TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214
DOI: 10.26599/TST.2020.901
Volume xx, Number x, xxxxxxx 20xx

Trident: Efficient and Practical Software Network Monitoring

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

Abstract: Network monitoring is receiving more attention than ever with the need for Self-Driving Network to tackle increasingly severe network management challenges. Advanced management applications rely on traffic data analysis, which requires 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. This paper proposes Trident, a policy-based network monitoring system at the host. 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 proposed to improve monitoring efficiency. Evaluated in 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 Self-Driving Network has been proposed to apply data-driven modeling and machine learning to traffic data analysis and closed-loop network automation [1,2]. Figure 1 shows the framework of Self-Driving Network. Recent works [3–5] start to build network data analytics platforms and provide logically centralized abstraction for learning applications. Therein, the first important step for traffic data analytics is network monitoring which collects the traffic information.

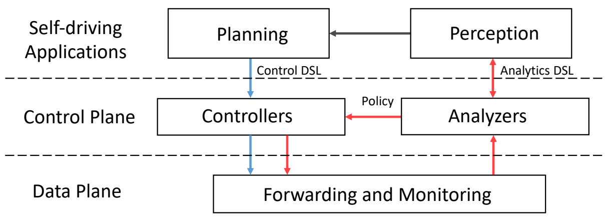


Fig. 1 A basic framework of Self-Driving Network

This paper focuses 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 facilitate network management not to interfere the original network delivery, *i.e.*, forwarding, especially

• 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.

in end hosts which have shared and limited resources for network processing. Moreover, mixed monitoring and forwarding logic increases the complexity of operation and troubleshooting. (2) *Comprehensiveness*. Self-Driving Network relies on a comprehensive knowledge of traffic and also perception algorithms need to work at different packet granularities from flow-level header to application-level payload. For example, flow scheduling [6, 7] bases 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, response traffic status quickly to support 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]. (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 original forwarding performance. On the other hand, local counting improves monitoring efficiency with careful data structure design, while it tailors the header structure and fails to support full packet view. None of the various existing software monitoring solutions meet all the requirements.

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

The main technical challenge of 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 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 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 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 analysis. Currently, Trident implementation monitors 200Kpps 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 the motivation of Trident design. Section 3 introduces Trident system architecture and describes the proposed algorithms and Trident modules. Section 4 presents Trident implementation and current limitation. Section 5 shows the evaluation. This paper is closed by conclusion and future work in Section 6.

2 Background

Network monitoring collects traffic data for better understanding and management of the running networks. Monitoring tools [17–19] 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, 20, 21], optimizing algorithm and/or system [14, 15, 22–25] to improve scalability with large traffic volume and limited resources, and exploring monitoring supported functionalities such as near-optimal traffic engineering [26, 27] and network-wide troubleshooting [24,25,28].

Data plane network monitoring has hardware form (in commodity switches) and software form (at end

	Local Counting	**Direct Streaming**
On-Path	Hash-based: UMON [13], On-path-FCAP [30] Sketch-based: On-path-SMON [30]	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 [30] Sketch-based: SketchVisor [15], Off-path-SMON [30]	Configured monitoring policies: Trident

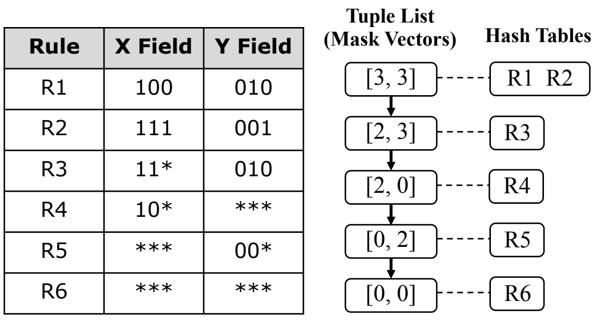
Table 1 Summary of software network monitoring work in the data plane

hosts). Monitoring schemes adopted in hardware and software are similar. On hardware monitoring, OpenSample [29], Planck [27] and Everflow [25] directly stream data packets to remote analyzers by sampling or forwarding rules. FlowRadar [22] and OpenSketch series works [20,21,23] embed optimized hash and sketch logic into programmable silicons.

Compared to 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 [30], evaluation of which shows off-path approaches reduce the forwarding delay introduced by monitoring. UMON [13] monitors kernel space traffc 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 robust software sketch solution with 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 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. Also, Trident provides policy-based streaming model for remote analyzers. The policies are in widely-used match-action form. To meet the *high efficiency* requirement and improve monitoring policies processing speed, Trident proposes a novel fast path packet classification algorithm USS with a cached flow form uniflow.



Rule	X Field	Y Field
R1	100	010
R2	111	001
R3	11*	010
R4	10*	***
R5	***	00*
R6	***	***

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

Packet classification is an important research topic and used in various network functions, such as switch, firewall, and QoS. Classical fast packet classification algorithms are decision-tree-based [31–33], 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) [34] , are adopted as the fast update feature of hash schemes. A example of TSS algorithm is shown in Figure 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, then use the result as the key to lookup the tuple hash table. Assume a hash table lookup/update complexity is $O(1)$, then TSS lookup/update complexity is $O(T)$ (to traverse the tuple list, T represents the tuple number).

As the rule number increases, tuple number, *i.e.*, lookup complexity, increases and TSS classification speed is slower than decision-tree algorithms. Fast path can be used to increase system average classification speed. Fast path design is a general cache approach used in networks. Packets arriving at a network system will be first classified and executed in fast path. If the packet is not matched in fast path, it will be classified in slow path with the complete rule set and a computed sub-rule will be cached in fast path for the classification

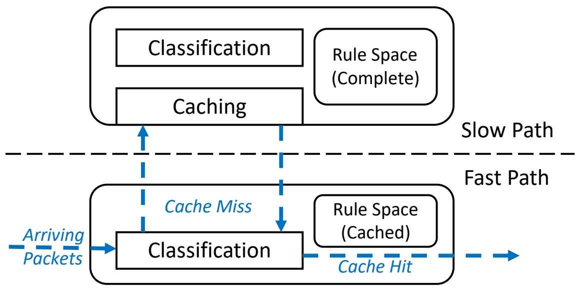


Fig. 3 The fast and slow path framework.

of subsequent packets. The fast and slow path framework is shown in Figure 3.

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

3 Design

This section starts by providing an overview of Trident architecture. Then this section describes the Trident monitoring approach to realize noninterference and comprehensiveness. Next, how Trident classifies packets according to monitoring policies is elaborated. Finally, this section 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:

Native kernel space function. The kernel space

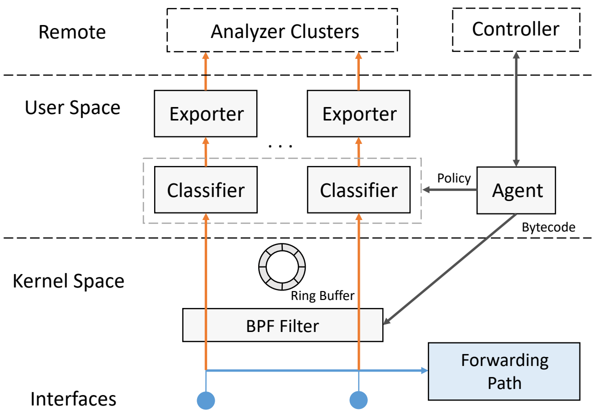


Fig. 4 The architecture of Trident. Trident does off-path traffic monitoring within the host hypervisor and interacts with the remote controller and analyzers.

functions is 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) [35, 36]. In this way, Trident provides a mechanism as acceptable as tcpdump, and therefore alleviates cloud provides' 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 (such as control messages) and redundant (such as bridged and duplicated interfaces in OpenStack) packets, Trident does a light pre-filter according to network interface index $ifindex$ using BPF which is configured by the bytecode loaded from user space.

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

3.2 Monitoring

Trident proposes a novel monitoring approach, *i.e.*, off-path configurable streaming, as categorized in Table 1. Compared to other monitoring approaches, Trident meets the noninterference and comprehensiveness requirements. This subsection shows Trident's monitoring approach in details.

Noninterference. Trident is decoupled from the forwarding path of network processing, which is responsible for 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 parameter: the maximum core (*i.e.*, CPU usage) allocated for Trident and the allocated cores shared with other network processing. Trident agent reads the configuration and checks the corresponding CPU usage periodically. By default, Trident monitors every incoming packet. Once CPU usage exceeds the limit, Trident agent will push a new BPF bytecode with updated sample ratio to kernel. Current Trident ratio control policy uses the additive increase multiplicative decrease (AIMD) strategy as 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 tenant. The monitoring policies of Trident employ the $match$-$action$ model, which is pervasively leveraged by firewall ACLs, OpenFlow, and many other network functions. Current $match$ supports 6 tuples, $interface$, src_ip, dst_ip, src_port, dst_port, $protocol$, described in prefix/mask format. Trident provides two $actions$: (1) $header$, to send 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

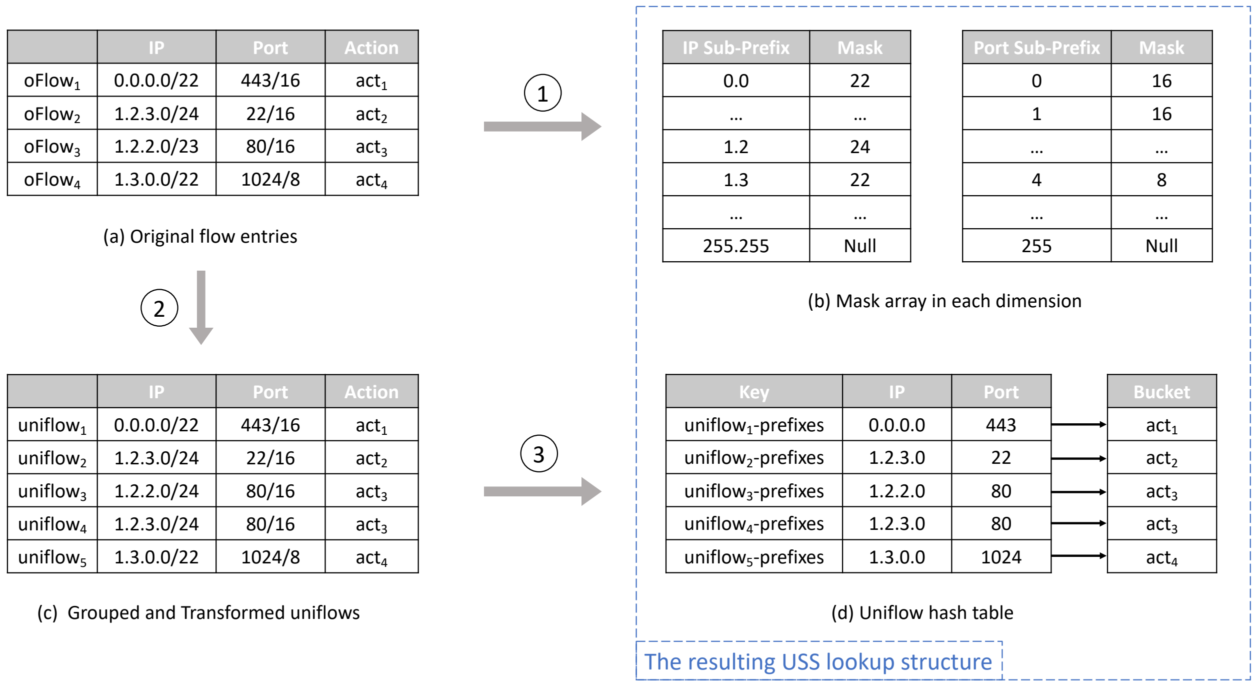
Separating monitoring policies from forwarding policies drives Trident to design a novel scheme for monitoring policy classification. Trident adopts the slow and fast path design to improve average processing speed. Trident keeps TSS as slow path classification algorithm, the same as Open vSwtich, given the fast-update and linear-memory properties of TSS. To tackle the limitations of TSS on 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 multi-dimension space, and a wildcard flow entry represents a hyper-rectangle in multi-dimension 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 signle hash table. Therefore, the problem boils down to transform 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 transformed flow entries as keys of the hash table. Considering that real-life flow entries are unevenly distributed and long masks are co-located [37, 38], USS uses hierarchical and grouped mask structure to mitigate the space explosion due to mask alignment. USS data structure construction and lookup is illustrated with an example shown in Figure 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 hyper-parameter). Refer to the first X bits as sub-prefix for conciseness. Then, store the longest mask of each sub-space in mask arrays for the mask alignment of next stage. In the example, IP space is cut by the first 16 bits and port space is cut by the first 8 bits. Figure 5 (b) uses a 64K-unit array for IP mask mapping and a 256-unit array for Port mask. In the sub-space of IP sub-prefix 0.0, the longest mask is set to 22 instead of global longest mask 24, thus saving the aligned flow entry number. (2) *Grouped 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 IP sub-prefix 1.2, the longest mask is set to 24, and original flow entry $oFlow_3 : \langle 1.2.2.0/23, 80/16 \rangle$ is cut to two trans-



	IP	Port	Action
oFlow$_1$	0.0.0.0/22	443/16	act$_1$
oFlow$_2$	1.2.3.0/24	22/16	act$_2$
oFlow$_3$	1.2.2.0/23	80/16	act$_3$
oFlow$_4$	1.3.0.0/22	1024/8	act$_4$

(a) Original flow entries

IP Sub-Prefix	Mask
0.0	22
…	…
1.2	24
1.3	22
…	…
255.255	Null

Port Sub-Prefix	Mask
0	16
1	16
…	…
4	8
…	…
255	Null

(b) Mask array in each dimension

	IP	Port	Action
uniflow$_1$	0.0.0.0/22	443/16	act$_1$
uniflow$_2$	1.2.3.0/24	22/16	act$_2$
uniflow$_3$	1.2.2.0/24	80/16	act$_3$
uniflow$_4$	1.2.3.0/24	80/16	act$_3$
uniflow$_5$	1.3.0.0/22	1024/8	act$_4$

(c) Grouped and Transformed uniflows

Key	IP	Port	Bucket
uniflow$_1$-prefixes	0.0.0.0	443	act$_1$
uniflow$_2$-prefixes	1.2.3.0	22	act$_2$
uniflow$_3$-prefixes	1.2.2.0	80	act$_3$
uniflow$_4$-prefixes	1.2.3.0	80	act$_3$
uniflow$_5$-prefixes	1.3.0.0	1024	act$_4$

(d) Uniflow hash table

Fig. 5 An example of USS construction process.

formed 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 Figure 5 (c). (3) *Uniflow hashing*, finally 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 uniflows, and the values are uniflow actions. For example, $uniflow_1 : \langle 0.0.0.0/22, 443/16\rangle$ in Figure 5 (c) corresponds to the key $uniflow_1\text{-}prefixes : \langle 0.0.0.0, 443\rangle$ in Figure 5 (d). The resulting lookup data structure composes of the mask arrays and the uniflow hash table. The USS construction algorithms are shown in Appendix A.

During the lookup process, the mask arrays are first accessed to get the corresponding longest mask in each field according to 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, use the key to do hash lookup in the uniflow hash table and get the matched action. For example, given the structures in Figure 5 (b) and (d), say, if a packet p that $p.ip = 1.2.2.3$ and $p.port = 80$ arrives, according to mask arrays, IP mask of sub-prefix 1.2 is 24 and 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 $uniflow_3\text{-}prefixes$, 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 A.

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

In 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 complete rule set in slow path. The uniflows are generated and cached packet by packet dynamically from the slow path, and when cached uniflow number exceeds given fast path maximum uniflow cache size, the cache replace approach is adopted. Therefore, through adopting USS classification and uniflow caching, Trident fast path gains the wildcard-match capability and preserves near-session lookup speed, *i.e.*, the number of mapping array accesses plus one uniflow hash table lookup.

3.4 Exporter

Exporters are responsible to execute policy actions and deliver monitored traffic data. The exported traffic data is encapsulated by either UDP or 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, while 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. This can result in indistinguishable 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 host identity and interface identity. The combined identity is encoded in a self-defined field over the UDP channel or the VNI field in the VXLAN channel.

Another adopted approach in exporters is compression when executing the policy $header$ action. As header statistics is 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 [39–41]. Recent monitoring work NetSight [24] also uses a compression algorithm based on RFC1144 [39]. 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 details, the first packet is not compressed, and sender/receiver initials 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 while have the same field can be compressed. For example, for a sequence of packets $\langle udp, ethernet, icmp, tcp\rangle$, the IP field of $udp, icmp, tcp$ can be compressed, and the port field of udp, tcp can be compressed.

4 Implementation

Trident is implemented in Go. Benefit from decoupling monitoring and forwarding as well as 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 15MB, and its bootstrap stage takes only 1 second. 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 DPI analyzers.

5 Evaluation

Trident implementation is evaluated and analyzed in a multi-tenant cloud environment. First, Trident fast path algorithm is evaluated and compared with Open vSwitch fast path algorithm, then Trident performance on resource usage at given workload is illustrated, and the evaluation and analysis of header compression is 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 2.00GHz and 96GB DRAM. Fast path algorithm comparison uses the public data sets Classbench [42] in packet classifica-

Rule Set	ACL1_100	ACL1_1K	ACL1_10K	FW1_100	FW1_1K	FW_10K
TSS with Megaflow	1	1	1	3	2	2
USS with Uniflow	1	1	1	1	1	1

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.

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%

Table 3 Trident CPU usage at 200 Kpps packet rate. $Random\text{-}x$ traffic pattern represents packet source IP and port are varied randomly and packet length is set to xB. Mon represents the CPU usage of Trident process. $Copy$ represents the CPU usage of copy overhead introduced to forwarding path. Sum represents the sum of Mon and $Copy$.

tion. On system evaluation, the server is configured as an OpenStack compute node with Linux KVM and OVS and connect it to a controller server through a 1Gbps NIC and to a TOR switch through two 10Gbps NICs. Trident is deployed on this server, monitoring the traffic of virtual interfaces and exporting desired states to a remote analyzer through the 10Gbps NICs. Traffic is replayed by tcpreplay, including CAIDA Internet trace [43] 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 10Gbps NICs.

Fast path algorithm. Trident generates uniflows and classifies uniflows with USS in fast path, while Open vSwitch generates megaflows and classifies megaflows with TSS in fast path. Tested slow path rule sets include Classbench synthesized ACL rules from 100 size to 10K size and Firewall rules from 100 size to 10K size. Traces are generated randomly according to the corresponding rule sets. The experiment fast path flow entry size, *i.e.* cache size, is set to 1000 (corresponding to 1000%, 100%, 10% of slow path rule size 100, 1000, 10K) for both Trident and Open vSwitch.

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, experiments show that hash table lookup times of TSS with Megaflow is almost as linear as USS with Uniflow. The reason is that to maintain the semantic consistency of cached flow entries in fast path and original flow entries in slow path, generated megaflows need to intersection slow path tuples, resulting in near one minimal tuple. Therefore, as shown in Figure 6, the average expressed

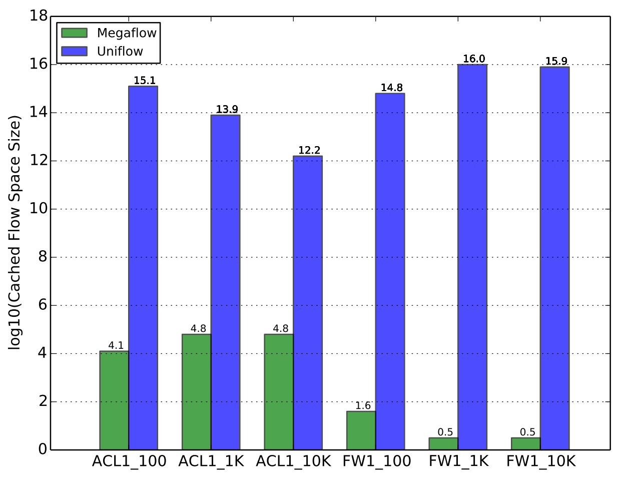


Fig. 6 The average cached flow size of megaflow and uniflow, representing the fast path caching hit rate. Y-axis is shown in log scale. On each flow entry, the space size is calculated by multiplying each field size.

flow size (cache hit rate) of megaflow is serveral order of magnitude smaller than uniflow. The overhead of linear hash lookup and high cache hit rate properties of USS with uniflow is the bounded-memory mask arrays, which consumes less than 132KB (2^{16} sub-prefixes of src_ip and dst_ip, 2^8 sub-prefixes of src_port and dst_port, 2^4 sub-prefixes of $protocol$) for any 5-tuple rule sets.

Resource usage. Trident resource usage on 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 packet length ranging from 64B to 1458B and one CAIDA trace stripped payload due to anonymity.

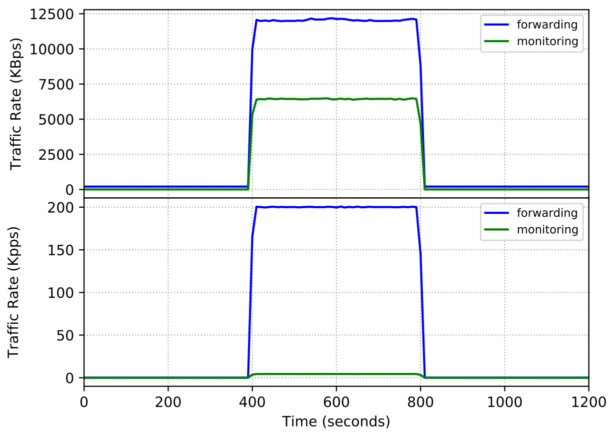


Fig. 7 A traffic rate example that Trident monitors and compresses header statistics.

Table 3 shows the CPU usage results at 200Kpps packet rate. Decoupling monitoring from forwarding adds one-copy overhead to 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 packet IO. For large packet length, 1458B, the exported packet will be encapsulated and fragmented, thus resulting in higher CPU usage. On memory usage, Trident consumes around 329.6MB memory in no-load condition, which is mainly from $mmap$, packet-sending buffers and fast-path tables. Due to the garbage collection mechanism of Go, accurate memory usage for random traces can not be shown. For processing the CAIDA traces, Trident consumes 336.9MB memory. Trident increases slight memory consumption while monitoring traffic due to the wildcard flow caching of fast path.

Header compression. 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 seconds. Trident achieves around 45-to-1 compression ratio on pps. In quantitative analysis, the compression algorithm generates 22B$\sim$49B-length headers and assembles these headers in 1500B-length packets which contains 48B meta-data. Therefore, the compression ratio is theoretically ranging from 29-to-1 to 66-to-1. Besides, according to the profiling result, the computation overhead of header compression in the Trident process is 15.94% (8.52% comparing + 7.42% appending), while 50.66% computation is from $mmap$ read operation.

Adaptive sampling. How adaptive sampling adjusts of CPU usage of Trident is shown. The desired CPU usage is set to 10%. The sent traffic rate is varied between 150Kpps and 250Kpps. Figure 8 shows the results.

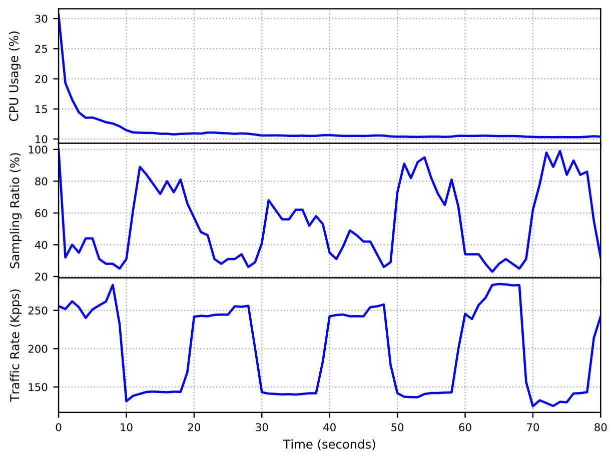


Fig. 8 A demo that Trident can dynamically vary sampling ratio to keep CPU usage at 10%. The sampling ratio here means dividing the number of monitored packets by the number of forwarded packets.

6 Discussion

Currently, Trident is limited in two aspects. (1) Overhead of off-path monitoring. This is due to one-copy on incoming packets. This introduces additional computation in forwarding path and may also limit monitoring throughput. The latest AF_PACKET v4 [44], not merged to kernel main branch yet, is a promising way to mitigate this overhead, which realizes zero-copy by mapping DMA packet buffers to user space. (2) Lack of kernel bypassing [45] support. Current Trident design is based on kernel IO mode, which can not achieve 10Gbps throughput for 64B packets with one core. As Go provides DPDK binding operation, future version is going to integrate the kernel-bypassing IO mode. Besides the limitations and corresponding optimization, Trident is also going to adapt to standard control protocol, 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. This paper proposes Trident, a forwarding-independent system for software network monitoring, designed with a new angle of off-path configurable streaming. Trident is policy-oriented to support both header statistics and full packet delivery. Trident realizes negligiable interference and high efficiency with novel fast path design, compression algorithm, and adaptive sampling. Current Trident implementation processes header statistics of 200Kpps 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 Trident programming model to adapt to standard control protocol, such as OpenFlow, and more traffic analyzers.

Acknowledgment

This work is supported by National Natural Science Foundation (No. 61872212).

References

[1] Juniper, What is Self-Driving Network? https://www.juniper.net/us/en/products-services/what-is/self-driving-network/. Accessed: 2020-01-13.

[2] N. Feamster and J. Rexford, Why (and how) networks should run themselves, in *Proceedings of the Applied Networking Research Workshop*, 2018.

[3] J. Jiang, V. Sekar I. Stoica, and H. Zhang, Unleashing the Potential of Data-Driven Networking, in *Proceedings of the 9th International Conference on COMmunication Systems NETworkS*, 2017.

[4] Y. Yuan, D. Lin, A. Mishra, S. Mishra, R. Alur, and B. T. Loo, Quantitative network monitoring with NetQRE, in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 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 *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, 2010, pp. 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, *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, 2011.

[8] M. Roesch, Snort: Lightweight intrusion detection for networks, *LISA*, 1999.

[9] Z. Yuan, Y. Xue, and M. v. d. Schaar, BitMiner: bits mining in internet traffic classification, in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 93-94.

[10] OpenStack Tap-as-a-Service, https://opendev.org/x/tap-as-a-service. Accessed: 2020-01-13.

[11] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, The design and implementation of Open vSwitch, in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015, pp. 117–130.

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

[13] A. Wang, Y. Guo, F. Hao, T. V. Lakshman, and S. Chen, UMON: Flexible and fine grained traffic monitoring in Open VSwitch, in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015, pp. 7.

[14] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, Trumpet: Timely and Precise Triggers in Data Centers, in *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, 2016, pp. 129–143.

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

[16] X. Wang, Z. Liu, Y. Qi, and J. Li, LiveCloud: A lucid orchestrator for cloud datacenters, in *Proceedings of 4th IEEE International Conference on Cloud Computing Technology and Science*, 2012, pp. 341-348.

[17] sFlow, https://sflow.org. Accessed: 2020-01-13.

[18] NetFlow, https://www.ietf.org/rfc/rfc3954.txt. Accessed: 2020-01-13.

[19] Tcpdump, https://www.tcpdump.org. Accessed: 2020-01-13.

[20] M. Yu, L. Jose, and R. Miao, Software defined traffic measurement with OpenSketch, in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013, pp. 29–42.

[21] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, One sketch to rule them all: Rethinking network flow monitoring with UnivMon, in *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, 2016, pp. 101–114.

[22] Y. Li, R. Miao, C. Kim, and M. Yu, FlowRadar: A better NetFlow for data centers, in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, 2016, pp. 311–324.

[23] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, SCREAM: Sketch Resource Allocation for Software-defined Measurement, in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015, pp. 13.

[24] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières and Nick McKeown, I know what your packet did last hop: Using packet histories to troubleshoot networks, in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014, pp. 71–85.

[25] Y. Zhu, N. Kang, J. Cao, A. G. Greenberg, G. Lu, R. Mahajan, D. A. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, Packet-level telemetry in large datacenter networks, in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 479–491.

[26] T. Benson, A. Anand, A. Akella, and M. Zhang, MicroTE: Fine grained traffic engineering for data centers, in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, 2011, pp. 12.

[27] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. B. Carter, and R. Fonseca, Planck: Millisecond-scale monitoring and control for commodity networks, in *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication*, 2014, pp. 407–418.

[28] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, OFRewind: Enabling record and replay troubleshooting for networks, in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, 2011, pp. 29.

[29] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. B. Carter, OpenSample: A low-latency, sampling-based measurement platform for commodity SDN, in *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, 2014, pp. 228–237.

[30] Z. Zha, A. Wang, Y. Guo, D. Montgomery, and S. Chen, Instrumenting Open vSwitch with monitoring capabilities: Designs and challenges, in *Proceedings of the Symposium on SDN Research*, 2018, pp. 7.

[31] P. Gupta and N. McKeown, Packet classification using hierarchical intelligent cuttings, in *Proceeding of Hot Interconnects*, 1999.

[32] S. Singh, F. Baboescu, G. Varghese, and J. Wang, Packet classification using multidimensional cutting, in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2003, pp. 213–224.

[33] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, Packet classification algorithms: From theory to practice, in *Proceedings of IEEE INFOCOM 2009*, 2009, pp. 648-656.

[34] V. Srinivasan, S. Suri, and G. Varghese, Packet classification using tuple space search, *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 135–146.

[35] S. McCanne, and V. Jacobson, The BSD packet filter: A new architecture for user-level packet capture, in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, 1993, pp. 2.

[36] A. Begel, S. McCanne, and S. L. Graham, BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture, in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1999, pp. 123–134.

[37] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, Scalable flow-based networking with DIFANE, in *Proceedings of the 2010 ACM Conference on Special Interest Group on Data Communication*, 2010, pp. 351–362.

[38] Z. Liu, S. Sun, H. Zhu, J. Gao, and J. Li, BitCuts: A fast packet classification algorithm using bit-level cutting, *Computer Communications*, 2017, vol. 109, pp. 38-52.

[39] V. Jacobson, Compressing TCP/IP headers for low-speed serial links, *RFC*, 1990.

[40] M. Degermark, B. Nordgren, and S. Pink, IP header compression, *RFC*, 1999.

[41] L-E. Jonsson, G. Pelletier, and K. Sandlund, The RObust Header Compression (ROHC) framework, *RFC*, 2007.

[42] D. E. Taylor, and J. S. Turner, ClassBench: a packet classification benchmark, *IEEE/ACM Trans. Netw.*, 2007, vol. 14, no. 3, pp. 499–511.

[43] The CAIDA anonymized Internet traces 2016 Dataset, http://www.caida.org/data/passive/passive_2016_dataset.xml. Accessed: 2020-01-13.

[44] Introducing AF_PACKET V4 support, https://lwn.net/Articles/737947/. Accessed: 2020-01-13.

[45] DATA PLANE DEVELOPMENT KIT, https://dpdk.org. Accessed: 2020-01-13.

A Appendix

USS construction algorithms contain rule mask alignment function, mask array construction function and uniflow table construction function.

Algorithm 1 Rule Mask Alignment

```
Let $DIM$ be the dimension bit number vector, $D$ 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 $i = 1 \to 2^{longest\_masks[d]-r.mask[d]}$ 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 $i = 0 \to D-1$ do
        Allocate memory size $2^{X[i]}$ for $mask\_arrays[i]$
    end for
    for $i = 0 \to rs.size-1$ do
        for $j = 0 \to D-1$ 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 $i = 0 \to rs.size - 1$ do
        for $j = 0 \to D - 1$ 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 $i = 0 \to trs.size - 1$ do
        $table.add(trs.rule[i].prefix, trs.rule[i].action)$
    end for
end function
```

USS lookup algorithm uses the constructed mask arrays and uniflow table.

Algorithm 4 USS Lookup

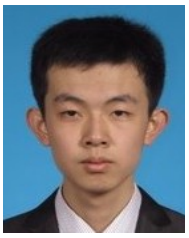
```
function USSLookup($mask\_arrays$, $table$, $p$)
    for $j = 0 \to D - 1$ do
        $sub\_prefix = p.header[i] >> (DIM[i] - X[i])$
        $mask = mask\_arrays[j][sub\_prefix]$
        $key[i] = p.header[i]\ \&\ \sim (1 << (DIM[i] - mask) - 1)$
    end for
    $table.lookup(key, action)$
    return $action$
end function
```

Uniflow caches are drived from arriving packets.

Algorithm 5 Uniflow Caching

```
function UniflowCaching($fpath\_mask\_arrays$, $fpath\_hash\_table$, $p$, $action$)
    for $j = 0 \to D - 1$ do
        $sub\_prefix = p.header[i] >> (DIM[i] - X[i])$
        $mask = fpath\_mask\_arrays[j][sub\_prefix]$
        $key[i] = p.header[i]\ \&\ \sim (1 << (DIM[i] - mask) - 1)$
    end for
    $fpath\_hash\_table.add(key, action)$
end function
```

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 in 1997. Currently, he is a professor at the Research Institute of Information Technology, Tsinghua University, China. His research interests include network security, pattern recognition, and image processing.



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

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

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

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