACM SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 45–60, 2002.

[28] O. Spatscheck and L. L. Peterson, "Defending against denial of service attacks in scout," *Proc. Symp. Oper. Syst. Des. Implementation*, vol. 99, pp. 59–72, 1999.

[29] D. J. Bernstein, "SYN cookies," 2018. Accessed: Oct. 23, 2018. [Online]. Available: https://cr.yp.to/syncookies.html
[30] A. Juels and J. G. Brainard, "Client puzzles: A cryptographic countermeasure against connection depletion attacks," *Proc. Netw. Distrib. Syst. Secur. Symp.*, vol. 99, pp. 151–165, 1999.
[31] X. Wang and M. K. Reiter, "Defending against denial-of-service attacks with puzzle auctions," in *Proc. IEEE Symp. Secur. Privacy*, 2003, pp. 78–92.
[32] Subin, "Default TTL (time to live) values of different OS," 2019. Accessed: Aug. 19, 2019. [Online]. Available: https://subinsb.com/default-device-ttl-values/
[33] P. Patel *et al.*, "Ananta: Cloud scale load balancing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 207–218, 2013.
[34] D. E. Eisenbud *et al.*, "Maglev: A fast and reliable software network load balancer," in *Proc. 13th USENIX Symp. Netw. Syst. Des. Implementation*, 2016, pp. 523–535.
[35] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 97–112.
[36] J. Sherry *et al.*, "Rollback-recovery for middleboxes," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 227–240, 2015.
[37] M. Zhang *et al.*, "Tripod: Towards a scalable, efficient and resilient cloud gateway," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 570–585, Mar. 2019.
[38] CAIDA, "The CAIDA anonymized internet traces 2016 dataset," 2018. Accessed: Oct. 19, 2018. [Online]. Available: https://www.caida.org/data/passive/passive_2016_dataset.xml
[39] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 203–216.
[40] A. T. ARTICLES, "5 most famous DDoS attacks," 2018. Accessed: Jan. 19, 2019. [Online]. Available: https://www.a10networks.com/resources/articles/5-most-famous-ddos-attacks
[41] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 27–38, 2015.
[42] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, 2013.
[43] B. Networks, "Second-generation of world's fastest P4-programmable ethernet switch ASICs," 2019. Accessed: Mar. 13, 2019. [Online]. Available: https://barefootnetworks.com/products/brief-tofino-2/
[44] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.
[45] A. Sivaraman *et al.*, "Packet transactions: High-level programming for line-rate switches," in *Proc. ACM SIGCOMM Conf.*, ACM, 2016, pp. 15–28.
[46] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with* flow," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 823–835.
[47] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "TurboFlow: Information rich flow record generation on commodity switches," in *Proc. 13th EuroSys Conf.*, 2018, p. 11.
[48] J. Bai, J. Bi, M. Zhang, and G. Li, "Filtering spoofed IP traffic using switching ASICs," in *Proc. ACM SIGCOMM Conf. Posters Demos*, 2018, pp. 51–53.
[49] V. Paxson, "Empirically derived analytic models of wide-area TCP connections," *IEEE/ACM Trans. Netw.*, vol. 2, no. 4, pp. 316–336, Aug. 1994.
[50] M. E. Crovella and A. Bestavros, "Self-similarity in world wide web traffic: Evidence and possible causes," *IEEE/ACM Trans. Netw.*, vol. 5, no. 6, pp. 835–846, Dec. 1997.
[51] R. Jaiswal, S. Lokhandes, A. Bakre, and K. Gutte, "Performance analysis of IPv4 and IPv6 internet traffic," *ICTACT J. Commun Tec*, vol. 6, no. 4, pp. 1208–1217, 2015.
[52] K. Qian *et al.*, "FlexGate: High-performance heterogeneous gateway in data centers," in *Proc. 3rd Asia-Pacific Workshop Netw.*, 2019, pp. 36–42.
[53] V. Paxson, "End-to-end routing behavior in the internet," *IEEE/ACM Trans. Netw.*, vol. 5, no. 5, pp. 601–615, Oct. 1997.
[54] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang, "BGP routing stability of popular destinations," in *Proc. 2nd ACM SIGCOMM Workshop Internet Measurment*, 2002, pp. 197–202.
[55] Wikipedia, "SYN flood," 2020. Accessed: Apr. 3, 2021. [Online]. Available: https://en.wikipedia.org/wiki/SYN_flood
[56] G. IPv6, "Google IPv6 adoption statistics," 2019. Accessed: Mar. 13, 2019. [Online]. Available: https://www.google.com/intl/en/ipv6/statistics.html
[57] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks," *Proc. Symp. Oper. Syst. Des. Implementation*, vol. 10, pp. 1–6, Oct. 2010.
[58] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic external memory for switch data planes," in *Proc. 17th ACM Workshop Hot Topics Netw.*, 2018, pp. 1–7.
[59] A. Mahimkar, J. Dange, V. Shmatikov, H. M. Vin, and Y. Zhang, "dFence: Transparent network-based denial of service mitigation," *Proc. Conf. Networked Syst. Des. Implementation*, vol. 7, pp. 327–340, 2007.
[60] Y. Afek, A. Bremler-Barr, and L. Shafir, "Network anti-spoofing with SDN data plane," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.
[61] B. Networks, "P4 studio: The next generation software development environment," 2019. Accessed: Oct. 13, 2018. [Online]. Available: https://www.barefootnetworks.com/products/brief-p4-studio/
[62] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. Newton, MA, USA: O'Reilly Media, Inc., 2013.
[63] T. L. F. Project, "Developer quick start guide," 2019. Accessed: Oct. 13, 2018. [Online]. Available: https://www.dpdk.org/
[64] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 161–176.
[65] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 85–98.
[66] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 357–371.
[67] X. Jin *et al.*, "NetChain: Scale-free sub-RTT coordination," in *Proc. 15th USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 35–49.
[68] R. Meier, P. Tsankov, V. Lenders, L. Vanbever, and M. Vechev, "NetHide: Secure and practical network topology obfuscation," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 693–709.
[69] M. Zhang *et al.*, "Poseidon: Mitigating volumetric DDoS attacks with programmable switches," in *Proc. 27th Netw. Distrib. Syst. Secur. Symp.*, 2020.
[70] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[71] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *Proc. Eur. Symp. Algorithms*, Springer, 2006, pp. 684–695.
[72] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the difference? Efficient set reconciliation without prior context," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 218–229, 2011.
[73] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
[74] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
[75] B. Vöcking, "How asymmetry helps load balancing," *J. ACM*, vol. 50, no. 4, pp. 568–589, 2003.
[76] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better netflow for data centers," in *Proc. 13th USENIX Symp. Netw. Syst. Des. Implementation*, 2016, pp. 311–324.
[77] D. C. Feldmeier, "Improving gateway performance with a routing-table cache," in *Proc. 7th Annu. Joint Conf. IEEE Comput. Commun. Societies. Netw.: Evol. Revolution?*, 1988, pp. 298–307.
[78] C. Kim, M. Caesar, A. Gerber, and J. Rexford, "Revisiting route caching: The world should be flat," in *Proc. Int. Conf. Passive Act. Netw. Meas.*, Springer, 2009, pp. 3–12.
[79] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging zipf's law for traffic offloading," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 1, pp. 16–22, 2012.
[80] M. Bienkowski, J. Marcinkowski, M. Pacut, S. Schmid, and A. Spyra, "Online tree caching," in *Proc. 29th ACM Symp. Parallelism Algorithms Architectures*, 2017, pp. 329–338.

[81] R. Draves, C. King, S. Venkatachary, and B. D. Zill, "Constructing optimal IP routing tables," *Proc. Annu. Joint Conf. IEEE Comput. Commun. Soc.*, vol. 99, pp. 88–97, 1999.
[82] B. Cain, "Auto aggregation method for IP prefix/length pairs," U.S. Patent 6,401,130, Jun. 4, 2002.
[83] X. Zhao, Y. Liu, L. Wang, and B. Zhang, "On the aggregatability of router forwarding tables," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.
[84] Z. A. Uzmi *et al.*, "SMALTA: Practical and near-optimal FIB aggregation," in *Proc. 7th Conf. Emerg. Netw. Exp. Technol.*, 2011, p. 29.
[85] T. Yang *et al.*, "Approaching optimal compression with fast update for large scale routing tables," in *Proc. IEEE 20th Int. Workshop Qual. Serv.*, 2012, p. 32.
[86] Y. Liu, B. Zhang, and L. Wang, "FIFA: Fast incremental FIB aggregation," in *Proc. IEEE INFOCOM*, 2013, pp. 1–9.

Menghao Zhang received the BS and PhD degrees in computer science from Tsinghua University, China, in 2016 and 2021, respectively. He is currently a joint postdoctoral with Tsinghua University and Kuaishou Technology. His research interests include programmable network, high-performance network, and network security.

Guanyu Li received the BS degree from the School of Compute Science & Technology, Huazhong University of Science & Technology, China. He is currently wroking toward the PhD degree with the Institute for Network Science and Cyberspace, Tsinghua University. His research interests include software-defined networking, network function virtualization, and cyber security.

Xiao Kong received the B.S. degree in computer science & technology from Nankai University. He is currently working toward the MS degree with the Institute for Network Science and Cyberspace, Tsinghua University. His research interests include software-defined networking, network function virtualization, and cyber security.

Chang Liu received the BS degree from the School of Computer Science & Technology, Beijing Institute of Technology, China. He is currently working toward the PhD degree with the Institute for Network Science and Cyberspace, Tsinghua University. His research interests include software-defined networking, programmable data plane, and cyber security.

Mingwei Xu received the BS and PhD degrees from Tsinghua University. He is currently a full professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include computer network architecture, high-speed router architecture, and network security.

Guofei Gu received the PhD degree in computer science from the College of Computing, Georgia Tech in 2008. He is currently a professor with the Department of Computer Science & Engineering, Texas A&M University. His research interests include network and systems security, such as malware and APT defense, software-defined networking (SDN/NFV) security, mobile and IoT security, and intrusion/anomaly detection.

Jianping Wu (Fellow, IEEE) received the BS, MS, and PhD degrees from Tsinghua University, Beijing, China. He is currently a full professor and the director of the Network Research Center and a PhD supervisor with the Department of Computer Science and Technology, Tsinghua University. Since 1994, he has been in charge of China Education and Research Network. His research interests include the next-generation Internet, IPv6 deployment and technologies, and Internet protocol design and engineering.