NETHCF: Enabling Line-rate and Adaptive Spoofed IP Traffic Filtering

Guanyu Li*, Menghao Zhang*, Chang Liu*, Xiao Kong*, Ang Chen†, Guofei Gu‡, Haixin Duan*
*Institute for Network Sciences and Cyberspace, Tsinghua University
*Department of Computer Science and Technology, Tsinghua University
*Beijing National Research Center for Information Science and Technology (BNRist)
†Department of Computer Science, Rice University
‡Department of Computer Science and Engineering, Texas A&M University

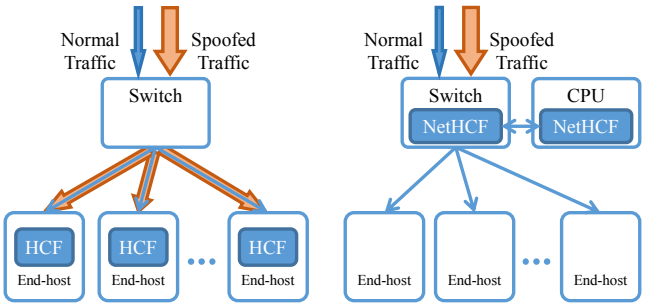
***Abstract*—In this paper, we design NETHCF, a line-rate in-network system for filtering spoofed traffic. NETHCF leverages the opportunity provided by programmable switches to design a novel defense against spoofed IP traffic, and it is highly efficient and adaptive. One key challenge stems from the restrictions of the computational model and memory resources of programmable switches. We address this by decomposing the HCF system into two complementary components—one component for the data plane and another for the control plane. We also aggregate the IP-to-Hop-Count (IP2HC) mapping table for efficient memory usage, and design adaptive mechanisms to handle end-to-end routing changes, IP popularity changes, and network activity dynamics. We have built a prototype on a hardware Tofino switch, and our evaluation demonstrates that NETHCF can achieve line-rate and adaptive traffic filtering with low overheads.**

I. INTRODUCTION

Spoofed IP traffic remains a significant threat to the Internet, and such traffic typically originates from malicious network activities, especially Distributed Denial of Service (DDoS) attacks [1]. Although existing work has studied traffic spoofing extensively, spoofing attacks are still prevalent and frequently reported in the news [2]. According to the Spoofer Project of CAIDA [3], 24.4% of the Autonomous Systems (ASes) have not deployed any countermeasure to disable spoofed IP traffic, and around 16.1% IP addresses in the Internet can be spoofed. Furthermore, these addresses can be found world-wide.

Although attackers can forge any field in the IP header, they cannot easily control the number of hops an IP packet takes to reach its destination. This is because the number of traversed hops depends on the underlying network paths and routing mechanisms. Driven by this observation, previous works [4], [5] propose hop-count based defense mechanisms called *Hop-Count Filtering* (HCF). An HCF defense 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 mapping table is only updated by legitimate packets—for instance, we would 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.

978-1-7281-2700-2/19/$31.00 2019 © IEEE



(a) Existing HCF Scheme (b) NETHCF

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

State-of-the-art HCF filtering mechanisms [4], [5] 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 at the end hosts. 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 (§II-B). 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 [6], [7] provides an exciting opportunity to re-think this design. Programmable switches can run attack detection and mitigation algorithms inside the network, which we can leverage to design an in-network defense that runs at line rate. Since the programmable switching ASICs (Application-Specific Integrated Circuits) can easily process a few billion packets per second (∼Tbps throughput) [8], [9], one switch could match the packet processing power of tens to hundreds of servers. This would also free up the available computation and memory resources on end servers for application-level tasks. Besides, an in-network system can filter spoofed traffic early, avoiding unnecessary bandwidth waste. A switch-based design also has a network-wide view of the traffic space, so it could set up the full mapping table much faster than host-based designs. Note

that even by implementing the HCF scheme in server-based middleboxes, some advantages such as low latency and jitter that are crucial to latency-sensitive applications [10], [11] in today's data centers are hard to achieve; the key reason lies in the notable performance gap between switching ASICs and general computation resources (e.g., CPU).

However, applying traditional HCF techniques to 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. For a HCF defense to take effect, the switch must maintain a correct and up-to-date IP2HC mapping table for to be looked up. Besides, it should also adapt to network activity dynamics (e.g., attacks) quickly. However, switching ASICs have limited on-chip memory and a restrictive computational model, which makes it challenging to *store the full IP2HC mapping table*, *update the table in a timely manner*, and *adapt to network dynamics quickly*.

To address the challenges above, we present NETHCF, an in-network spoofed traffic filtering system. As shown in Figure 1, different from traditional HCF designs, we decouple the HCF defense into two complementary components. The data plane *cache* runs on on switching ASICs, and the control plane *mirror* has access to general-purpose computing resources. The cache serves the "hot keys" (i.e., the legitimate packets), at line rate in the data plane. The mirror processes the packets that miss the cache, maintains the IP2HC mapping table, and adjusts the state of NETHCF to adapt to network dynamics. These two components work with each other to overcome their limitations, achieving both correctness and efficiency. We have implemented a prototype of NETHCF on a Barefoot Tofino switch [6]. Our prototype and evaluation demonstrate that NETHCF can achieve line-rate and adaptive spoofed traffic filtering with only minimal overheads.

In summary, our contributions in this paper include:

- We analyze the limitations of the traditional HCF designs, and identify new opportunities for a novel re-design on programmable switching ASICs (§II).
- We propose NETHCF, a line-rate in-network spoofed traffic filtering system. We decouple the traditional HCF system into two complementary components, the data plane cache and the control plane mirror. NETHCF also aggregates the IP2HC mapping table to cache more entries in the data plane, and designs several mechanisms to adapt to end-to-end routing changes, IP popularity changes, and network activity dynamics (§III).
- We have implemented a prototype of NETHCF, and conducted extensive evaluations to show that NETHCF is highly effective with negligible overheads (§V, §VI).

We then discuss several issues in §IV, describe related works in §VII and conclude this paper in §VIII.

II. BACKGROUND AND MOTIVATION

In this section, we present more background on spoofed packet filtering techniques and the HCF scheme, describe the disadvantages of the existing HCF designs, and discuss why programmable switches provide new opportunities.

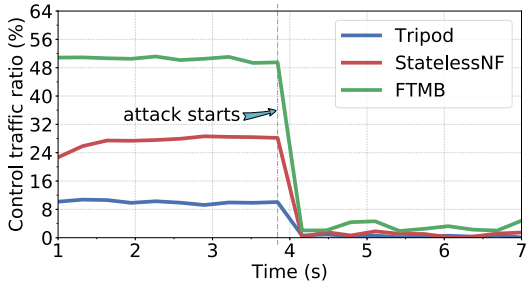


Figure 2: End host-based HCF leads to bandwidth waste.

A. Background on HCF

Source address spoofing is among one of the most serious problems that plague the Internet. Existing defenses can be categorized into *router-based* and *host-based* defenses. Router-based approaches install defense mechanisms inside routers to trace the sources of attacks [12], [13], [14], [15], [16], or detect and block the attack traffic in a coordinated manner [17], [18], [19], [20], [21], [22], [23]. However, these solutions require not only modifications to the routers, but also coordination across routers or even networks. In contrast, host-based approaches can be deployed much easier. End systems, such as edge networks, small ISPs, data centers, or enterprises, also have a natural incentive to deploy defense mechanisms against spoofed traffic. Host-based approaches use sophisticated source-discrimination schemes [24], [25], [26], or reduce the resource consumption of each request [27], [28], [29] to mitigate attacks.

HCF [4], [5] is a simple but effective host-based solution to this problem. It can validate incoming IP packets at an Internet server without any cryptographic operations or router modifications, making it lightweight and practical. HCF also effectively breaks the cost asymmetry between attackers and victims—victims can easily build up legitimate IP2HC mappings for the defense, whereas attackers cannot easily obtain the mapping between arbitrary IP address and its hop count to the victims. Specifically, 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 destinations. Since these hop counts are determined by the Internet routing infrastructure, this also makes it difficult for attackers to use correct hop-count values in their attack traffic.

B. Problem Statement

Existing HCF designs are all located at the end hosts. For instance, the original HCF proposal runs the defense inside the Linux kernel and integrates it with the network stack. However, there are several practical problems that hinder the effectiveness of this design point.

First, existing HCF suffers from bandwidth waste. Spoofed IP traffic cannot be filtered until it arrives at the end hosts. This still consumes bandwidth resource of the victim networks and cause performance drops. Such drops are not only to Internet traffic but also west-east traffic across different servers (e.g., inter-service or control traffic) [30], [31], [32], [33], [34].

Table I: Resource consumptions of the original HCF scheme.

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

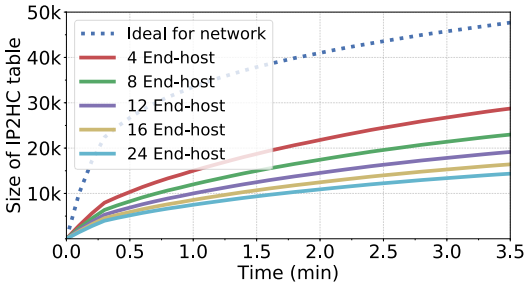
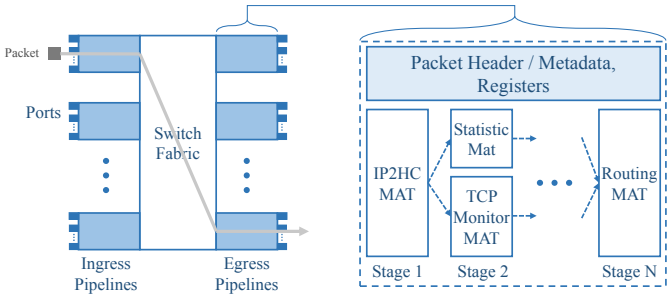


Figure 3: Slow setup of the original HCF design.



(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

Figure 4: Programmable switch data plane.

To validate this claim, we select three systems—Tripod [34], StatelessNF [32] and FTMB [33]—for an experiment. In these systems, each server runs some network functions (NFs), and NF states are replicated among different servers to maintain high availability. The HCF kernel module is deployed on each server for defense. Figure 2 shows that, when spoofed traffic arrives, although each server can filter most spoofed packets, these packets have already consumed the network bandwidths. This leads to network congestion and packet loss, which has a detrimental effect on the deployed systems.

Second, existing HCF needs to be deployed on each end host to conduct spoofed IP packet filtering, which not only leads to deployment redundancy but also consumes general-purpose computing resources (e.g., CPU, memory) at the end hosts. This takes away valuable resources from application-level tasks. We measure the resource consumption of the HCF kernel module on the server, and show the results in Table I. The server is equipped with a 40Gbps 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 frequency has already reached its maximum. When the NIC load is 75%, the HCF kernel module (excluding the normal protocol stack) uses up an entire CPU core. In addition, the HCF kernel module also requires 1.98 GB memory to store the IP2HC table. Moreover, each server has to be equipped with the full HCF kernel with the same CPU and memory consumption redundantly.

Third, with the rapid growth of network traffic, modern data centers often use load balancing to distribute large volumes of traffic to clusters of servers [30], [31], [8]. Therefore, each end host can only see a portion of incoming traffic, which makes setting up a full IP2HC mapping table extremely slow. This in turn would delay the effect of the HCF protection. In Figure 3, the blue dotted line shows the ideal IP2HC setup time if a defense sees the full traffic trace of the entire network; the solid lines show the IP2HC setup time on each end host, when the traffic is load-balanced to different numbers of hosts. As we can see, as the number of hosts increases, the speed of setting up a IP2HC table on the host becomes slower, and this will greatly delay the time for HCF to take effect.

An intuitive solution to address the problems above is to implement HCF in in-network, server-based middleboxes. However, software-based HCF has two fundamental limitations. First, software-based packet processing has limited capacity, usually processing ∼10 Gbps traffic or ∼M packets per second [35]. We can scale out the packet processing capacity by adding more servers, but doing so also increases costs and operational complexity. For example, handling a typical attack traffic volume (∼Tbps) [36] would require hundreds of servers. Second, processing packets in software incurs high latency and jitter, and it provides poor performance isolation. Software processing adds a latency of 50us–1ms when handling as few as 100K packets per second [37], [8]. This is problematic for latency-sensitive applications [10], [11] in data centers today. When software experiences a flash crowd, legitimate traffic served by the software would also experience increased delays, or unexpected packet drops, which makes things even worse.

C. Programmable Switches

Recent developments in Software-Defined Networking (SDN) have resulted in programmable switching ASICs [38], [6], [39] and domain-specific languages (e.g., P4 [40]) to extend network programmability from the control plane to the data plane. Compared to traditional fixed-function switches, emerging programmable switches offer hardware programmability without sacrificing performance. They also have similar levels of power consumption and price with regular switches [6]. The new hardware provides unique opportunities to overcome the shortcomings of the traditional HCF design.

There are multiple ingress and egress pipelines in programmable switches, and each has multiple ingress and egress ports (Figure 4(a)). When packet reaches one of the ingress ports, it is first processed by the ingress pipeline, then switched

to one of the egress pipelines to be processed, and finally sent to the specified egress port. Inside a pipeline, packets are processed sequentially in each stage (Figure 4(b)), which has its own dedicated resources such as match-action tables and registers. Match-action tables match on certain header fields or metadata of the packets, and perform programmable actions (e.g., modifying header fields/metadata, read/write registers or drop packets) based on the match results (Figure 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 [40], developers can customize the data plane logic. To do this, programmers can write P4 programs to define packet headers, build packet processing graphs, and specify the match fields and actions of each table. The compiler provided by switch vendors can compile the programs to binaries and generate interactive APIs. The binaries are loaded into the data plane, and the APIs are used by control plane applications to interact with the data plane.

The programmable switching ASICs and P4 language make it easy to implement customized packet processing logic at Tbps. One constraint here is that the logic needs to fit 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 [41], [9], [42], [43]: 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 constraints as well, 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 to perform a simple function, such as simultaneous read/write, conditional updates, and basic arithmetic operations.

To summarize, programmable switching ASICs can process packets with high throughput, low latency and jitter, and high performance isolation, which provide unprecedented opportunities for a better HCF design. But on the other hand, they have their own limitations in the computational model and on-chip resources. We must get around these limitations and come up with careful designs to achieve our goal: serving the legitimate traffic with low latency and filter the spoofed traffic effectively.

III. Our Approach: NetHCF

A. NetHCF *Overview*

While the on-chip memory size (TCAM and SRAM) in the switching ASICs (50-100MB [8]) has grown rapidly in recent years, it is still challenging to store the whole IP2HC mapping table directly in the switching ASICs. A typical IP2HC mapping entry should store the mapping from IP (32 bit) to hop-count (5 bit), thus simply storing the entire IP2HC mapping table requires at least $2^{32} \times (32+5)$ bits ($\sim$10 GB) in the switching ASICs. An intuitive approach would be to store a small hash of the match field (IP) and handle hash collisions using special processing. Although techniques for resolving hash collisions exist (e.g., as in SilkRoad [8]), it is still impossible to eliminate all hash collisions, which would result in disruption of legitimate flows. As most malicious flows do not have to follow normal TCP state transitions, collisions will become a much worse problem. Moreover, even after hashing, storing all the IP2HC mapping items would still be 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 [44]. But if an incoming packet cannot find its IP in the IP2HC mapping table on the switch, then handling this packet would become a problem.

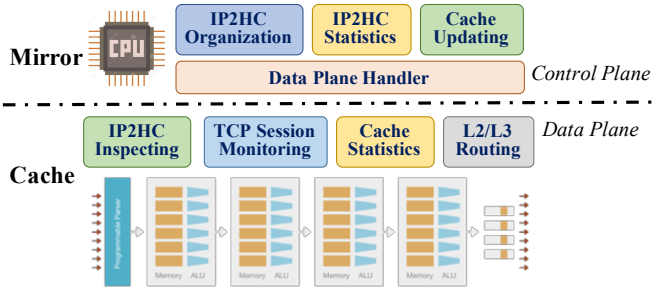


Figure 5: The NetHCF architecture.

Our approach is to decouple the existing HCF design into two parts, an HCF *cache* in the data plane (programmable switching ASICs), and an HCF *mirror* in the control plane (general-purpose computing resources, e.g., in server clusters). Figure 5 shows this architecture. The data plane cache serves most active and legitimate IPs at line rate. The control plane mirror handles the remaining IPs with slower and more complex logic, maintains the IP2HC mapping table, and updates the state of NetHCF to adapt to network dynamics. These two parts interact with each other through a set of coordination mechanisms to achieve both the advantages of programmable switching ASICs (high performance) and those of the general computation resources (high flexibility).

The data plane cache handles the most frequent IPs in the data plane 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 these IPs at line rate, a *TCP Session Monitoring* module which captures the legitimate hop-count values and updates these IP2HC items if necessary, and a *Cache Statistics* module which maintains real-time statistics to maintain counters for each cached IP2HC mapping item, and reports hot items to the control plane for cache updates.

[1]Silkroad [8] compresses both match field and action data of each entry, and it can only store $\sim$10M entries. This number is still far smaller than the entire 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.

As a complementary component, the control plane mirror maintains a full view of IP2HC mapping and IP2HC statistics. It also plays an important role in aggregating IP2HC entries and handling the packets whose IPs miss the cache. More critically, it is responsible for updating the state of NETHCF to adapt to network dynamics, i.e., routing changes, IP popularity changes, and network activity dynamics.

When NETHCF runs for the first time, the network operator 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 achieve a high coverage of the entire IP space. The duration for this would depend on the amount and diversity of traffic the victim may receive. For example, for popular sites, a few days could be sufficient, while for lightly loaded sites, a few weeks might be needed. 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 (§III-B). Meanwhile, the control plane mirror would constantly insert new legitimate IP2HC items into the data plane cache until the number of items in the data plane cache becomes relatively stable.

After this initialization, NETHCF would continue adding new entries to the IP2HC mapping table when previously unseen IP addresses arrive (by sending *packet digests* to the control plane). More importantly, at this relatively stable running state, NETHCF needs to adapt to end-to-end routing changes, IP popularity changes, and network activity dynamics (§III-C). First, NETHCF should capture legitimate hop-count changes and update the corresponding IP2HC mapping items resulting from end-to-end routing changes in the Internet. Second, NETHCF should accommodate the dynamic incoming traffic and ensure that the hottest IPs are always stored in the cache. Third, to minimize collateral damage and adapt to attacks, NETHCF has two running states, a *learning* state when packets with wrong hop-count are forwarded 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.

B. 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 memory and further stress the memory bottleneck. We observe that many IP addresses with the same prefix share the same hop-count values, e.g., addresses in the same subnet. Therefore, we can 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.

To achieve an efficient and correct aggregation, we represent the IP2HC mapping table in a binary tree, as shown in Figure 6. Each leaf in the tree represents a valid IP address, while the other nodes represent specific IP address prefixes. Each node in the tree has three attributes, indicating whether it is a representative node, its "hotness", and its hop-count value. We define the representative node as a basic item/entry for the IP2HC mapping table, and only the representative 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 on 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 representative 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 representative nodes, so we can store some of them in the data plane cache independently.

C. Adapting To Network Dynamics

NETHCF should accommodate end-to-end routing changes, which would affect legitimate hop-count values. And it should also adapt to traffic dynamics, since the popularity of IPs may change over time. Besides, NETHCF should also minimize potential collateral damage, so legitimate packets do not go through control plane processing unnecessarily. To achieve the first goal, NETHCF should capture the up-to-date IP2HC varieties timely and update the items at both the data plane cache and the control plane mirror immediately. To achieve the second, NETHCF should capture the IP access statistics and change the IP2HC mapping items in the data plane cache to ensure cache hotness. For the third goal, we use two running states to make NETHCF adapt to the attacks.

1) Capturing legitimate hop-count changes: Although hop-counts have been shown to be relatively stable [45], [46], [5], there are still cases when hop-counts may change, such as due to routing instability and network address reallocation. These changes should be captured as soon as possible to update the IP2HC mapping in both the cache and the mirror. First, the new IP2HC mapping information should be reported to update the IP2HC mapping table in the control plane mirror immediately. As shown in Algorithm 1, if the new IP2HC mapping corresponds to a representative node (32 bit prefix), we update it directly and aggregate it with its sibling node iteratively when possible (**Function** `aggregation()`). Otherwise, we split the tree and re-select the new representative nodes (**Function** `split()`).

Second, it is also essential to update the hop-count of the IP2HC mapping items in the data plane cache in a timely manner. A strawman solution is to report the TCP handshake

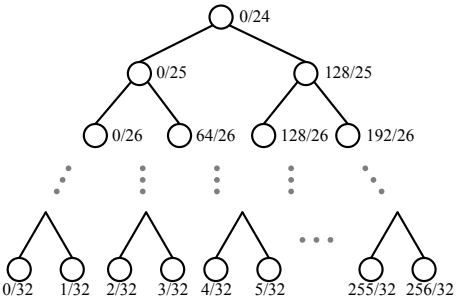


Figure 6: The IP2HC table organization.

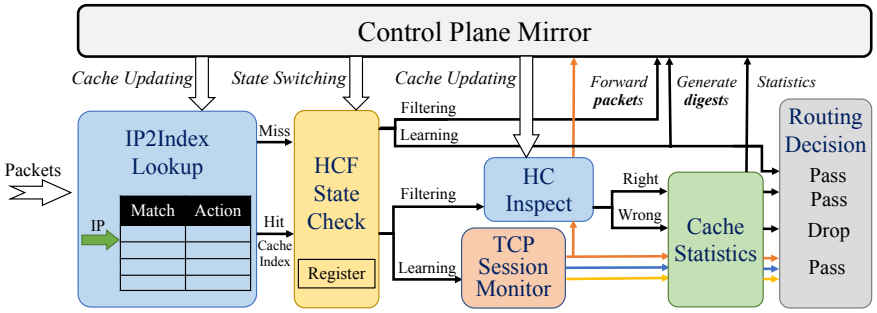


Figure 7: The workflow of the data plane cache.

Algorithm 1: IP2HC Partial 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 partialUpdate(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
```

packets to the mirror and to update the hop-count of the IP2HC mapping item with issued control messages. However, entry insertions at the control plane are not atomic, and they take several milliseconds [8]. This means that an IP address may have many packets arrived before the control plane completes entry insertion into the cache, and these packets may be classified as spoofed packets. If NETHCF simply discards these packets, this would cause a significant performance penalty. To solve this problem, we introduce a *valid* flag and a *temporary bitmap* in the IP2HC inspection module (Figure 8). The TCP Session Monitoring module upstream first monitors the TCP handshake packets using expected TCP state transitions. To prevent pollution from attacks, only packets that follow correct Seq/Ack number transitions are accepted as a legitimate TCP connection[3]. After the handshake completes and the connection establishes, the updated legitimate hop-counts would be encapsulated into metadata and be delivered to the subsequent stage. The subsequent stage at the switch pipeline would update the corresponding hop-count register's valid flag as invalid, and record the new hop-count value in the temporary bitmap immediately. For a packet whose matching entry is invalid, if its hop-count value matches the temporary bitmap, we regard it as valid; and vice versa. In this way, we achieve the line-rate hop-count update in the switching ASICs, avoiding potential race conditions and ensuring the per-IP packet-processing consistency. Note that the hop-count is generally stable, so these cases would be rare and the temporary bitmap could be very small. And all the invalid IP2HC entries in the data plane would be updated in the upcoming update period.

2) Handling IP popularity changes: To cope with traffic dynamics where IP popularity may change, the mirror of NETHCF should frequently update the cache with the hottest IP2HC mapping items. NetCache [9] provides an useful approach to realizing a similar goal. The data plane selects and reports the hot keys from the uncached items with a heavy-hitter detector, and the control plane compares the hits of the heavy-hitter detectors with the counters of the cached items to evict less popular keys and insert more popular keys. However, this methodology does not apply to NETHCF, since NETHCF is facing a more adversarial scenario. Simply adopting this approach would lead to an attack, where legitimate hot IPs in the data plane may be replaced by the fake IPs accessed deliberately by the attackers; this would degrade the performance of the legitimate IPs and packets.

We observe that the main reason that makes this attack possible is that the cache does not have a full view for

[3]One potential disadvantage is that the TCP Session Monitor module may be vulnerable to TCP SYN flooding attacks, but actually spoofed SYN packets replacing the cached legitimate SYN packets is a rare event, since the period for the TCP handshake is very short.

the uncached IP2HC mapping and cannot determine whether uncached IP access is legitimate or not with its hop-count. To handle this, 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 has the full views for the whole IP2HC mapping, it can easily distinguish the fake ones from uncached legitimate IP accesses. In particular, in the mirror, we maintain a hit counter (i.e., hotness) for each legitimate IP and another counter for packets that fail at IP2HC checking. To reduce the communication cost between the data plane and the control plane, only digests (IP address, IP TTL, and TCP flag) instead of the whole packet are delivered to the control plane at the learning state. At the filtering state, this cost is unavoidable. For the control plane 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. As there are hundreds of thousands of items in the cache, it is too expensive. To reduce this overhead, we handle this by sharing the overhead across a period: when the counter of an item is larger than a pre-determined threshold, the cache reports the IP/IP prefix of the item to the mirror. A bitmap is attached to remove duplicate hot item reports. The control plane screens out the unreported items of the cache, and update these unreported items in the data plane with the hottest items from uncached IPs/IP prefixes.

3) Running States of NETHCF*:* Even though we have already offloaded several IP2HC mapping components into the switching ASICs, it is still impossible to offload all of them. As a result, packets that miss the cache must be directed to the mirror for decision (pass/drop), which would cause additional delays 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. By default, NETHCF stays in 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 much smaller than T_1 for better stability, which can avoid the frequent transitions 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 always hold. There

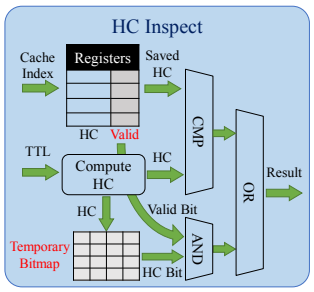


Figure 8: The HC inspect table.

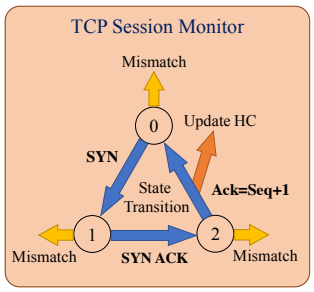


Figure 9: The TCP session monitor module.

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 that legitimate packets from known IP addresses are still served properly during an attack. Certainly, such collateral damage could be extremely low if the IP2HC mapping table becomes more and more complete (i.e., if NETHCF spends more time in the learning state).

D. Putting It All Together

Data plane cache pipeline. The data plane cache is the core component of NETHCF. It mainly hosts (1) an *IP2HC Inspecting* module to inspect the validity of packets, (2) a *TCP Session Monitoring* module to capture the changes of legitimate hop-counts 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 Figure 7. The *IP2HC inspecting* module is located at two separated stages of the pipeline. An *IP2Index lookup table* first matches on the IP address and gives a cache index as the action data. Then the cache index would be used as the index to extract the hop-count value from the *hop-count register array* for packet hop-count checking (Figure 8). A *valid* bit is attached to each entry of hop-count register array to indicate whether this entry is still valid. Along with the hop-count register array, there is a temporary bitmap, which is used to record the valid hop-count values for invalid IPs/IP prefixes. The *TCP Session Monitoring* module consists of two register arrays, one for TCP seq/ack number and the other for TCP state flags. Only flows which strictly conform to legitimate TCP state transitions are allowed to update the hop-count register array (Figure 9). The *Cache Statistics* module is composed of a counter array, a bitmap 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 cache 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 bitmap 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 mapping from IP/IP prefix to hop-count index, which coordinates with the hop-count array to issue the mapped entries for the IP2Index lookup table and the hop-count register array in the data plane cache. It also maintains an IP2HC mapping items management table, which records whether each representative 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 representative 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.

IV. DISCUSSION

NETHCF filters spoofed IP packets with inconsistent hop counts, and it shares a similar threat model as the origin HCF scheme [5]. From the algorithm level, it also shares similar limitations as the original HCF scheme, e.g., it is difficult to handle Network Address Translator (NAT) scenarios, where multiple hop-count values correspond to the same IP address. We also refer interested readers to the original paper on the robustness of NETHCF against various evasion techniques. NETHCF mainly improves the performance of the original end host based HCF scheme with new system-level designs, so we mainly discuss these aspects.

Deployment. In real-world deployment scenarios, the spoofed traffic is usually dispersed and high-volume, which may exceed the capability of one server. To handle this, the control plane mirror can be deployed in a cluster of servers [47], which communicates with the switch control plane agent to update the entries in the data plane cache. Each server in the cluster would host a portion of uncached spoofed IP traffic, which would achieve high scalability. In this way, NETHCF achieves the goals that serving the legitimate traffic with low latency and filtering the spoofed traffic effectively.

Resource constraints. In the testing of our prototype, we find that the main bottleneck does not come from the throughput of the cache/mirror channel. Instead, NETHCF mainly stresses the memory resources. This may become a potential attack vector to our NETHCF system. Fortunately, recent designs have proposed to couple programmable switches with external DRAM in servers to alleviate the resource pressure [48]. This would help NETHCF cache more (even complete) IP2HC mapping entries in the data plane, and mitigate this potential attack.

V. IMPLEMENTATION

We have implemented an open source prototype of NETHCF, including all components of the cache and the mirror described in §III[4].

Table II: Replayed traffic workloads.

#	Traffic	Avg. Flow Length	Avg. Packet Size	Size
1	BigFlow	19.0 packets/flow	451B/packet	1.50GB
2	SmallFlow	3.3 packets/flow	291B/packet	1.60GB
3	Enterprise	9.5 packets/flow	622B/packet	1.74GB

The cache is implemented with ∼1K lines of P4 [40] code for the Barefoot Tofino ASIC [6]. The IP2Index Lookup Table has 256K entries in total, with a sub-table to store aggregated entries and another sub-table for 32-bit un-aggregated IP address. Correspondingly, the size of the hop-count register (5-bit for hop-count value, 1-bit for valid flag) array is also 256K. Since legitimate hop-count changes happen rarely, we set the temporary bitmap with a size of 1024. For the Cache Statistic module, the hits counter (8-bit) array and the filtering bitmap is also 256K. The TCP Session Monitoring Table contains two register arrays, one for TCP flags (2-bit) and another for TCP seq/ack number (4-byte), each with 1M slots. Note that the size of TCP Monitoring Table should be larger than that of IP2HC Table, because each IP may have multiple connections. All of these above result in only a small portion of TCAM and SRAM occupation (§VI-D), leaving enough space for traditional network processing. For TCP Session Monitoring Table, we use built-in hash functions in P4 on both 5-tuple and reverse 5-tuple[5], 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.

The control plane mirror is written in ∼3K lines of Python code. Currently it runs on the switch control plane directly leveraging the general computation resources on the Tofino switch, because we find that our Tofino switch has 8 Intel(R) Pentium(R) CPU D1517 @1.60GHz cores and 8 GB memory, which are sufficient to accommodate the requirements of the basic experiments. As we discussed, one could also implement the control plane mirror in a cluster of servers to achieve high scalability even under high-volume spoofed traffic. We leave this as future work.

VI. EVALUATION

Our evaluation seeks to answer the following key questions:

- How does NETHCF perform compared with the original HCF [5]?
- How effective are the techniques and optimization we have designed in NETHCF?
- What are the resource utilization and overhead of NETHCF?

A. Experimental Setup

Our testbed is composed of one 3.3Tb/s Barefoot Tofino switch and two servers, each of which is equipped with 12

[4]Due to the non-disclosure agreement with Barefoot, we only open the source code of our BMv2 version [49]: https://github.com/NetHCF/NetHCF.

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

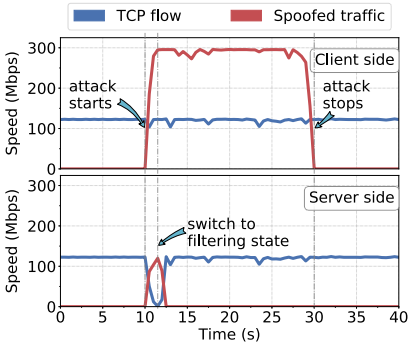


Figure 10: Bandwidth savings with NETHCF.

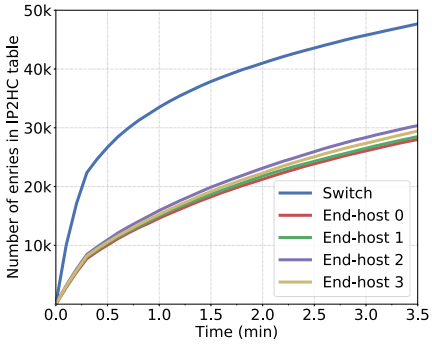


Figure 11: Setup speed for the IP2HC mapping table.

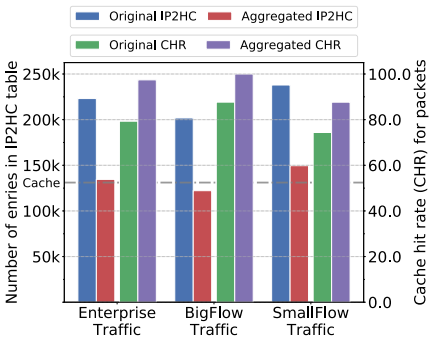


Figure 12: Effectiveness of the IP2HC aggregation mechanism.

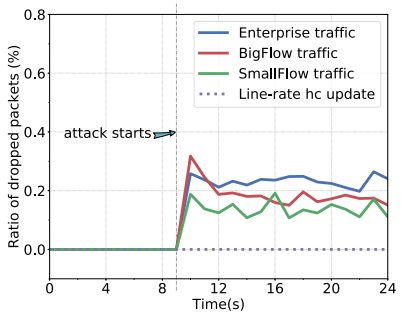


Figure 13: Effectiveness of the line-rate hop-count updating.

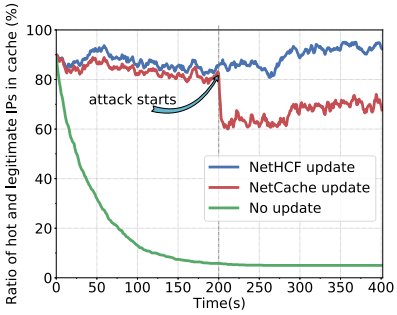


Figure 14: Effectiveness of the NETHCF cache update mechanism.

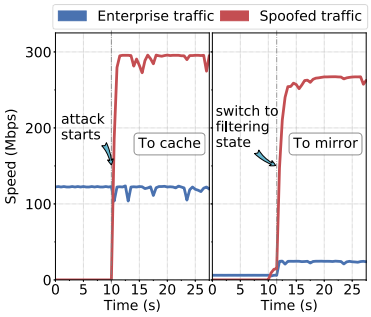


Figure 15: Communication overhead between cache and mirror.

Intel(R) Xeon(R) E5-2698 v4 CPUs and 128GB memory. The two servers are connected to the switch via 40Gbps Intel XL710 NICs. In particular, one server runs an Apache HTTP server with default settings, and the other one is used as the normal 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 [50], including an *Enterprise* traffic trace, a *BigFlow* traffic trace, and a *SmallFlow* traffic trace, for an extensive evaluation on traces with different characteristics (Table II). To simulate an advanced attacker, we generate spoofed traffic whose hop-count distribution follows a Gaussian distribution (μ = 16.5, σ = 4) given in [5], so that more attack packets may happen to have correct hop-counts.

B. Performance Improvement

First, we run `wget` to generate normal TCP traffic and replay spoofed traffic on the client machine simultaneously to show the effectiveness of NETHCF. Figure 10 shows that the server only receives few spoofed packets. This is because NETHCF detects the attack and switches to filtering state to adapt to network activity dynamics immediately. Compared with HCF, by pushing intelligence into the network, NETHCF is able to prevent attack traffic from entering the host network and preserve bandwidth for legitimate packets.

Then, we add three more servers with HCF and replay the workload traffic traces from the client, using ECMP to distribute the traffic to the four servers. The results for the three traces are very similar, so we only show the result for the *Enterprise* traffic trace. As shown in Figure 11, the IP2HC mapping table of NETHCF is set up faster than that of HCF, since NETHCF can view the full traffic space. The total size of IP2HC on four hosts is much larger than that of NETHCF, which means there are plenty of identical entries for IP2HC mapping table on end hosts. In other words, NETHCF can avoid performing duplicate work.

C. Optimization Effectiveness

IP2HC mapping table aggregation. We demonstrate the effectiveness of the IP2HC aggregation technique in Figure 12. For the control plane, aggregation significantly reduces the number of IP2HC entries that need to be maintained, saving considerable memory resources. More importantly, for the data plane, 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

Table III: Resource utilization.

	IP2HC Inspecting	*TCP Session Monitoring*	*Cache Statistic*	**Total**
Computing				
Tables	6.77%	9.38%	2.08%	**18.23%**
sALUs	6.25%	4.16%	10.42%	**20.83%**
HashBits	5.35%	1.62%	0.96%	**7.93%**
VLIWs	4.95%	1.69%	1.30%	**7.94%**
Memory				
SRAM	20.42%	28.65%	2.81%	**51.88%**
TCAM	33.68%	0.00%	0.00%	**33.68%**

server side. As shown in Figure 13, 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 (§III-C1), while line-rate hop-count update of NETHCF avoids this and keeps no packets dropped.

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 the three different cache updating mechanisms, and select the percentage of both legitimate and hot entries in the cache as the metric. Figure 14 demonstrates that the update mechanism of NETHCF performs well all the time, while that of NetCache [9] falls short as a result of attacks (§III-C2). This indicates that NETHCF can adapt well to IP popularity changes even in adversarial scenarios. If we do not update the cache, just few entries for long connections are continuously hot while most entries may not produce a match any more.

D. Micro Benchmarks

Resource utilization. Table III shows the resource usage of NETHCF in our Tofino switch. As we can see, NETHCF occupies less than 20% of computational resources, and uses about one-third of the TCAM and half of the SRAM. This is because the Barefoot Capilano software suite compiles the sub-table (§V) storing aggregated IP2Index entries into TCAM and the sub-table storing un-aggregated entries whose match field is exactly IP address into SRAM. Even with all the data plane components, NETHCF still leaves enough space for traditional network processing, and this can even be further optimized with more tuning.

Communication between mirror and cache. To measure the communication overhead between the two parts of NETHCF, we replay the *Enterprise* traffic as workload and initiate the attack at 10s. We use the bytes of traffic transferred from the cache to the mirror as the metric. As shown in Figure 15, in the learning state (0∼10s), only a small portion of traffic is steered from the cache to the mirror. When the attack starts and NETHCF switches to the filtering state (after 10s), a large portion of spoofed traffic needs to be processed in the mirror.

Table IV: Latency results.

Processing Path	**Processing Latency**
L2/L3 Routing	0.256 μs
L2/L3 Routing + NETHCF *cache*	0.347 μs
L2/L3 Routing + NETHCF *mirror*	27.983 μs
NETHCF with IP2HC Aggregation	30.106 μs
Original HCF	27.579 μs

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.

Processing latency. Table IV 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 tens of microseconds. Actually, in a typical gateway (or ToR switch), 256K items are sufficient to serve for almost all the concurrent legitimate IPs [8], 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.

VII. RELATED WORK

Besides the most relevant anti-spoofing works discussed in Section §II-A, our work is also inspired by the current trends in networking and distributed systems, which leverage programmable switching ASICs to accelerate various applications: layer-4 load balancing [8], [51], key-value store [9], coordination services [52], [53], congestion control and load balancing protocols [54], fast connectivity recovery [55], network monitoring and measurement tasks [43], [42], [56], [57]. 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, spoofed IP traffic filtering, and adopts new techniques and optimization to achieve our goal.

VIII. 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 HCF system into two components, aggregate the IP2HC mapping table 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 have implemented a prototype of NETHCF in a Barefoot Tofino switch and conducted extensive experiments. Evaluations demonstrate that NETHCF can achieve line-rate and adaptive spoofed IP packet filtering with only minimal overheads.

IX. Acknowledgement

We thank anonymous ICNP reviewers for their valuable comments. We also thank Jiasong Bai and Mingwei Xu from Tsinghua University, Bingyang Liu from Huawei Technologies Co. Ltd. for joining some discussions of this paper. Guanyu, Menghao, Chang and Xiao are also sincerely graceful for their former Ph.D. advisor, Jun Bi from Tsinghua University, for his strong support. This work is supported by the National Key R&D Program of China (2017YFB0801701), the National Science Foundation of China (No.61872426) and Tsinghua Scholarship for Overseas Graduate Studies. It is also based upon work supported in part by the National Science Foundation (NSF) under Grant No. 1617985, 1642129, 1700544, and 1740791. Menghao Zhang is the corresponding author.

References

[1] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 39–53, 2004.

[2] Cloudflare, "The real cause of large ddos - ip spoofing," https://blog.cloudflare.com/the-root-cause-of-large-ddos-ip-spoofing/, 2018, [Online; accessed Oct. 11, 2018].

[3] CAIDA, "State of ip spoofing," https://spoofer.caida.org/summary.php, 2018, [Online; accessed Oct. 16, 2018].

[4] C. Jin, H. Wang, and K. G. Shin, "Hop-count filtering: an effective defense against spoofed ddos traffic," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 30–41.

[5] H. Wang, C. Jin, and K. G. Shin, "Defense against spoofed ip traffic using hop-count filtering," *IEEE/ACM Transactions on Networking (ToN)*, vol. 15, no. 1, pp. 40–53, 2007.

[6] B. Networks, "Tofino: World's fastest p4-programmable ethernet switch asics," https://barefootnetworks.com/products/brief-tofino/, 2018, [Online; accessed Oct. 13, 2018].

[7] XPliant, "Xpliant ethernet switch product family," https://www.cavium.com/xpliant-ethernet-switch-product-family.html, 2018, [Online; accessed Oct. 19, 2018].

[8] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 15–28.

[9] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 121–136.

[10] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation." in *OSDI*, vol. 16, 2016, pp. 249–264.

[11] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 523–536.

[12] J. Li, M. Sung, J. Xu, and L. Li, "Large-scale ip traceback in high-speed internet: Practical techniques and theoretical foundation," in *Security and privacy, 2004. Proceedings. 2004 IEEE symposium on*. IEEE, 2004, pp. 115–129.

[13] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical network support for ip traceback," in *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 4. ACM, 2000, pp. 295–306.

[14] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer, "Hash-based ip traceback," in *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4. ACM, 2001, pp. 3–14.

[15] D. X. Song and A. Perrig, "Advanced and authenticated marking schemes for ip traceback," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2. IEEE, 2001, pp. 878–886.

[16] R. Stone *et al.*, "Centertrack: An ip overlay network for tracking dos floods." in *USENIX Security Symposium*, vol. 21, 2000, p. 114.

[17] J. Ioannidis and S. M. Bellovin, "Implementing pushback: Router-based defense against ddos attacks." in *NDSS*, vol. 2, 2002.

[18] A. D. Keromytis, V. Misra, and D. Rubenstein, "Sos: Secure overlay services," in *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 4. ACM, 2002, pp. 61–72.

[19] J. Li, J. Mirkovic, M. Wang, P. Reiher, and L. Zhang, "Save: Source address validity enforcement protocol," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 2002, pp. 1557–1566.

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

[21] K. Park and H. Lee, "On the effectiveness of route-based packet filtering for distributed dos attack prevention in power-law internets," in *ACM SIGCOMM computer communication review*, vol. 31, no. 4. ACM, 2001, pp. 15–26.

[22] 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 Transactions on Networking (TON)*, vol. 13, no. 1, pp. 29–42, 2005.

[23] X. Liu, A. Li, X. Yang, and D. Wetherall, "Passport: secure and adoptable source authentication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2008, pp. 365–378.

[24] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *OSDI*, vol. 99, 1999, pp. 45–58.

[25] X. Qie, R. Pang, and L. Peterson, "Defensive programming: Using an annotation toolkit to build dos-resistant software," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 45–60, 2002.

[26] O. Spatscheck and L. L. Peterson, "Defending against denial of service attacks in scout," in *OSDI*, vol. 99, 1999, pp. 59–72.

[27] D. J. Bernstein, "Syn cookies," https://cr.yp.to/syncookies.html, 2018, [Online; accessed Oct. 23, 2018].

[28] A. Juels and J. G. Brainard, "Client puzzles: A cryptographic countermeasure against connection depletion attacks." in *NDSS*, vol. 99, 1999, pp. 151–165.

[29] X. Wang and M. K. Reiter, "Defending against denial-of-service attacks with puzzle auctions," in *Security and Privacy, 2003. Proceedings. 2003 Symposium on*. IEEE, 2003, pp. 78–92.

[30] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu *et al.*, "Ananta: Cloud scale load balancing," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 207–218.

[31] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 523–535.

[32] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 97–112.

[33] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo *et al.*, "Rollback-recovery for middleboxes," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 227–240.

[34] M. Zhang, J. Bi, K. Gao, Y. Qiao, G. Li, X. Kong, Z. Li, and H. Hu, "Tripod: Towards a scalable, efficient and resilient cloud gateway," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 570–585, 2019.

[35] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of nfv," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 203–216.

[36] A. T. ARTICLES, "5 most famous ddos attacks," https://www.a10networks.com/resources/articles/5-most-famous-ddos-attacks, 2018, [Online; accessed Jan. 19, 2019].

[37] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.

[38] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[39] B. Networks, "Second-generation of worlds fastest p4-programmable ethernet switch asics," https://barefootnetworks.com/products/brief-tofino-2/, 2019, [Online; accessed Mar. 13, 2019].

[40] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[41] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 15–28.

[42] 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 *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018.

[43] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: information rich flow record generation on commodity switches," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 11.

[44] J. Bai, J. Bi, M. Zhang, and G. Li, "Filtering spoofed ip traffic using switching asics," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. ACM, 2018, pp. 51–53.

[45] V. Paxson, "End-to-end routing behavior in the internet," *IEEE/ACM Transactions on Networking (ToN)*, vol. 5, no. 5, pp. 601–615, 1997.

[46] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang, "Bgp routing stability of popular destinations," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 2002, pp. 197–202.

[47] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *OSDI*, vol. 10, 2010, pp. 1–6.

[48] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic external memory for switch data planes," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. ACM, 2018, pp. 1–7.

[49] P. L. Consortium *et al.*, "Behavioral model (bmv2)," 2014, [Online; accessed Oct. 10, 2018].

[50] CAIDA, "The caida anonymized internet traces 2016 dataset," https://www.caida.org/data/passive/passive_2016_dataset.xml, 2018, [Online; accessed Oct. 19, 2018].

[51] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 18, 2018, pp. 125–139.

[52] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018.

[53] J. Li, E. Michael, and D. R. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 104–120.

[54] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation." in *NSDI*, 2017, pp. 67–82.

[55] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 161–176.

[56] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 85–98.

[57] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: query-driven streaming network telemetry," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 357–371.