TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 06/15 pp452–463
DOI: 10.26599/TST.2020.9010018
Volume 26, Number 4, August 2021

Trident: Efficient and Practical Software Network Monitoring

Xiaohe Hu, Yang Xiang, Yifan Li, Buyi Qiu, Kai Wang, and Jun Li*

Abstract: Network monitoring is receiving more attention than ever with the need for a self-driving network to tackle increasingly severe network management challenges. Advanced management applications rely on traffic data analyses, which require network monitoring to flexibly provide comprehensive traffic characteristics. Moreover, in virtualized environments, software network monitoring is constrained by available resources and requirements of cloud operators. In this paper, Trident, a policy-based network monitoring system at the host, is proposed. Trident is a novel monitoring approach, off-path configurable streaming, which offers remote analyzers a fine-grained holistic view of the network traffic. A novel fast path packet classification algorithm and a corresponding cached flow form are also proposed to improve monitoring efficiency. Evaluated in a practical deployment, Trident demonstrates negligible interference with forwarding and requires no additional software dependencies. Trident has been deployed in production networks of several Tier-IV datacenters.

Key words: cloud networking; software network monitoring; network programmability; network management

1 Introduction

Network management has been more challenging than ever as network complexity dramatically increases and failures become severe and knotty. Following the great success of software-defined networking, the vision of a self-driving network has been proposed to apply data-driven modeling and machine learning to traffic data analyses and closed-loop network automation[1, 2]. Figure 1 shows the framework of the self-driving network. Recent works[3–5] have started to build network data analytics platforms and provided logically centralized abstraction for learning applications. Therein, the first important step for traffic data analytics is network monitoring which collects the traffic information.

The focus of this study is on software network monitoring in the data plane. Software network monitoring is part of software network processing, which runs network software at end hosts. Software network processing, such as network virtualization and network function virtualization, is a pillar of multi-tenant cloud datacenters. It is flexible to develop new functionality and programmable model with software network processing. The combined constraints of cloud virtualization environment and traffic data analytics highlight three key requirements for software network monitoring.

(1) Noninterference. The intention of monitoring is to

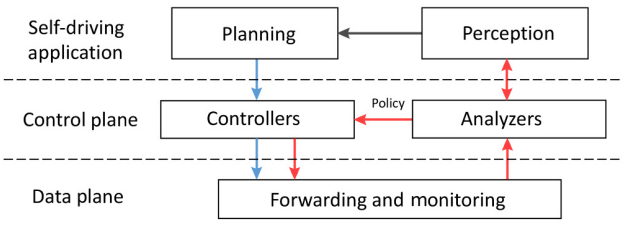


Fig. 1 Basic framework of a self-driving network.

• Xiaohe Hu and Yifan Li are with the Department of Automation, Tsinghua University, Beijing 100084, China. E-mail: hu-xh14@mails.tsinghua.edu.cn; liyifan18@mails.tsinghua.edu.cn.
• Yang Xiang, Buyi Qiu, and Kai Wang are with Yunshan Networks, Beijing 100084, China. E-mail: xiangyang@yunshan.net.cn; buyi@yunshan.net; wangkai@yunshan.net.
• Jun Li is with Research Institute of Information Technology, Tsinghua University, Beijing 100084, China. E-mail: junl@tsinghua.edu.cn.
* To whom correspondence should be addressed.
Manuscript received: 2020-01-17; revised: 2020-05-07; accepted: 2020-05-08

facilitate network management to not interfere with the original network delivery, i.e., forwarding, especially in end hosts that have shared and limited resources for network processing. Moreover, mixed monitoring and forwarding logic increase the complexity of operation and troubleshooting. (2) Comprehensiveness. A self-driving network relies on a comprehensive knowledge of traffic, and perception algorithms need to work at different packet granularities from a flow-level header to application-level payload. For example, flow scheduling[6, 7] based on flow statistics and deep inspection[8, 9] mines and matches payload signature patterns. (3) High efficiency. Software network processing is both time and resource consuming. Software network monitoring should be efficient to handle traffic with limited resources, quickly respond to the traffic status to support the fast control loop, and save network bandwidth due to the increasing traffic volume in cloud datacenters.

Previous works can be categorized into two directions: (1) direct streaming, which sends original traffic to remote analyzers from switches by mirroring[10] or configured forwarding rules[11, 12], and (2) local counting, which runs local algorithms to count traffic and sends statistics data to collectors, such as hash-based[13, 14] and sketch-based[15]. Although direct streaming can provide comprehensive packet information, it suffers from high resource consumption and interferes with forwarding, which degrades the original forwarding performance. On the other hand, local counting improves monitoring efficiency with careful data structure design, but it tailors the header structure and fails to support a full packet view. Hence, none of the various existing software monitoring solutions meet all the requirements.

In this paper, Trident, a novel software network monitoring approach at the host, is proposed to realize the noninterference and comprehensiveness requirements. Trident integrates the design of off-path monitoring and configurable streaming. It is decoupled from the forwarding path and trades the overhead of copying incoming packets for noninterference. When shortage of Central Processing Unit (CPU) resources happens due to competition from forwarding, Trident adaptively samples traffic to save resources for forwarding and guarantees its noninterference. Trident provides a full packet view by streaming the monitored packets and flexibly interacts with traffic analyzers with a policy-oriented programming model. Analyzers can define desired packets with match-action rules and acquire packets at different granularities from the header to payload.

The main technical challenge of the host monitoring design with the new approach of off-path configurable streaming is to realize high efficiency. Trident incorporates a wildcard-match fast path to improve the average classification performance. A novel hash-based packet classification algorithm, Unified Space Search (USS), and a corresponding cached flow form, uniflow, are proposed. USS maps original flow entries to unified non-overlapping flow entries, i.e., uniflows, which can be stored in one hash table. Fitting well with fast-path flow caching and classification, uniflows and USS achieve a high cache hit rate and near single-hash-lookup speed with limited memory usage. In addition, a lightweight compression algorithm is proposed for header delivery, saving bandwidth usage.

To mitigate the complexity of system deployment in cloud, Trident adopts a widely-used kernel module (the same interface as tcpdump) and requires no additional kernel dependencies. It has been deployed in production networks of several Tier-IV datacenters and provides a stronger capability of network traffic analyses. Currently, Trident implementation monitors 2×10^5 packets per second (abbreviated as pps below) traffic for header delivery consuming at most 0.3 core of Intel E5 CPU.

The rest of this paper is organized as follows. Section 2 describes the background of network monitoring, related work, and motivation of the Trident design. Section 3 introduces the Trident system architecture and describes the proposed algorithms and Trident modules. Section 4 presents the Trident implementation. Section 5 shows the evaluation. Section 6 discusses the current limitation of Trident. Section 7 concludes the paper by presenting the conclusion and future work.

2 Background

Network monitoring collects traffic data for better understanding and management of the running networks. Monitoring tools[16–18] have been embedded within network elements, such as servers, switches, and routers for several decades. Network monitoring schemes evolve with the development of programmable networking and datacenter networking. Recent works include designing expressive monitoring primitives and/or interfaces[4, 5, 14, 19, 20], optimizing algorithms and/or systems[14, 15, 21–24] to improve scalability with large traffic volume and limited resources, and exploring monitoring supported functionalities such as near-

optimal traffic engineering[25, 26] and network-wide troubleshooting[23, 24, 27].

Data plane network monitoring has hardware form (in commodity switches) and software form (at end hosts). Monitoring schemes adopted in hardware and software are similar. In hardware monitoring, OpenSample[28], Planck[26], and Everflow[24] directly stream data packets to remote analyzers by sampling or forwarding rules. FlowRadar[21] and OpenSketch series works[19, 20, 22] embed optimized hash and sketch logic into programmable silicon.

Compared with hardware solutions, software monitoring is more flexible to develop desired functionality and provide more detailed characteristics of virtual networks in cloud. Data plane software network monitoring is designed by two dimensions and falls into four main categories, as summarized in Table 1. The first design dimension is whether monitoring is processed on or off the forwarding path. The second design dimension is whether monitoring is counting statistics locally or streaming traffic to remote analyzers directly.

Specifically, FCAP and SMON are designed in both on-path and off-path forms[29], the evaluation of which shows that off-path approaches reduce the forwarding delay introduced by monitoring. UMON[13] monitors kernel space traffic with configured flow entries and collects statistics in user space tables. Trumpet[14] uses hash tables to monitor network events at Google end hosts. SketchVisor[15] realizes a robust software sketch solution with an augmented fast path and control plane recovery algorithm. OpenStack Tap-as-a-Service[10] uses the port mirroring method and sends traffic remotely. Software switches, such as Open vSwitches[11] and VFP[12], can be used to direct desired traffic with configured forwarding rules in the same way Everflow conducts with hardware switches.

Trident fills in the blank of the off-path streaming design, meeting the software network monitoring requirements of noninterference and comprehensiveness. Trident realizes off-path processing using the Linux shared memory interface, mmap, the same way as SketchVisor. Moreover, Trident provides a policy-based streaming model for remote analyzers. The policies are in a widely-used match-action form. To meet the high efficiency requirement and improve the processing speed of monitoring policies, Trident proposes a novel fast path packet classification algorithm, i.e., USS, with a cached flow form, i.e., uniflow.

Packet classification is an important research topic and used in various network functions, such as switch, firewall, and quality of service. Classical fast packet classification algorithms are decision-tree-based[30–32], which trades pre-processing time for compact tree structure and fast speed. Due to the dynamic rule update requirement in cloud[11], hash-based packet classification algorithms, such as Tuple Space Search (TSS)[33], are adopted as the fast update feature of hash schemes. An example of the TSS algorithm is shown in Fig. 2. TSS groups the rule set with tuples, i.e., header mask vectors. Rules with the same tuple can be classified with a hash table, the key of which is the rule prefix. To classify a packet, first, do AND operation on the packet header and the tuple, and then use the result as the key to look up the tuple hash table. Assume that a hash table lookup/update complexity is $O(1)$, and then TSS lookup/update complexity is $O(T)$ (to traverse the tuple list, T represents the tuple number).

As the rule number increases, the tuple number, i.e., lookup complexity, increases and TSS classification speed is slower than that of decision-tree algorithms. The fast path can be used to increase the system average classification speed. The fast-path design is a general cache approach used in networks. Packets arriving at

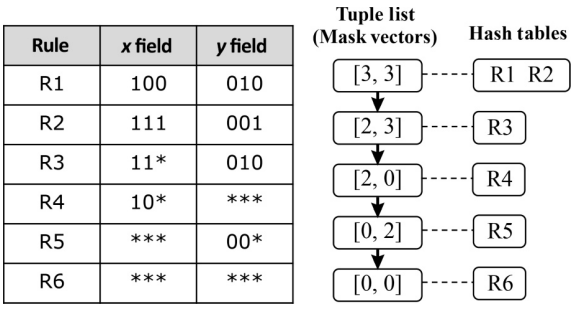


Fig. 2 TSS algorithm example with a two-field rule set.

Table 1 Summary of the software network monitoring work in the data plane.

Category	Local counting	Direct streaming
On-path	Hash-based: UMON[13], On-path-FCAP[29] Sketch-based: On-path-SMON[29]	Port mirroring: OpenStack Tap-as-a-Service[10] Configured forwarding rules: Open vSwitch[11], VFP[12]
Off-path	Hash-based: Trumpet[14], Off-path-FCAP[29] Sketch-based: SketchVisor[15], Off-path-SMON[29]	Configured monitoring policies: Trident

a network system will be first classified and executed in the fast path. If the packet is not matched in the fast path, then it will be classified in the slow path with the complete rule set and a computed sub-rule will be cached in the fast path for the classification of subsequent packets. The fast- and slow-path framework is shown in Fig. 3.

Open vSwitch adopts the fast-path design and uses TSS in the slow and fast paths. Given that the matched rule in the slow path cannot be directly cached into the fast path (if rules with higher priorities exist in the slow path and are not cached, then the fast path classification semantic becomes incorrect), Open vSwitch generates wildcard megaflows into the fast path to increase the cache hit rate (compared with the exact match). Megaflows are in the same prefix/mask form as rules in the TSS structure. When a packet is classified in the slow path, an all-zero mask vector $\langle m_1, m_2, \ldots, m_F \rangle$ (F represents the field number) is initiated and performed by the AND operation with each TSS tuple. Then, the cached megaflow is $\langle p_1/m_1, p_1/m_2, \ldots, p_1/m_F \rangle$ ($\langle p_1, p_2, \ldots, p_F \rangle$ represents the arriving packet header). However, due to the exhaustive AND operations in slow path, megaflow masks are close to the longest rule masks, resulting in a low wildcard space size, i.e., cache hit rate. Trident proposed uniflows cover a larger space than megaflows, improving the fast path cache hit rate.

3 Design

This section starts by providing an overview of the Trident architecture. Then, it describes the Trident monitoring approach to realize noninterference and comprehensiveness. Next, it elaborates how Trident classifies packets in accordance with monitoring policies. Finally, it describes how Trident exports desired traffic data.

3.1 Architecture

Trident is designed with the objective to be an independent, lightweight, and policy-based programmable monitoring system. Figure 4 shows the architecture of Trident.

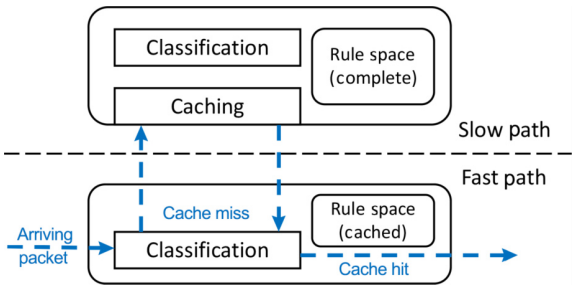


Fig. 3 Fast- and slow-path framework.

Native kernel space function. The kernel space functions are responsible for packet capturing. Same as tcpdump and wireshark, Trident uses the general and widely-used interfaces AF_PACKET/libpcap and Berkeley Packet Filter (BPF)[34, 35]. In this way, Trident provides a mechanism as acceptable as tcpdump, and therefore alleviates cloud providers' concern on inserting uncertain modules. It pulls packets from kernel space to user space with a zero-copy ring buffer through mmap. Moreover, to avoid getting unnecessary (e.g., control messages) and redundant (e.g., bridged and duplicated interfaces in OpenStack) packets, Trident performs a light pre-filter in accordance with the network interface index ifindex using BPF which is configured by the bytecode loaded from the user space.

Single user space process. Trident runs its main functions within a user space process. Regarding programmability, the process has an agent thread that receives monitoring policies from the remote controller. The agent is responsible for updating local policies and BPF bytecode. The Trident process involves classifiers and exporters, which stream desired traffic to remote analyzers for further processing, such as statistics and deep inspection. On receiving packets from the kernel space, Trident executes the corresponding classifier in accordance with the ifindex, and then exports policy-selected packets. The design of the classifier and exporter is elaborated in Sections 3.3 and 3.4, respectively.

3.2 Monitoring

Trident proposes a novel monitoring approach, i.e.,

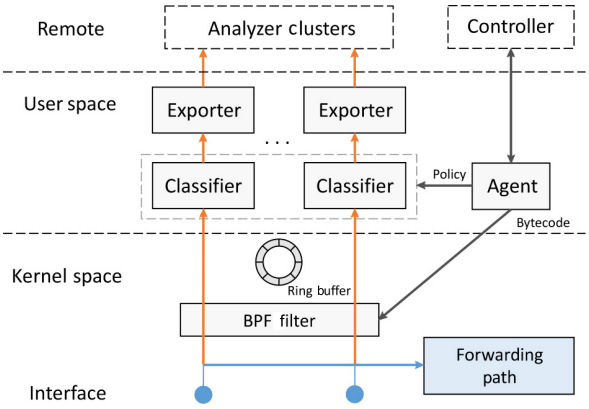


Fig. 4 Architecture of Trident. Trident performs off-path traffic monitoring within the host hypervisor and interacts with the remote controller and analyzers.

off-path configurable streaming, as categorized in Table 1. Compared with other monitoring approaches, Trident meets the noninterference and comprehensiveness requirements. This subsection shows Trident's monitoring approach in detail.

Noninterference. Trident is decoupled from the forwarding path of network processing, which is responsible for the application traffic delivery. In this way, packet copying is traded for an independent monitoring process. Therefore, Trident can be scheduled to either share the same core with forwarding or run on a separate core from forwarding, depending on the cloud provider's strategy.

Furthermore, given that current datacenters allocate limited resources for software network processing, Trident supports adaptive traffic sampling, making it run with any workload without interference with forwarding resources. Trident can be configured with two kinds of parameters: the maximum core (i.e., CPU usage) allocated for Trident and the allocated cores shared with other network processing. The Trident agent reads the configuration and periodically checks the corresponding CPU usage. By default, Trident monitors every incoming packet. Once the CPU usage exceeds the limit, the Trident agent will push a new BPF bytecode with an updated sample ratio to the kernel. The current Trident ratio control policy uses the Additive Increase Multiplicative Decrease (AIMD) strategy as the Transmission Control Protocol (TCP) rate control.

Trident supports controller-defined policies which specify the desired traffic with different header fields and corresponding actions. Hence, cloud providers can flexibly steer selected traffic data to destined analyzers and depict the traffic profile of individual cloud tenants. The monitoring policies of Trident employ the match-action model, which is pervasively leveraged by firewall Access Control Lists (ACLs), OpenFlow, and many other network functions. Current match supports 6 tuples, i.e., interface, src_ip, dst_ip, src_port, dst_port, and protocol, described in a prefix/mask format. Trident provides two actions: (1) header, to send the packet header and related states, such as timestamp, sample ratio (if enabled), etc. and (2) payload, to send the original packet. Trident acts as a white-list, thus any unmatched packet will be omitted by Trident.

3.3 Classifier

Separating monitoring policies from forwarding policies drives Trident to design a novel scheme for monitoring policy classification. Trident adopts the slow and fast path design to improve the average processing speed. Trident keeps TSS as the slow path classification algorithm, the same as Open vSwitch, given the fast-update and linear-memory properties of TSS. To tackle the limitations of TSS on the slow classification speed and fast-path low cache hit rate, Trident optimizes the fast path classifier design. A general hash-based packet classification algorithm USS is proposed, and then it is applied to fast path caching and classification.

To elaborate further details about USS, each rule in policies is referred to as a flow entry. In the view of computational geometry, an exact flow entry represents a point in multidimensional space, and a wildcard flow entry represents a hyperrectangle in the multidimensional space.

USS is hash-based and aims to decrease the time complexity from TSS $O(T)$ to the optimal one hash table lookup $O(1)$. USS exploits the latent capacity of non-overlapping flow entries and constructs a kind of non-overlapping uniflows, which can be classified with a single hash table. Therefore, the problem boils down to transform the original flow entries to a specific form of non-overlapping flow entries, i.e., uniflows, and then make hashing work on them.

USS construction and lookup. The basic idea of USS is to align the masks of flow entries to the longest mask in each field, i.e., to cut flow entries by the longest mask, and then use the prefixes of the transformed flow entries as keys of the hash table. Considering that real-life flow entries are unevenly distributed and long masks are collocated[36, 37], USS uses a hierarchical and grouped mask structure to mitigate the space explosion due to the mask alignment. The USS data structure construction and lookup are illustrated with an example shown in Fig. 5.

The construction process consists of three stages. (1) Mask mapping. First, cut each field into sub-spaces by the first X bits (X is a predefined hyperparameter). Refer to the first X bits as a sub-prefix for conciseness. Then, store the longest mask of each sub-space in mask arrays for the mask alignment of next stage. For example, the Internet Protocol (IP) space is cut by the first 16 bits, and the port space is cut by the first 8 bits. Figure 5b uses a 6.4×10^4 unit array for IP mask mapping and a 256-unit array for Port mask. In the sub-space of the IP sub-prefix 0.0, the longest mask is set to 22 instead of the global longest mask 24, thus saving the aligned flow entry number. (2) Grouped

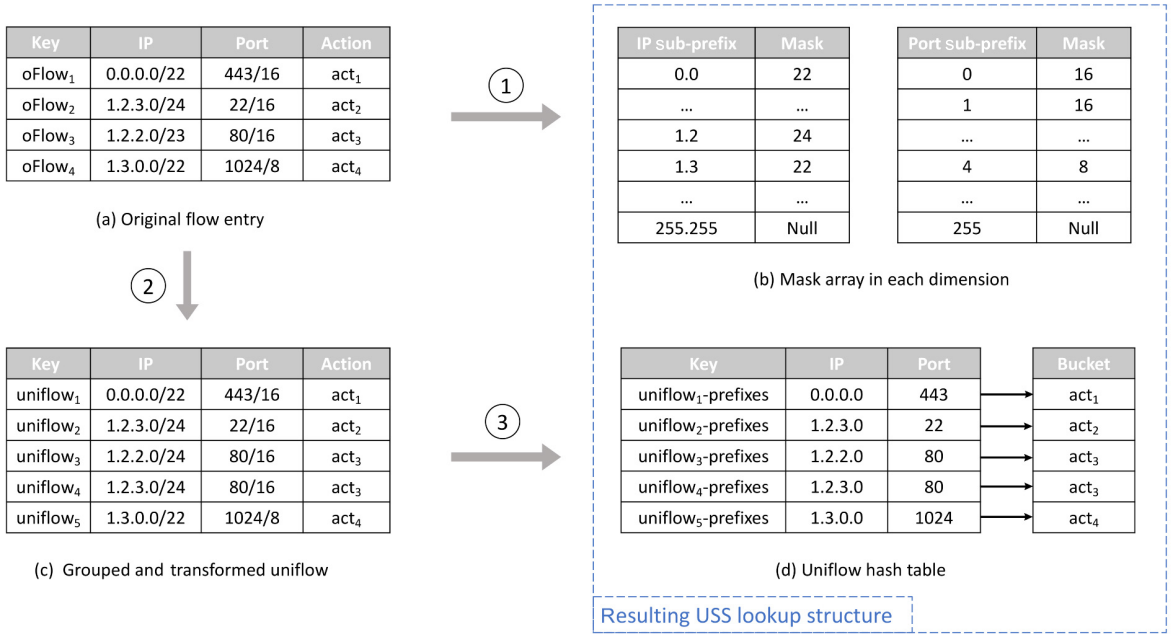


Fig. 5 Example of the USS construction process.

transformation. Align the original flow entry mask in each sub-space, i.e., to cut the flow entry by the longest mask, and then generate non-overlapping uniflows. For example, in the sub-space of the IP sub-prefix 1.2, the longest mask is set to 24, and the original flow entry oFlow_3: $\langle 1.2.2.0/23, 80/16\rangle$ is cut to two transformed uniflows uniflow_3: $\langle 1.2.2.0/24, 80/16\rangle$ and uniflow_4: $\langle 1.2.3.0/24, 80/16\rangle$. The grouped and aligned uniflows are shown in Fig. 5c. (3) Uniflow hashing. Due to the non-overlapping property, the transformed uniflows can be classified within a single hash table. The hash table keys are the prefixes of the uniflows, and the values are the uniflow actions. For example, uniflow_1: $\langle 0.0.0.0/22, 443/16\rangle$ in Fig. 5c corresponds to the key uniflow_1-prefixes: $\langle 0.0.0.0, 443\rangle$ in Fig. 5d. The resulting lookup data structure is composed of the mask arrays and the uniflow hash table. The USS construction algorithms are shown in Appendix (Agorithms 1–3).

During the lookup process, the mask arrays are first accessed to get the corresponding longest mask in each field in accordance with the sub-prefixes of the arriving packet header. Then, the lookup key is computed by an AND operation of the packet header and field masks. Finally, the key is used to perform a hash lookup in the uniflow hash table and derive the matched action. For example, given the structures in Figs. 5b and 5d, if a packet p that p.ip $=$ 1.2.2.3 and p.port $=$ 80 arrive, according to mask arrays, the IP mask of sub-prefix 1.2 is 24 and the port mask of sub-prefix 0 is 16. Then the computed key is $\langle 1.2.2.0, 80\rangle$. The hashed value is act_3 in the bucket of uniflow_3-prefixes, and finally p will be executed with the action act_3, which is the action of original oFlow_3. The USS lookup algorithm is shown in Appendix (Aglorithm 4).

Uniflow caching and fast-path USS classification. To use USS in the fast path, firstly, construct mask arrays (Fig. 5b) from the original flow entries and initial an empty uniflow hash table (Fig. 5d). Then, cache wildcard flow entries into the fast path. Instead of the AND operations performed on the megaflow mask and each TSS tuple to ensure semantic correctness, uniflows can be directly cached from the slow path to the fast path because of the non-overlapping property of uniflows. Therefore, the caching process is that when an arriving packet is classified in the slow path, (1) the longest mask of each field is identified through the sub-prefix of the packet header, (2) an AND operation of field masks and the packet header is performed, (3) the AND operation result is the uniflow-prefixes key and the slow-path classification result action is the bucket value, (4) the $\langle$key, value$\rangle$ pair is inserted into the fast path hash table. Finally, subsequent packets can be classified in the fast path with the normal USS lookup process. The uniflow caching algorithm is shown in Appendix (Algorithm 5).

In a general packet classification scenario, USS may still suffer from the space-consuming problem when the flow entry distribution is quite spread out. However, when USS is applied to fast path classification, the

problem is not significant at all due to the fast path cache property, i.e., dynamically maintaining part of the rule set instead of the complete rule set in the slow path. The uniflows are generated and dynamically cached packet by packet from the slow path, and when the cached uniflow number exceeds the given fast path maximum uniflow cache size, the cache replace approach is adopted. Therefore, by adopting USS classification and uniflow caching, Trident fast path gains the wildcard-match capability and preserves a near-session lookup speed, i.e., the number of mapping array accesses and one uniflow hash table lookup.

3.4 Exporter

Exporters are responsible of executing policy actions and delivering monitored traffic data. The exported traffic data are encapsulated by either User Datagram Protocol (UDP) or Virtual eXtensible Local Area Network (VXLAN) header, and its destination is set to a dispatcher switch. Using the UDP channel requires a Trident collector process to parse monitored data in the remote analyzer. By contrast, for private third-party analyzers, Trident supports original packet delivering through the VXLAN channel, which is parsed and decapsulated by dispatcher switches.

Note that modern cloud datacenters leverage tunnel policies in forwarding virtual switches to realize network virtualization. However, the tunnel policies are not visible to Trident, resulting in indistinguishable Virtual Machine (VM) traffic in analyzers when Trident monitors traffic from more than one VM with the same address but in different virtual networks. To monitor the original forwarding traffic, a key observation is that traffic from one single interface belongs to a virtual network, i.e., encapsulated by the same tunnel header. Therefore, monitored traffic can be unambiguously identified through the combination of a host identity and an interface identity. The combined identity is encoded in a self-defined field over the UDP channel or the Virtual Network Identifier (VNI) field in the VXLAN channel.

Another adopted approach in exporters is compression when executing the policy header action. As header statistics are enabled almost in every monitoring scenario, compressing the transmitted headers can save a considerable amount of bandwidth. Header compression is a common approach in low-speed-link and wireless communication fields[38–40]. Recent monitoring work NetSight[23] also uses a compression algorithm based on RFC1144[38]. However, NetSight leverages the similarity between successive packets in the same flow, thus having to maintain per-flow states. Instead, given that traffic from one interface still has a relatively high similarity, Trident leverages the similarity between packets from the same interface and maintains per-interface states, saving states and realizing a lightweight algorithm.

Trident uses the same difference-based compression algorithm as NetSight and RFC1144. In detail, the first packet is not compressed, and the sender/receiver initiates and maintains a packet state, i.e., values of each header field, of each interface by the first packet. The subsequent packet is compared with the maintained packet, and only the field value of the subsequent packet which is different from that of the maintained packet is sent. Then, the sent field values are updated to the maintained packet state. With Trident's per-interface compression, packets that are not successive but have the same field can be compressed. For example, for a sequence of packets ⟨udp, ethernet, icmp, tcp⟩, the IP field of udp, icmp, and tcp can be compressed, and the port field of udp and tcp can be compressed.

4 Implementation

Trident is implemented in Go. Benefitting from decoupling monitoring and forwarding and using general interfaces, such as AF_PACKET, libpcap, and Go packages, Trident can be deployed without additional dependencies. Trident can be used with hypervisors, such as Linux KVM and VMware ESXi, and virtual switches, such as Open vSwitch and VMware vSphere Distributed Switch. As a lightweight solution for deployment and management, Trident's bin file is less than 15 MB, and its bootstrap stage takes only 1 s. Cloud providers can start to use Trident on all servers with one single policy to acquire VM traffic header statistics. For a deeper analysis on specific applications, cloud providers can enforce fine-grained monitoring policies and stream the application traffic to specific Deep Packet Inspection (DPI) analyzers.

5 Evaluation

Trident implementation is evaluated and analyzed in a multi-tenant cloud environment. Firstly, Trident's fast-path algorithm is evaluated and compared with Open vSwitch fast path algorithm, then Trident's performance on resource usage at a given workload is illustrated, and the evaluation and analysis of the header compression

are shown. Finally, a demo of adaptive sampling for resource noninterference is shown.

Experiment setup. The experiments are run on a server with an 8-core Intel Xeon CPU E5-2650 2.00 GHz and 96 GB DRAM. The fast-path algorithm comparison uses the public datasets ClassBench[41] in packet classification. On system evaluation, the server is configured as an OpenStack compute node with Linux KVM and OVS, which is connected to a controller server through a 1 Gbps Network Interface Card (NIC) and to a Top Of Rank (TOR) switch through two 10 Gbps NICs. Trident is deployed on this server, monitoring the traffic of virtual interfaces and exporting desired states to a remote analyzer through the 10 Gbps NICs. Traffic is replayed by tcpreplay, including CAIDA Internet trace[42] and random traces generated by Linux pktgen. A CentOS VM is launched in a virtual network on this server as the sender, which sends traces to a gateway VM in another server through the 10 Gbps NICs.

Fast-path algorithm. Trident generates uniflows and classifies uniflows with USS in fast path, whereas Open vSwitch generates megaflows and classifies megaflows with TSS in the fast path. Tested slow-path rule sets include ClassBench synthesized Access Control List (ACL) rules from 100 to 10^4 size and Firewall rules from 100 to 10^4 size. Traces are randomly generated in accordance with the corresponding rule sets. The experiment fast-path flow entry size, i.e., cache size, is set to 1000 (corresponding to 1000%, 100%, and 10% of the slow-path rule sizes 100, 10^3, and 10^4) for both Trident and Open vSwitch.

The fast-path hash table lookup times with full cached flow size are evaluated and shown in Table 2. Although TSS needs to classify a group of hash tables, the experiments show that the hash table lookup times of TSS with megaflows are almost as linear as those of USS with uniflows. The reason is that to maintain the semantic consistency of cached flow entries in the fast path and original flow entries in the slow path, generated megaflows need to intersect slow-path tuples, resulting in near one minimal tuple. Therefore, as shown in Fig. 6, the average expressed flow size (cache hit rate) of a megaflow is serveral order of magnitude smaller than

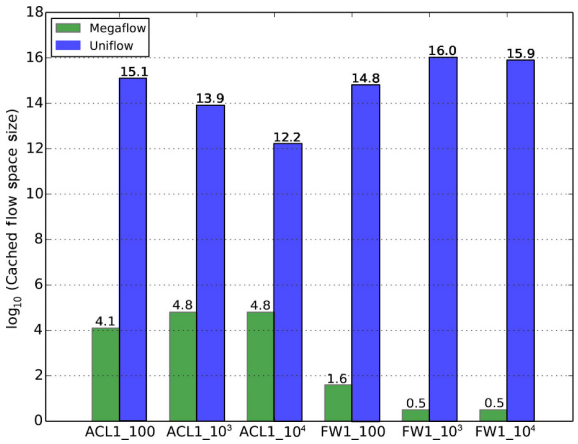


Fig. 6 Average cached flow size of the megaflow and uniflow, representing the fast path caching hit rate. *Y*-axis is shown in the log scale. On each flow entry, the space size is calculated by multiplying each field size.

that of a uniflow. The overhead of the linear hash lookup and high cache hit rate properties of USS with the uniflow is the bounded-memory mask arrays, which consume less than 132 KB (2^{16} sub-prefixes of src_ip and dst_ip, 2^8 sub-prefixes of src_port and dst_port, and 2^4 sub-prefixes of protocol) for any 5-tuple rule set.

Resource usage. Trident resource usage on the CPU and memory is evaluated. Trident is configured in two modes: header statistics with header-action policies and selective packet mirroring with payload-action policies. Five types of traces are used: four randomly generated traces with the packet length ranging from 64 to 1458 bytes and one payload-stripped CAIDA trace due to anonymity.

Table 3 shows the CPU usage results at 2×10^5 pps. Decoupling monitoring from forwarding adds one-copy overhead to the forwarding path. This overhead is shown in the middle column of each monitoring mode. The Sum column represents the actual CPU usage of Trident. Trident consumes more CPU resources when mirroring traffic, which is sourced from the packet Input/Output (I/O). For a large packet length, i.e., 1458 bytes, the exported packet will be encapsulated and fragmented, thus resulting in a higher CPU usage. On memory usage, Trident consumes around 329.6 MB memory in the no-load condition, which is mainly from mmap, packet-

Table 2 Fast-path hash table lookup times compassion on TSS with megaflow and USS with uniflow. *x* in rule set ACL1_*x* and FW1_*x* represents rule size in the set.

Rule set	ACL1_100	ACL1_10^3	ACL1_10^4	FW1_100	FW1_10^3	FW_10^4
TSS with megaflow	1	1	1	3	2	2
USS with uniflow	1	1	1	1	1	1

Table 3 Trident CPU usage at 2×10^5 pps. Random-*x* traffic pattern represents the packet source IP and randomly varying ports. The packet length is set to *x*. Mon represents the CPU usage of the Trident process. Copy represents the CPU usage of the copy overhead introduced to the forwarding path. Sum represents the sum of Mon and Copy.

(%)

Traffic pattern	Header statistics			Packet mirroring		
	Mon	Copy	Sum	Mon	Copy	Sum
Random-64	12.23	11.91	24.14	58.32	9.37	67.69
Random-512	10.43	8.72	19.15	59.18	9.83	69.01
Random-1024	11.38	14.15	25.53	62.69	13.28	75.97
Random-1458	12.15	16.21	28.36	100	17.85	117.85
CAIDA	14.68	3.64	18.32	55.97	3.58	59.55

sending buffers, and fast-path tables. Due to the garbage collection mechanism of Go, the accurate memory usage for random traces cannot be shown. For processing the CAIDA traces, Trident consumes 336.9 MB memory. Trident increases slight memory consumption while monitoring traffic due to the wildcard flow caching of the fast path.

Header compression. The header compression ratio is evaluated with CAIDA traces. Figure 7 shows the original packet rates of forwarding and after-compression packet rates of monitoring during 400 s. Trident achieves around 45:1 compression ratio on pps. In the quantitative analysis, the compression algorithm generates 22-49 bytes-long headers and assembles these headers in 1500 bytes-long packets that contain 48 bytes metadata. Therefore, the compression ratio is theoretically ranging from 29:1 to 66:1.

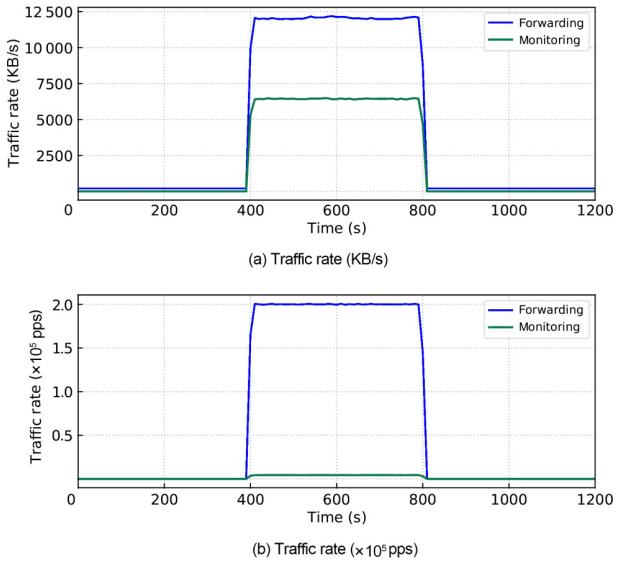


Fig. 7 Traffic rate example that Trident monitors and compresses the header statistics.

Moreover, according to the profiling result, the computation overhead of the header compression in the Trident process is 15.94% (8.52% comparing +7.42% appending), whereas 50.66% computation is from the mmap read operation.

Adaptive sampling. The adaptive sampling of the CPU usage of Trident is shown in Fig. 8. The desired CPU usage is set to 10%. The sent traffic rate varies between 150 and 250 kpps.

6 Discussion

Currently, Trident is limited in two aspects. (1) Overhead of the off-path monitoring. This aspect is due to the copy of incoming packets and introduces additional computation in the forwarding path and may also limit the monitoring throughput. The latest AF_PACKET v4[43], not yet merged to the kernel main branch, is a promising way to mitigate this overhead, which realizes zero-copy by mapping Direct Memory Access (DMA) packet buffers to the user space. (2) Lack of kernel bypassing support[44]. The current Trident design is based on the kernel I/O mode, which can not achieve 10 Gbps throughput for 64 bytes packets with one core. As Go provides a Data Plane Development Kit (DPDK) binding operation, the future version will integrate the kernel-bypassing I/O mode. Besides the limitations

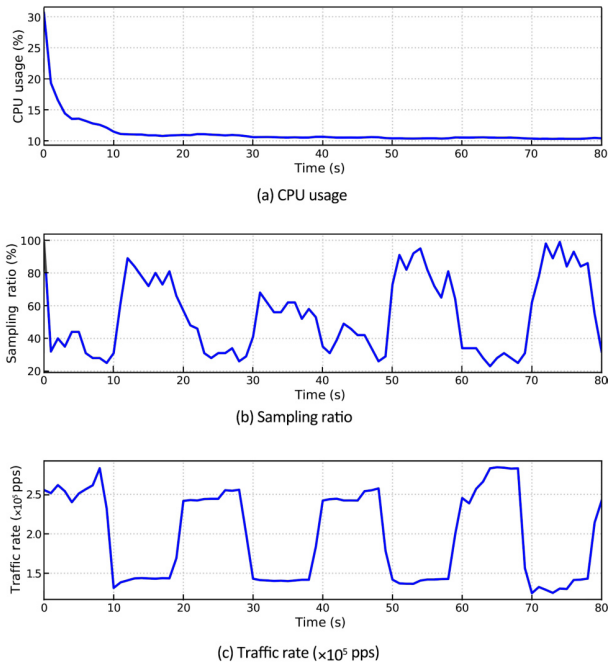


Fig. 8 Demo that Trident can dynamically vary the sampling ratio to keep the CPU usage at 10%. The sampling ratio here means dividing the number of monitored packets by the number of forwarded packets.

and corresponding optimization, Trident will adapt to standard control protocols, such as OpenFlow, and more traffic analyzers.

7 Conclusion

Self-driving networks aim to make network management simple and intelligent, which depends on data analytics and closed-loop control, and network monitoring is a fundamental pillar of the closed-loop network. In this paper, Trident, a forwarding-independent system for software network monitoring, designed with a new angle of off-path configurable streaming, is proposed. Trident is policy-oriented to support both header statistics and full packet delivery. Trident realizes negligible interference and high efficiency with a novel fast path design, compression algorithm, and adaptive sampling. The current Trident implementation processes header statistics of 2×10^5 pps traffic with at most 0.3 core of Intel E5 CPU.

Future work will incorporate zero-copy and kernel bypassing techniques into Trident and enhance the Trident programming model to adapt to standard control protocols, such as OpenFlow, and more traffic analyzers.

Appendix

USS construction algorithms contain rule mask alignment function, mask array construction function, and uniflow table construction function (Algorithms 1–3).

USS lookup algorithm uses the constructed mask arrays and uniflow table (Algorithm 4).

Uniflow caches are drived from arriving packets (Algorithm 5).

Algorithm 1 Rule mask alignment

```
Let DIM be the dimension bit number vector, and D be the
dimension number
Execute AlignRule(r, out_rs, tr, longest_masks, D − 1)
function (AlignRule r, out_rs, tr, longest_masks, d)
    if d = 0 then
        tr.action = r.action
        out_rs.add(deep_copy(tr))
        return
    end if
    for i = 1 → 2^(longest_masks[d]−r.mask[d]) do
        tr.prefix[d] = r.prefix[d]+((i − 1) ≪ (DIM[d]−
                       longest_masks[d]))
        tr.mask[d] = longest_masks[d]
        AlignRule(r, out_rs, tr, longest_masks, d − 1)
    end for
end function
```

Algorithm 2 Mask array construction

```
Let X be the first bit number vector for grouping
function (SetMaskArray (rs, mask_arrays))
    Initiate dimension mask array vector mask_arrays[D]
    for i = 0 → D − 1 do
        Allocate memory size 2^X[i] for mask_arrays[i]
    end for
    for i = 0 → rs.size − 1 do
        for j = 0 → D − 1 do
            sub_prefix = rs.rule[i].prefix[j]≫(DIM[j] − X[j])
            if rs.rule[i].mask[j]>mask_arrays[j][sub_prefix]
              then
                mask_arrays[j][sub_prefix]=rs.rule[i].mask[j]
            end if
        end for
    end for
end function
```

Algorithm 3 Uniflow table construction

```
function (SetUniflowTable rs, mask_arrays, table)
    Let trs be the transformed uniflows
    for i = 0 →rs.size −1 do
        for j = 0 → D − 1 do
            sub_prefix = rs.rule[i].prefix[j] ≫ (DIM[j]−X[j])
            longest_masks[j] = mask_arrays[j][sub_prefix]
        end for
        AlignRule(rs.rule[i], trs, tr, longest_masks, D − 1)
    end for
    for i = 0 →trs.size −1 do
        table.add(trs.rule[i].prefix, trs.rule[i].action)
    end for
end function
```

Algorithm 4 USS lookup

```
function (USSLookup mask_arrays, table, p)
    for j = 0 → D − 1 do
        sub_prefix = p.header[i] ≫ (DIM[i] − X[i])
        mask = mask_arrays[j][sub_prefix]
        key[i] = p.header[i] & ∼ (1 ≪ (DIM[i] − mask) −1)
    end for
    table.lookup(key, action)
    return action
end function
```

Algorithm 5 Uniflow caching

```
function (UniflowCaching fpath_mask_arrays, fpath_hash_
table, p, action)
    for j = 0 → D − 1 do
        sub_prefix = p.header[i]≫(DIM[i] −X[i])
        mask = fpath_mask_arrays[j][sub_prefix]
        key[i] = p.header[i] & ∼ (1 ≪ (DIM[i] − mask) − 1)
    end for
    fpath_hash_table.add(key, action)
end function
```

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 61872212).

References

[1] Juniper, Expel complexity with a self-driving network, https://www.juniper.net/us/en/products-services/whatis/selfdriving-network/, 2020.

[2] N. Feamster and J. Rexford, Why (and how) networks should run themselves, in *Proc. Applied Networking Research Workshop*, Montreal, Canada: ACM, 2018, p. 20.

[3] J. C. Jiang, V. Sekar, I. Stoica, and H. Zhang, Unleashing the potential of data-driven networking, in *Proc. 9th Int. Conf. on Communication Systems and Networks*, Bengaluru, India, 2017, pp. 110–126.

[4] Y. F. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo, Quantitative network monitoring with NetQRE, in *Proc. Conf. of the ACM Special Interest Group on Data Communication*, Los Angeles, CA, USA, 2017, pp. 99–112.

[5] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, Sonata: Query-driven streaming network telemetry, in *Proc. 2018 Conf. of the ACM Special Interest Group on Data Communication*, Budapest, Hungary, 2018, pp. 357–371.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, Hedera: Dynamic flow scheduling for data center networks, in *Proc. 7th USENIX Conf. on Networked Systems Design and Implementation*, San Jose, CA, USA, 2010, p. 19.

[7] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, DevoFlow: Scaling flow management for high-performance networks, in *Proc. ACM SIGCOMM 2011 Conf.*, Toronto, Canada, pp. 254–265, 2011.

[8] M. Roesch, Snort-lightweight intrusion detection for networks, in *Proc. 13th USENIX Conf. on System Administration*, Seattle, WA, USA, 1999, pp. 229–238.

[9] Z. L. Yuan, Y. B. Xue, and M. van der Schaar, BitMiner: Bits mining in internet traffic classification, in *Proc. 2015 ACM Conf. on Special Interest Group on Data Communication*, London, UK, 2015, pp. 93–94.

[10] OpenStack, Tap as a Service (TAPaaS), https://docs.openstack.org/dragonflow/latest/specs/tap_as_a_service.html, 2020.

[11] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al., The design and implementation of Open vSwitch, in *Proc. 12th USENIX Conf. on Networked Systems Design and Implementation*, Oakland, CA, USA, 2015, pp. 117–130.

[12] D. Firestone, VFP: A virtual switch platform for host SDN in the public cloud, in *Proc. 14th USENIX Conf. on Networked Systems Design and Implementation*, Boston, MA, USA, 2017, pp. 315–328.

[13] A. Wang, Y. Guo, F. Hao, T. V. Lakshman, and S. Q. Chen, UMON: Flexible and fine grained traffic monitoring in open vSwitch, in *Proc. 11th ACM Conf. on Emerging Networking Experiments and Technologies*, Heidelberg, Germany, 2015, p. 15.

[14] M. Moshref, M. L. Yu, R. Govindan, and A. Vahdat, Trumpet: Timely and precise triggers in data centers, in *Proc. 2016 ACM SIGCOMM Conf.*, Florianopolis, Brazil, 2016, pp. 129–143.

[15] Q. Huang, X. Jin, P. P. C. Lee, R. H. Li, L. Tang, Y. C. Chen, and G. Zhang, SketchVisor: Robust network measurement for software packet processing, in *Proc. Conf. of the ACM Special Interest Group on Data Communication*, Los Angeles, CA, USA, 2017, pp. 113–126.

[16] sFlow, https://sflow.org, 2020.

[17] NetFlow, https://www.ietf.org/rfc/rfc3954.txt, 2020.

[18] Tcpdump, https://www.tcpdump.org, 2020.

[19] M. L. Yu, L. Jose, and R. Miao, Software defined traffic measurement with OpenSketch, in *Proc. 10th USENIX Conf. on Networked Systems Design and Implementation*, Boston, MA, USA, 2013, pp. 29–42.

[20] Z. X. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, One sketch to rule them all: Rethinking network flow monitoring with UnivMon, in *Proc. 2016 ACM SIGCOMM Conf.*, Florianopolis, Brazil, 2016, pp. 101–114.

[21] Y. L. Li, R. Miao, C. Kim, and M. L. Yu, FlowRadar: A better NetFlow for data centers, in *Proc. 13th Usenix Conf. on Networked Systems Design and Implementation*, Santa Clara, CA, USA, 2016, pp. 311–324.

[22] M. Moshref, M. L. Yu, R. Govindan, and A. Vahdat, SCREAM: Sketch resource allocation for Software-defined measurement, in *Proc. 11th ACM Conf. on Emerging Networking Experiments and Technologies*, Heidelberg, Germany, 2015, p. 14.

[23] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, I know what your packet did last hop: Using packet histories to troubleshoot networks, in *Proc. 11th USENIX Conf. on Networked Systems Design and Implementation*, Seattle, WA, USA, 2014, pp. 71–85.

[24] Y. B. Zhu, N. X. Kang, J. X. Cao, A. Greenberg, G. H. Lu, R. Mahajan, D. Maltz, L. H. Yuan, M. Zhang, B. Y. Zhao, et al., Packet-level telemetry in large datacenter networks, in *Proc. 2015 ACM Conf. on Special Interest Group on Data Communication*, London, UK, 2015, pp. 479–491.

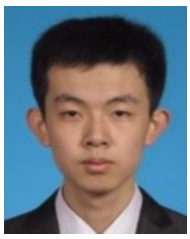
[25] T. Benson, A. Anand, A. Akella, and M. Zhang, MicroTE: Fine grained traffic engineering for data centers, in *Proc. Seventh Conf. on Emerging Networking Experiments and Technologies*, Tokyo, Japan, 2011, p. 8.

[26] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, Planck: Millisecond-scale monitoring and control for commodity networks, in *Proc. 2014 ACM Conf. on SIGCOMM*, Chicago, IL, USA, 2014, pp. 407–418.

[27] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, OFRewind: Enabling record and replay troubleshooting for networks, in *Proc. 2011 USENIX Conf. on USENIX Annu. Technical Conference*, Portland, OR, USA, 2011, p. 29.

[28] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, OpenSample: A low-latency, sampling-based measurement platform for commodity SDN, in *Proc. 2014 IEEE 34th Int. Conf. on Distributed Computing Systems*, Madrid, Spain, 2014, pp. 228–237.

[29] Z. L. Zha, A. Wang, Y. Guo, D. Montgomery, and S. Q. Chen, Instrumenting Open vSwitch with monitoring capabilities: Designs and challenges, in *Proc. Symp. on SDN Research*, Los Angeles, CA, USA, 2018, p. 16.
[30] P. Gupta and N. McKeown, Packet classification using hierarchical intelligent cuttings, in *Proc. Hot Interconnects*, Stanford, CA, USA, 1999.
[31] S. Singh, F. Baboescu, G. Varghese, and J. Wang, Packet classification using multidimensional cutting, in *Proc. 2003 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Karlsruhe, Germany, 2003, pp. 213–224.
[32] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, Packet classification algorithms: From theory to practice, in *Proc. IEEE INFOCOM 2009*, Rio de Janeiro, Brazil, 2009, pp. 648–656.
[33] V. Srinivasan, S. Suri, and G. Varghese, Packet classification using tuple space search, *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 135–146.
[34] S. McCanne and V. Jacobson, The BSD packet filter: A new architecture for user-level packet capture, in *Proc. USENIX Winter 1993 Conf. Proc. on USENIX Winter 1993 Conf. Proc.*, San Diego, CA, USA, 1993, p. 2.
[35] A. Begel, S. McCanne, and S. L. Graham, BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture, in *Proc. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Cambridge, MA, USA, 1999, pp. 123–134.
[36] M. L. Yu, J. Rexford, M. J. Freedman, and J. Wang, Scalable flow-based networking with DIFANE, in *Proc. ACM SIGCOMM 2010 Conf.*, New Delhi, India, 2010, pp. 351–362.
[37] Z. Liu, S. J. Sun, H. Zhu, J. Q. Gao, and J. Li, BitCuts: A fast packet classification algorithm using bit-level cutting, *Comput. Commun.*, 2017, vol. 109, pp. 38–52.
[38] V. Jacobson, Compressing TCP/IP headers for low-speed serial links, https://tools.ietf.org/html/rfc1144, 1990.
[39] M. Degermark, B. Nordgren, and S. Pink, IP header compression, https://tools.ietf.org/html/rfc2507, 1999.
[40] L. E. Jonsson, G. Pelletier, and K. Sandlund, The Robust Header Compression (ROHC) framework, https://tools.ietf.org/html/rfc5795, 2007.
[41] D. E. Taylor and J. S. Turner, ClassBench: A packet classification benchmark, *IEEE/ACM Trans. Netw.*, 2007, vol. 14, no. 3, pp. 499–511.
[42] CAIDA, The CAIDA anonymized Internet traces 2016 Dataset, https://www.caida.org/data/passive/passive_2016_dataset.xml, 2020.
[43] LWN, Introducing AF_PACKET V4 support, https://lwn.net/Articles/737947/, 2020.
[44] Data Plane Development Kit (DPDK), https://dpdk.org, 2020.



Xiaohe Hu received the BEng degree from Tsinghua University, China in 2014. He is now a PhD candidate at Department of Automation, Tsinghua University, China. His research interests include software-defined networking, cloud datacenter networks, and network monitoring and management.

Yang Xiang received the BS degree from Jilin University, China in 2008, and the PhD degree from Tsinghua University, China in 2013. He is currently a software engineer at Yunshan Networks. His research interests include software-defined networking, network architecture, and intrusion detection.

Jun Li received the BEng and MEng degrees from Tsinghua University, China in 1985 and 1988, respectively, and the PhD degree from New Jersey Institute of Technology, USA in 1997. Currently, he is a professor at Research Institute of Information Technology, Tsinghua University, China. His research interests include network security, pattern recognition, and image processing.

Buyi Qiu received the BEng degree from Northeastern University, China in 2012. He is now a software engineer at Yunshan Networks. His research interests include cloud datacenter networks, network monitoring, and network troubleshooting.

Kai Wang received the BS degree from Nanjing University, China in 2009 and the PhD degree from Tsinghua University, China in 2015. He is currently a software engineer at YunShan Networks. His research interests include network security and software-defined networking.

Yifan Li received the BEng degree from Tsinghua University, China in 2018. He is now a PhD candidate at the Department of Automation, Tsinghua University, China. His research interests include network verification, cloud datacenter networks, and network monitoring and management.