TOWARDS EFFICIENT AND SCALABLE SDN-BASED DATA CENTER MONITORING AND MANAGEMENT

by

Zili Zha

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

_____ Dr. Songqing Chen, Dissertation Director

_____ Dr. Fei Li, Committee Member

_____ Dr. Robert Simon, Committee Member

_____ Dr. Yang Guo, Committee Member

_____ Dr. David Rosenblum, Department Chair

_____ Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Summer Semester 2021
George Mason University
Fairfax, VA

Towards Efficient and Scalable SDN-based Data Center Monitoring and Management

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Zili Zha
Master of Applied Science
College of William and Mary, 2013
Bachelor of Computer Science
University of Science and Technology of China, 2009

Director: Dr. Songqing Chen, Professor
Department of Computer Science

Summer Semester 2021
George Mason University
Fairfax, VA

Dedication

I would like to dedicate this dissertation to my parents, Lihong Zha and Fangfang Zha, who provided endless support and love throughout my time at George Mason University.

Acknowledgments

I would like to extend my sincerest gratitude and appreciation to my advisor Professor Songqing Chen for his endless support during my Ph.D study and research at GMU, for his whole-hearted encouragement, careful instruction and immense knowledge. This dissertation would not have been possible without his continuous guidance and help. Besides my advisor, I would like to thank the rest of my dissertation committee: Professor Fei Li, Professor Robert Simon, Dr. Yang Guo, for their invaluable motivation and feedback that have shaped my final dissertation. My sincere thanks also go to my current and past research group members, Dr. An Wang, Dr. Mengbai Xiao and Dr. Li Liu, for their continuous support and stimulating discussions in the preparation of this dissertation.

Table of Contents

List of Tables

List of Figures

Abstract

TOWARDS EFFICIENT AND SCALABLE SDN-BASED DATA CENTER MONITORING AND MANAGEMENT

Zili Zha, PhD

George Mason University, 2021

Dissertation Director: Dr. Songqing Chen

Recent years have witnessed the great success of cloud computing that is powered by data center systems. As an essential building block for many of our cyber infrastructures, the monitoring, security, and management of data center traffic is of ultimate importance. However, the ever-increasing scale and speed of data center networks (DCNs) pose various challenges to data center traffic monitoring and management. The traditional monitoring approaches are neither flexible nor efficient to meet the security and performance needs of DCNs. With the flexibility and programmability of Software-Defined Networking (SDN) technology, this dissertation presents efficient solutions to address DCNs monitoring and management challenges.

First, we investigate the feasibility and efficiency of building software-defined monitoring (SDM) functionalities into network edges. For this purpose, We design and implement four measurement schemes in Open vSwitch (OVS), by either integrating forwarding and measurement functions into a pipeline, or decoupling them into parallel operations. We further include another eBPF-based approach in our comparative study. Then, we quantitatively show the various trade-offs among network performance and system overhead that different schemes strike to balance, and demonstrate the feasibility of instrumenting OVS

with monitoring capabilities.

Second, we leverage the flexibility and efficiency offered by SDM to enhance the security of DCNs. Accordingly, we design BotSifter, an SDN-based scalable, accurate and runtime bot detection framework for DCNs. To improve the detection scalability, BotSifter utilizes centralized learning with distributed detection by distributing detection tasks across the network edges. Furthermore, it employs multiple novel mechanisms for parallel detection of Command and Control (C&C) channels and botnet activities, which greatly enhances the detection robustness.

Third, the traffic surges in DCNs are common, due to the attacking (e.g., via botnets) traffic or the increase of legitimate traffic. They can easily degrade the performance of containerized clouds these days. To address this challenge, we further propose to adaptively offload the DCNs traffic and optimize the network virtualization performance in DCNs. For this purpose, we propose *EZPath*, a novel system that can seamlessly expedite container traffic by leveraging the programmable Top-of-Rack (ToR) switches in clouds. *EZPath* dynamically offloads selected (e.g., performance-critical) network flows to the in-network programmable hardware, which not only optimizes the network performance, but also effectively mitigates the negative impact of attack traffic.

Chapter 1: Introduction

With the great success of cloud computing, more and more traditional services have been migrated to or deployed on various cloud platforms. Naturally, cloud platforms become a critical cyber infrastructure these days. As an essential building block of a cloud computing platform, the data centers are of ultimate importance.

To respond to the ever-increasing demand, these days most data centers have adopted the modern networking technology, i.e., Software Defined Networking (SDN), in the data center networks (DCNs). Traffic monitoring has been playing a crucial role in a wide range of DCNs management tasks, such as traffic engineering, network diagnosis and troubleshooting, accounting, and anomaly detection. An ideal monitoring scheme would collect fine-grained per-flow statistics (i.e., packet/byte count for each measurement epoch). However, with the ever increasing network scales and network speeds of contemporary DCNs, the total number of traffic flows (5-tuples) could easily surpass the memory and processing capacity of traditional network devices. With these inherent hardware resource constraints, traditional commercial monitoring solutions (e.g., NetFlow, sFlow) only provide highly aggregated or sampled statistics that are insufficient to meet the monitoring needs of various management applications. Worse yet, low-rate flows that are missed by sampling based monitoring poses great threats to DCNs security.

With the increasing adoption of SDNs, nowadays software switches are gaining widespread deployment in the data centers and clouds, which opens up a new avenue for the development of network measurement frameworks. In SDN, the control plane and the data plane are decoupled to enable flexible and programmable network management. Typically the control plane is a logically centralized entity that maintains high level policies and communicates with the data plane using standard protocols such as OpenFlow [1]. The data plane consists of either software based or hardware based switches and routes traffic according

to the forwarding decisions configured by the control plane. Since software switches normally reside within the servers with abundant processing power and memory resources, it creates an enormous opportunity for developing flexible, efficient and scalable measurement frameworks.

1.1 Challenges in Data Center Networks

In this dissertation, we aim to address these challenges centering around the monitoring and management in DCNs, with the aid of programmable and flexible control enabled by SDNs.

Incorporating traffic monitoring capability into OVSes offers the opportunity to share the key functionalities required by monitoring that have been implemented in the switches. However, the design of such an integration faces a multitude of challenges in order to achieve minimal forwarding-monitoring function interference, optimal code sharing, and efficient CPU/memory resource usage. First, to guarantee easy deployability and minimal disruption to the normal service, the performance overhead, CPU utilization and memory usage introduced by the integrated monitoring functionalities should be acceptable. Second, the measurement results should be sufficiently accurate so as to allow accurate detection of a variety of network anomalies, such as heavy hitters, change detection, superspreader detection, port scans and other applications including flow distribution and entropy estimation. Lastly, how to enable efficient packet processing within the tight time budget in order to keep up with the ever increasing line speed remains to be an unsolved challenge. An in-depth investigation into various design options could provide valuable insights into building highly efficient and scalable DCNs management frameworks in general.

The success of cloud computing attracted not only various applications/services being deployed, but also attacks, such as those via botnets. Nonetheless, existing botnet detection schemes often fail to meet the requirements of accurate and expeditious detection in data centers due to their high resource demands. They often deploy a centralized network monitoring facility at the network gateway or firewall to capture incoming and outgoing

traffic. These solutions are hardly scalable considering the enormous traffic volumes in contempory DCNs. Moreover, earlier works are more focused on detection of malicious flows rather than identification of the bots, which is much more imperative for attack mitigation purposes. Only after the bots are identified, compromised nodes could be shut down or blocked to prevent future attacks. However, building a scalable and runtime bot detection and identification system is non-trivial and poses several challenges we need to address. First, the placement of the traffic capturing facilities is a key factor in building a scalable system given the network scales and traffic volumes. Second, bot identification in their earlier stages before any attacks are initiated requires a deep understanding of different phases in botnet lifecycles and dedicated designs of the runtime detection system. Guided by the monitoring options we explored, we aim to build a runtime, scalable and accurate bot detection framework for DCNs, by incorporating state-of-the-art deep learning techniques into our system design.

In addition to the adoption of SDN, the virtualization technology in cloud platforms also constantly evolves. Allured by its flexibility and low management overhead, many cloud platforms also start to deploy containers, in addition to the traditional virtual machines (VMs). In containerized clouds, applications are often deployed in VMs or containers as microservices, where software switches such as OVSes are employed for network virtualization. When they are under heavy load, caused by either attacking or legitimate traffic surges, the application performance would suffer from dramatic degradation. To deal with this challenge, we aim to optimize DCNs performance upon high traffic load. More specifically, we target a multi-tenant containerized environment, where OVS is leveraged to enable network virtualization and tenant isolation. We thoroughly investigate the network performance bottlenecks in container overlay networks and propose a novel design, *EZPath*, which aims to offload performance-critical network flows to the in-network programmable switching hardware to optimize the overall network performance. Nonetheless, migrating network functions and traffic to hardware raises several new challenges in a containerized environment. First, simply offloading all container traffic is clearly impractical considering the

sheer scale and density of container deployment. Hardware resources are highly constrained and could not accommodate all the metadata required by performing offloading network functions. Second, containers are much more short-lived than VMs. Constant updates of offloading selections could undesirably degrade the performance. Therefore, designing an adaptive offloading scheme in order to strike a balanced trade-off between resource utilization and network performance necessitates careful considerations.

In addition, our offloading design could also benefit network security scenarios, by offloading high rate application traffic to the fast path and minimizing the negative impacts that are introduced by the attacking traffic. In order to evade detection, nowadays network attacks have become much more sophisticated and per-flow attack traffic features a much lower rate, which is hard to contain by using conventional rate limiting. With hardware offloading, since high rate application traffic follows the fast path, the adverse effects could be effectively minimized.

1.2 Dissertation Contributions

This dissertation consists of the following contributions. First, we present a systematic study of software defined measurement with OVSes and provide various solutions for different scenarios. Network measurement is critical for various network operations and management but has been often constrained by the available resources in the traditional network devices. Recent advances in SDN have enabled flexible and programmable network measurement, which is referred to as SDM. A promising trend for SDM is to conduct network traffic measurement on widely deployed OVS in data centers. However, little attention has been paid to the design options for conducting traffic measurement on the OVS. In this study, we set to explore different design choices and investigate the corresponding trade-offs among resource consumption, measurement accuracy, implementation complexity, and impact on switching speed. For this purpose, we explore the design space and empirically design and implement four different measurement schemes in OVS, by either closely integrating forwarding and

measurement functions into a pipeline, or decoupling them into parallel operations. We further implement emerging eBPF-based monitoring approach that is independent from OVS and include in our comparative study. Through extensive experiments and comparisons, we quantitatively show the various trade-offs (e.g., among the metrics of throughput, latency, CPU overhead, memory overhead) that the different schemes strike to balance, and demonstrate the feasibility of instrumenting OVS with monitoring capabilities. These results provide valuable insights into which design will best serve different measurement and monitoring needs.

Second, to respond to attacks from botnets, we design and implement BotSifer, an SDN-based online bot detection framework. Botnets continue to be one of the most severe security threats plaguing the Internet. Recent years have witnessed the emergence of cloud-hosted botnets along with the increasing popularity of cloud platforms, which attracted not only various applications/services, but also botnets. However, even the latest botnet detection mechanisms (e.g., machine learning based) fail to meet the requirement of accurate and expeditious detection in data centers, because they often demand intensive resources to support traffic monitoring and collection, which is hardly practical considering the traffic volume in data centers. Furthermore, they provide little understanding on different phases of the bot activities, which is essential for identifying the malicious intent of bots in their early stages. In this work, we propose BotSifter, an SDN based scalable, accurate and runtime bot detection framework for data centers. To achieve detection scalability, BotSifter utilizes centralized learning with distributed detection by distributing detection tasks across the network edges in SDN. Furthermore, it employs a variety of novel mechanisms for parallel detection of C&C channels and botnet activities, which greatly enhance the detection robustness. Evaluations demonstrate that BotSifter can achieve highly accurate detection for a large variety of botnet variants with diverse C&C protocols.

Third, we develop EZPath to expedite container network traffic via programmable switches. Containerization, while getting popular in data centers, faces practical challenges due to the sharing nature of cloud networks among tens of thousands containers

simultaneously and dynamically. While the typical overlay approach enables network virtualization to facilitate multi-tenant isolation and container portability, this approach often suffers from degraded performance. Other proposed schemes addressing this performance bottleneck require either specialized hardware support, or customized software and extra maintenance. In this work, we propose *EZPath*, a novel approach that can seamlessly expedite the container traffic by leveraging the programmable Top-of-Rack (ToR) switches in clouds. By utilizing the underlying programmable switch's data plane capabilities, *EZPath* can adaptively offload heavy traffic directly from the container to the ToR switches, thus creating a fast and easy path to mitigate the network bottleneck. Such a ToR switches based solution is transparent to user applications, and does not require the change of OS kernel or the support of additional hardware. Using typical container workloads, we evaluate the performance of *EZPath*, and the results show that *EZPath* can significantly improve the application performance over the default overlay networking, e.g., a 35% throughput increase and a 42% tail latency reduction for Memcached.

1.3 Dissertation Organization

The remainder of the dissertation is organized as follows. Chapter 2 performs an in-depth investigation on the challenges of developing various software defined measurement frameworks in data center networks. Chapter 3 focuses on the design and implementation of accurate, scalable and runtime detection of cloud based botnet. Chapter 4 presents *EZ-Path*, a novel system that seamlessly expedite container overlay traffic by leveraging ToR programmable hardware switches in the clouds. Chapter 5 concludes this dissertation and proposes future works.

Chapter 2: Towards Software Defined Measurement with Open vSwitches: Designs, Implementation, and Evaluation

2.1 Introduction

Network measurement has been playing an essential role in a wide variety of network management tasks, ranging from traffic engineering, anomaly detection, to QoS provisioning, etc. Traditional monitoring tools, e.g., Netflow, sFlow, IPFIX, are usually deployed across in-network hardware devices to collect real-time traffic statistics. Nonetheless, due to the underlying hardware resource constraints, they only provide coarse-grained statistics that could not meet the monitoring demands of the diversified network applications. With the advancement of SDN and network function virtualization (NFV) [2] techniques, a series of research [3–6] have been proposed to enhance the existing measurement schemes. However, they are either not generic by requiring to implement multiple sketches for each measurement task [3–5], or too expensive to deploy in hardware devices [6]. In recent years, the emerging programmable dataplanes have spawned great opportunities for innovation in integrating monitoring solutions into the switching hardware. This trend enables SDM where users can flexibly program the monitoring rules. However, due to the hardware constraints of the switching ASICs, such as fixed hardware stages, limited per-stage actions and restricted stateful memory (e.g., Registers, Counters), collecting per-flow traffic statistics is not a trivial task. Instead of per-flow measurement, some prior work [7–9] focused on monitoring only heavy hitter flows.

The hardware-based monitoring frameworks all utilize sketch streaming algorithms aiming to minimize the memory consumption, since memory is the primary concern in hardware devices. In recent years, there is an increasing trend to build monitoring functionalities into end hosts as inspired by the following observations [10–12]. First, commodity servers are

in possession of plentiful CPU and memory resources. Compared to the hardware routers that often have limited computing and memory resources, data centers and clouds often have redundant resources in terms of computing power and memory capacity that are not fully utilized or idle.

Second, monitoring on the end hosts, i.e., the edge-based monitoring design, is much more scalable since each end host only needs to process much smaller amount of traffic as opposed to that of in-network hardware devices. This sheds light on SDM by using OVS. As an example, UMON [10] proposed a user-defined programmable traffic monitoring interface on OVS. Despite of the abundant hardware resources in commodity servers, building efficient monitoring frameworks into data center end hosts remains unexplored.

Incorporating traffic monitoring capability into a software switch offers the opportunity to share the key functionalities required by monitoring that have been implemented in a software switch. However, the design of such an integration is challenging in order to achieve minimal forwarding-monitoring function interference, optimal code sharing, and efficient CPU/memory resource usage. In this study, we set to empirically investigate the different design trade-offs using OVS [13] as a representative software switch. OVS has now become widely adopted for use as host-machine edge-routers in data centers. We start with an intuitive design, called FCAP (*Flow CAPture scheme*), where the forwarding and monitoring forms a pipeline in the OVS kernel. In FCAP, a packet traverses through the forwarding module before going through the monitoring module. The flow stats of interested traffic flows are first collected in the OVS kernel and then transferred to the user space for further processing. To reduce the memory consumption, we further design SMON, a *Sketch [14] based MONitoring* scheme that compresses the flow stats using sketches. Sketches are probabilistic data structures that trade off query accuracy for space efficiency and widely employed in a multitude of various applications. Since flow identifiers and per-flow traffic stats both need to be collected, we use an advanced sketch design, namely, invertible bloom lookup tables [14], in SMON, which allows us to easily recover the complete flow details in upper layer applications.

Since both FCAP and SMON work in a pipeline of forwarding and monitoring, monitoring could significantly impact the switch's forwarding throughput. To decouple the monitoring from the forwarding so as to minimize such impact, we propose to design off-path counterparts of FCAP and SMON by introducing a ring buffer in the kernel. The ring buffer temporally caches the packet headers of the interested traffic flows, which can then be processed independently by the monitoring module. In this way, the ring buffer effectively decouples the monitoring from the packet forwarding at the kernel data path.

FCAP/SMON designs all require extensive instrumentation into the OVS code base, which is not backwards-compatible. For practical deployment, we either need extra patches or re-install a modified version of OVS with the customized monitoring functionalities. Moreover, the SMON/FCAP monitoring modules are implemented within the kernel data path in order to achieve full visibility and minimize the impact on the forwarding performance. However, a single software flaw could crash the entire system. From this observation, we further propose an eBPF (extended Berkeley Packet Filter) enhanced [15] monitoring design. While its ancestor BPF is mostly used for in-kernel packet filtering, eBPF extends its architecture by integrating more features to support more types of events and actions other than filtering. eBPF offers the possibility to dynamically generate, load and execute code into the kernel using the `bpf()` system call, thus obviating the need to install customized kernel modules. Many eBPF-based tools are developed for performance debugging and troubleshooting, e.g., tracing the TCP sessions lifespan and the block device I/O latency, etc. Furthermore, BPF maps provide an asynchronous communication channel for sharing data between the userspace/kernel and across multiple runs of the kernel program. In our work, to gain full visibility of both inbound/outbound traffic, our monitoring programs are attached onto the Linux Traffic Control layer, while the monitoring filter and flow stats table are both implemented using eBPF maps.

In addition, we revisit UMON, a monitoring approach proposed in [10] that is largely implemented in the user space of OVS. We conduct extensive experiments to explore the

various trade-offs under the metrics of throughput, latency, CPU overhead, memory overhead etc. The results show that (1) compared to the on-path counterparts and eBPF, off-path approaches can significantly reduce the measurement delay introduced to the forwarding path, due to the decoupling enabled via the ring buffer; the off-path designs can also achieve the same measurement accuracy at the cost of higher memory consumption; (2) FCAP and eBPF can both achieve full measurement accuracy, by leveraging linked lists to resolve collisions in the underlying hash table implementations; while sketch based SMON can also achieve over 99% accuracy with less amount of memory consumption, it consumes more CPU cycles for sketch decoding compared to FCAP and eBPF; (3) on-path FCAP and eBPF are both built upon hash tables, with slight deviation in their performance due to the difference in underlying implementation; (4) UMON consumes a similar amount of memory as those of off-path schemes at the moderate packet rate, but consumes much more memory at the high packet rate; (5) compared to OVS kernel based schemes, UMON is most flexible and requires the least implementation efforts with no modifications to the Open vSwitch kernel code base. However, it derives fine-grained forwarding rules by combining the forwarding and monitoring functionality, which leads to the heaviest CPU load in the user space; (6) instrumenting OVS with monitoring capability is feasible without affecting the overall switching speed significantly; and (7) compared to the OVS-embedded designs, eBPF requires minimal maintenance efforts, since the monitoring module operates in parallel with and independently of OVS.

The reminder of the chapter is organized as follows. Section 2.2 describes some related work. We present our new designs and implementations in Section 2.3. We evaluate the proposed designs in Section 2.4 with more discussions in Section 2.5. Finally, we make concluding remarks in Section 2.6.

2.2 Related Work

Traditional Monitoring. Different network measurement frameworks have been investigated both in software and hardware switches. Traditional hardware-based solutions utilize tools such as Netflow [16], sFlow [17] and IPFIX [18], to collect IP Nework traffic. Other similar solutions include Jflow [19], Cflowd [20] and NetStream [21] etc. The drawbacks of these solutions are twofold: they are more expensive to deploy and they do not provide enough programmability for network management tasks.

SDN-enabled Monitoring. One of the earliest efforts is proposed by Yu et al. [3] called OpenSketch. In OpenSketch, different types of sketches are utilized to achieve different measurement goals. Furthermore, the controller optimizes the sketch allocation to balance the accuracy and the memory consumption. A followup prototype called DREAM [4] is proposed to dynamically assign TCAM counters to different measurement tasks across multiple hardware switches in the network. But the users could not customize measurement tasks other than the counter-based ones. In these earlier works, sketches are designed and implemented for specific monitoring tasks, which means that the monitoring devices must instantiate multiple sketches in order to support a variety of concurrent monitoring tasks. This places enormous burden on resource-constrained hardware devices and drastically degrades the network performance. To address this limitation, UnivMon [22] proposes a single universal sketch to support multiple measurement tasks simultaneously. Nonetheless, it requires to update multiple components for each packet, which also introduces noticeable overhead. Yu et al. also proposed FlowRadar [6] to improve the NetFlow based network measurement by encoding and decoding counters with the invertible bloom filter lookup table (IBLT). In this way, the communication overhead could be reduced. However, extra components are necessary to implement on hardware devices. Also, the decoding may introduce redundant overhead to the controller.

Monitoring within Programmable Dataplanes. *Space Saving* [23] is a widely known top-k algorithm to identify the first top-k frequent items in data streams. Compared to other counter-based streaming algorithms, *Space Saving* is much more resource efficient since it

only needs to maintain *O(k)* counters. Despite of its memory efficiency, *Space Saving* is not readily applicable for heavy flow detection within the emerging programmable hardware due to the underlying complexities in its data structure and algorithm design. Upon each new flow, the algorithm requires to find and replace the hash table entry with the minimum packet count, which cannot be easily implemented considering the hardware constraints of the hardware programming model. To adapt the classical algorithm into a hardware-friendly design, HashParallel and HashPipe [7] refactor the algorithm into a pipeline of hash tables that can fit in the programmable switches. This pipelined design helps to ensure that each stage only incurs a limited amount of processing in order to keep up with the line-rate switching throughput. Nonetheless, Precision [8] re-examines the problem and concludes that HashPipe is challenging to realize in the Reconfigurable Match Tables (RMT) [24] switch programming model since it does not satisfy the limited branching rule and single stage memory access rule imposed by the RMT model. To overcome the hardware limitations, Precision further improves the design by recirculating a small fraction of the packets at the cost of packet forwarding performance. Orthogonal to this direction, Memento [9] examines the problem from a different perspective by proposing a sliding window based heavy hitter detection model. It argues that sliding window models are more accurate and more efficient in terms of detection delay compared to the traditional interval based detection solutions. Further, it extends the algorithm to detect hierarchical heavy hitter (HHH) and network-wide scenarios. Sliding window based models have also been extensively studied in many earlier works [25, 26]. Marple [27] and Sonata [28] tackle the problem from a different perspective. Instead of designing new sketches to minimize the memory consumption in the hardware devices, Marple proposes a performance query language and designs new switch hardware primitives to support the language, which allows network operators to program their performance queries that collects customized fine-grained traffic statistics at a low processing overhead. Different from Sonata, it performs aggregations directly in the switch hardware, further reducing the data volume streamed to collection servers.

Edge-based Software Monitoring. Over the past few years, with the data center networks evolving to larger scales and the ever increasing line speeds, the resource constraints of hardware devices have become considerably more stringent. Comparatively, the edge servers are typically equipped with much more powerful hardware resources, e.g., CPU and memory. Motivated by this, there has been continuous efforts aiming to migrate monitoring functionalities from hardware devices to edge servers. Generally, existing software-based solutions can be broadly classified into two categories: passive monitoring system [12, 29, 30] and active monitoring system [10, 11]. The former category strives to collect traffic stats for all flows with minimal memory consumption and provable accuracy guarantees, by designing sophisticated sketches and algorithms. However, in order to keep up with high line-rates, they focus more on the accuracy of heavy flows while sacrificing that of the small flows, considering that heavy flows are usually more important than small flows in typical monitoring tasks. In contrast, active monitoring systems provide programmability that allows users to define their own monitoring tasks and only monitoring the traffic the network operators are interested in. This efficiently lessens the monitoring workload, further minimizing resource usage and impacts on the forwarding performance.

SketchVisor [12] focuses on accurate and timely network measurement under high traffic load. It proposes to combine a sketch based normal path and a top-k based fast path to achieve both high throughput and high accuracy. Under high traffic load, the fast path is activated to absorb the excessive traffic overflowed from the fast path with slight accuracy degradation. Further, it employs compressive sensing [31] to recover the flow stats information that serves as input for higher level monitoring applications. Following this work, Elastic Sketch [29] enhances SketchVisor by designing an elastic sketch with two components, a heavy part and a light part where the former maintains elephant flows and the latter records the mouse flows. Under heavy traffic load, only the heavy part is updated and the mouse flow information is lost. Compared to SketchVisor, Elastic Sketch achieves much higher performance since only one memory access is needed at high packet rate. In NitroSketch [30], it is pointed out that Elastic Sketch falls short in performance and

accuracy when the number of flows increases to a certain point. In comparison, NitroSketch proposes a generic sketching framework that addresses the bottlenecks of existing sketched designs and minimizes per-packet CPU and memory overhead. HeavyKeeper [32] further improves heavy hitter detection accuracy of Elastic Sketch via a new strategy, *count-with-exponential-decay*, to actively evict small flows through decaying. It reduces the error by 3 orders of magnitude compared to the state-of-the-art detection schemes. However, in certain applications, such as anomaly detection, small flows play an equally important role as heavy flows but cannot be captured by existing passive monitoring systems that focus on heavy flows.

Following this trend, Trumpet [11] is proposed to collect data from end-host machines to detect network-wide events. Though Trumpet is optimized to run on hardware network devices, it is independent of the existing network management framework.

2.3 Design and Implementation

Although existing software based measurement designs all function well under particular circumstances, an in-depth investigation about the resource-accuracy trade-offs is still lacking in the literature. We aim to fill this void by looking into the various software-based monitoring designs from a systematic view.

To empirically explore the various design trade-offs among multiple factors, including server resource consumption, monitoring overhead, and implementation complexities, in this section, we propose five novel monitoring designs, namely, on/off-path FCAP/SMON [33] and eBPF. Among them, four are incorporated into OVS, a widely deployed software switch in data centers, and an eBPF-based monitoring framework in parallel with OVS. Following a brief discussion about the design challenges arising from building monitoring logic into OVS, we walk through the design and implementation details of each monitoring framework.

2.3.1 OVS and Design Challenges

OVS consists of two major components: a userspace daemon (ovs-vswitchd), and a kernel datapath [34]. They work together to forward packets, with the userspace daemon as a full but slow path while the kernel datapath serving as a forwarding cache. Such a design aims to optimize the forwarding performance of the switch. More specifically, incoming packets are firstly matched against the flow table in the kernel datapath. The first packet of each new flow will encounter a flow miss in the kernel and is forwarded to the userspace by injecting a *upcall*. In the userspace, *upcall*s are handled by the *handler* threads, and a flow rule is generated and installed into the kernel flow cache. As a result, subsequent packets belonging to the same flow do not need to make detours through the userspace. Finer-grained kernel flow rules undoubtedly result in a larger number of flow misses and *upcall*s. This not only undermines the switch performance, but also introduces heavier workloads, thus higher CPU overhead for *handler*s. Fortunately, due to the locality of the network traffic, most packets are processed in the fast path.

The entries in the kernel flow cache and userspace flow tables contain fields such as packet and byte counts that provide built-in monitoring capabilities. These counters record the total packets processed by the corresponding flow entry. As aforementioned, the userspace does not have full visibility into all packets, since most packets are directly forwarded following the kernel cache. Thereby, the packet/byte counts in the userspace table entries need to be updated by polling the kernel cache entries. These are managed by the *revalidator* threads, which periodically poll the kernel cache for each flow's packet and byte counts and aggregates them into the userspace flow table. In addition, *revalidator*s are also responsible for maintaining the kernel cache entries. Similarly as *handler*s, a larger kernel cache introduces heavier workloads for *revalidator*s.

Nevertheless, this built-in feature in OVS flow tables is neither flexible nor sufficient for the dynamic monitoring needs, since flows that are relevant for monitoring and forwarding might not be overlapped. For example, the flow rule might specify the forwarding actions for packets with a specific destination IP address, while the monitoring applications need

fine grained flow statistics for each 5-tuple subflow. Relying only on packet and byte counts of the flow rule could not achieve the desired monitoring granularity. To cope with this limitation, some works propose to dynamically install flow forwarding rules for each subflow into OVS. As a consequence, the first packet of each subflow has to be sent to the centralized controller for further handling. This drastically degrades the forwarding performance of the dataplane and a more efficient programmable monitoring solution is needed. UMON is one of the earliest efforts in this direction.

The fundamental idea of UMON is to decouple monitoring from forwarding logic in the userspace, while the kernel datapath remains intact. To achieve this, the cached entries in the kernel need to be much more fine-grained than the native OVS, which is elaborated in Section 2.3.2. As explained earlier, this inevitably decreases the switching performance and incurs heavier consumption of system resources. Inspired by this, our work investigates the monitoring design from a variety of aspects: monitoring accuracy, resource consumption, switching performance, and portability.

Overall, to build monitoring capabilities into OVS, there are a number of challenges we need to address: (1) The added monitoring logic should introduce minimal interference to the forwarding path in OVS to guarantee the forwarding and monitoring efficiency; (2) Due to the resource constraints, it is necessary to strike a balance among efficiency, resource consumption, and monitoring accuracy; (3) To maximize feasibility and compatibility, the monitoring function should be as portable as possible so that minimal effort would be required to accommodate the monitoring function.

Taking these into consideration, the monitoring function is decoupled from the forwarding function by maintaining an additional monitoring table, as illustrated in Figure 2.1. The default forwarding process is performed by OVS, while the additional monitoring table supports the added monitoring functionalities. Contrarily, in the kernel datapath, four monitoring designs are proposed, which differ in multiple aspects, including interaction between the monitoring and forwarding functions, the placement of the monitoring module (Challenge 1), the stats collection data structures and algorithms (Challenge 2). Beyond

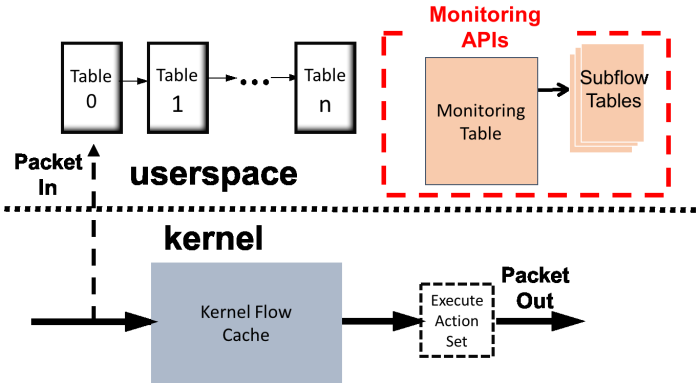

Figure 2.1: The OVS architecture

these, we also develop an eBPF-based monitoring framework that runs in parallel and independently with OVS (Challenge 3). In the following sections, we delve into the details of each specific design following a brief overview of UMON.

2.3.2 Recap of UMON

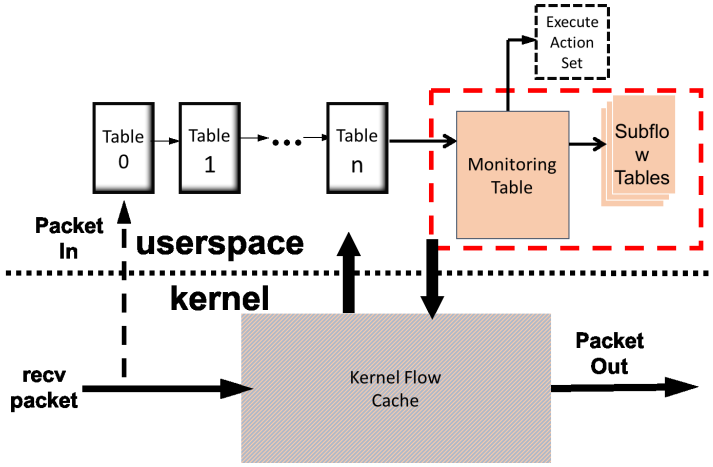

Figure 2.2: UMON architecture

As introduced in Section 2.3.1, the monitoring programmability of UMON is facilitated

through the introduction of an independent monitoring table in the userspace daemon. The overall architecture of UMON is depicted in Figure 2.2. Comparing this figure with Fig. 2.1, we can observe that UMON preserves the original architecture of OVS. It simplifies the monitoring design by only instrumenting the OVS userspace module, leaving the kernel datapath untouched. Specifically, the monitoring table maintains rules that monitor specific TCP traffic, such as TCP SYN packets, or/and collects subflows in the subflow tables. Besides, the monitoring table provides APIs for the controller to install and update monitoring rules via an extended OpenFlow protocol. UMON performs monitoring functions by chaining the monitoring table with the forwarding pipeline in the userspace. To support the user-defined monitoring granularities, the forwarding and monitoring flow rules are compiled together to generate cache entries in the kernel. For example, a flow rule forwards packets destined to host B to port 1 while the monitoring rule needs to collect the packet/byte counts originated from host A. UMON combines the two rules to generate a more fine-grained rule that forwards packets with source IP of A and destination IP of B to port 1 instead. Following this design, an incoming packet with *(srcIP=C, dstIP=B)* cannot find a match in the kernel cache and will raise a flow miss that needs to be sent to the userspace for further processing. As aforementioned, due to the lack of visibility in the userspace, flow statistics need to be properly populated from the kernel space to the userspace flow table. In the meanwhile, the flow stats in the monitoring table should also be updated. In UMON, this credit logic is piggybacked in the *revalidator* thread of OVS because it maintains the flow statistics periodically. As in the above example, the *revalidator* threads polls the kernel space for packet/byte counts in the cached flow entries and aggregates the micro-flows by different fields, e.g., dstIP for the flow table and srcIP for the monitoring table.

The downside of UMON is that the kernel flow table might get inflated with a vast amount of fine-grained flows during peak traffic. Furthermore, a significantly larger amount of flow misses are generated, thereby, both *handler* and *revalidator* threads will potentially experience much heavier workloads. On the other hand, in UMON, the monitoring packets

are reactively collected instead of proactively investigated in the kernel space. Therefore, the latency caused by monitoring interruptions will be reduced. Moreover, UMON does not require any modifications in the kernel, thus it could be easily ported to other edge devices and platforms, such as DPDK and NetFPGA. More details are discussed in [10].

In this study, we only use UMON as a comparison against the other designs. These designs, including UMON, embody different trade-offs between resource (e.g., CPU, memory) consumption, monitoring efficiency, forwarding efficiency (throughput and latency). One has to strike a balance among these considerations. In the following, we discuss the specific design considerations in greater detail.

2.3.3 Design of Flow Capture (FCAP)

The added monitoring capability inevitably introduces some workload to the OVS and can affect the forwarding performance. Such effect depends primarily on where the monitoring functions are placed. Intuitively, a separate monitoring function in the userspace provides better isolation. However, (1) the userspace agent does not have enough visibility of the flows in the fast path since only certain packets will traverse the userspace pipeline; (2) the collection of monitoring stats should be prompt to preserve accuracy. Based on these considerations, we propose to build a separate monitoring phase in the kernel datapath in OVS.

To guarantee the accuracy of monitoring tasks, we need to maintain statistics of all the related packets efficiently. The micro-flow information is more preferable than the mega-flow information because monitoring tasks often have dynamic granularity requirements and micro-flows simplify the aggregation operations. Thus, we first design two different schemes, FCAP (`Flow CAPture`) and SMON (`Sketch based MONitoring`), to collect micro-flows. In our current designs, we use 6-tuples (source/destination IP addresses, source/destination ports, protocol and TCP flags) to represent each micro-flow.

Fig. 2.3 shows the architecture of FCAP and SMON. In this figure, since userspace pipeline is similar as UMON, the forwarding pipeline in the userspace is omitted here for

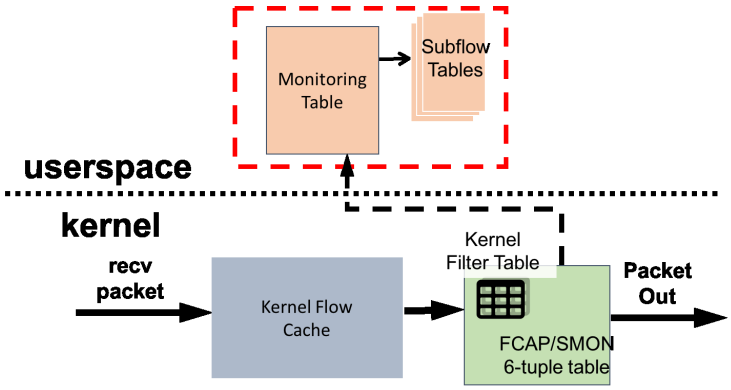

Figure 2.3: FCAP/SMON architecture

clarity. For both FCAP and SMON, we introduce an additional kernel filter table that classifies packets based on the user-defined monitoring tasks. The workflow of FCAP is described in Algorithm 1. Once the packet is determined to be relevant to a monitoring task (*line 1*), the 6-tuple information will be stripped off and kept in the custom 6-tuple flow stats tables (*line 5-8*).

However, the way for FCAP and SMON to store such information is different. FCAP employs a straightforward mechanism by storing the 6-tuple flow stats in a hash table. Linked lists are used to resolve hash collisions, in order to maintain full accuracy. With the hash index, the monitoring thread scans through the linked list to find the flow entry with the same 6-tuple identifier as the incoming packet(*line 4*). Due to its ability to preserve the complete 6-tuple information, the aggregation operations required by monitoring tasks are simplified. Furthermore, the collected statistics are accurate without any loss.

As illustrated in Fig. 2.3, the monitoring pipeline consists of two stages, including a kernel filter table and a 6-tuple table. Only packets finding a match in the filter table are counted towards the latter. The entries in the kernel filter table are populated from the userspace monitoring table. Note that the kernel table differs from the userspace table from two aspects. First, the kernel table employs longest prefix matching to find any rule that matches against the header. Instead, the monitoring rules in the userspace table are

matched against one by one since the monitoring rules may overlap. Secondly, it is not necessary to maintain stats of the headers in this table.

To aggregate the collected 6-tuple flows, we implement a thread that employs similar mechanism with that of the *revalidator* thread in OVS [34]. The thread retrieves the flow stats from the 6-tuple stats table at fixed time intervals and updates the counters associated with the rules in the userspace monitoring table. The credit function is implemented in a similar way as UMON, which credits both flow stats and subflows to the monitoring table. To enhance the efficiency, we further cache the matching results of the 6-tuple information with an extra hash table, where entries expire with the default timeout *ofproto_max_idle* value.

Algorithm 1. **Input:** $FlowStatsTable, flowTuple$

```
1: isMonitored ← LookupMonitorFilter(flowTuple)
2: if isMonitored = True then
3:   hash ← Hash(flowTuple)
4:   bucket ← FindBucket(FlowStatsTable, hash, flowTuple)
5:   if bucket ≠ null then
6:     UpdateFlowStats(FlowStatsTable, bucket, flowTuple)
7:   else
8:     InsertFlowTuple(FlowStatsTable, hash, flowTuple)
9:   end if
10: end if
```

2.3.4 Design of Sketch based Monitoring (SMON)

However, highly accurate statistic is not always affordable and also sometimes may not be necessary. Inspired by previous work, sketches have great potential in reducing memory consumption on end hosts. Sketches are space-efficient probabilistic data structures that are extensively used in streaming applications to process and store summary information. Examples include bitmaps, bloom filters, and count-min sketch, which serve diverse purposes. They provide provable guarantees on the storage usage and error bounds. In previous work,

sketches have been used for traffic monitoring in hardware network devices, where memory is a primary concern. Nonetheless, the performance of sketch monitoring built into software entities remains unexplored. Intuitively, to achieve higher memory efficiency, sketches involve more complex computation logic, e.g., more hash computations, to compress the memory. To investigate the trade-offs between memory/CPU consumption and monitoring accuracy, we propose SMON, a sketch-based mechanism to maintain the compressed 6-tuple flow information. As just mentioned, many research works have focused on utilizing various sketch mechanisms to perform monitoring [35–38]. Among all existing solutions, the bloom filter has a strong space advantage over other data structures. However, the primary drawback of bloom filter is that it does not store the data elements themselves. Therefore, we cannot retrieve the item based on its key, which limits the capability to collect subflows.

Goodrich *et al.* proposed invertible bloom lookup tables (IBLT), which consists of three components in each bucket to store a key/value pair and the corresponding count [14]. In this way, the 6-tuple flow IDs that are hashed into the same bucket are *XOR*ed and stored in a single bucket, as depicted in Algorithm 2. Instead of using linked lists as in FCAP, flows that fall into the same bucket are compressed in order to save memory space (*line 4-6*). In the user space, a customized thread periodically retrieves the bloom filter from the kernel datapath via Netlink socket interface and recovers all flows from the sketch. The size of the sketch structure is adjusted according to the estimated number of flows in each time interval to guarantee successful decoding of flows at a high rate while ensuring a minimum amount of memory usage. The detailed decoding process is explained as follows. It iteratively finds the elements in the bloom filter that contain a single flow and remove its stats from all the other encoded cells that the flow is hashed to, until all the buckets are decoded. Ideally, if the size of the bloom filter is sufficiently large, and exported to the user space at a high frequency, all flows could be successfully recovered. In a cloud setting, where OVSes are deployed at the network edge, each switch only sees a moderate amount of flows, indicating that we can achieve a high decoding rate with a moderate amount of memory. Apparently, the flow decoding time grows with the size of the sketch and the

number of flows in each measurement epoch. Similarly with FCAP, the decoded flows are aggregated into in the user space monitoring table. Besides, the filter table in SMON has the same designs as FCAP. Later we will show in Section 2.4 that with a small amount of memory consumption, we manage to preserve highly accurate statistics.

Algorithm 2. **Input:** $IBLT, flowTuple$

1: $isMonitored \leftarrow$ LookupMonitorFilter($flowTuple$)
2: **if** $isMonitored = True$ **then**
3: **foreach** $k \in [1..H]$ **do**
4: $h_k \leftarrow Hash_k(flowTuple)$
5: **if** IsNewFlow($flowTupe$) **then**
6: CompressFlowID($IBLT, h_k, flowTuple$)
7: **end if**
8: UpdateFlowStats($IBLT, h_k, flowTuple$)
9: **end for**
10: **end if**

2.3.5 Off-path Designs of FCAP/SMON

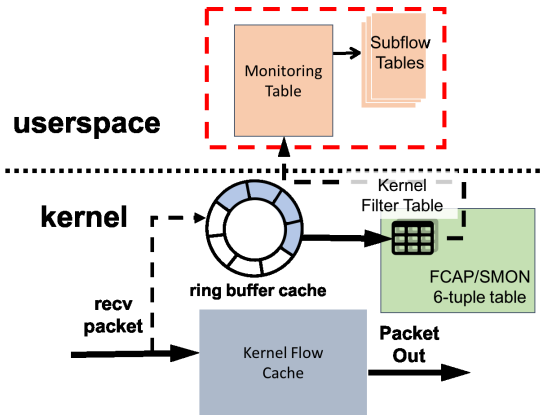

Figure 2.4: Off-path FCAP/SMON architecture

The intuitive designs of FCAP and SMON integrate monitoring logic into the normal

packet forwarding path in OVS kernel datapath, thereby are called on-path designs. Such on-path designs introduce extra processing delay to the OVS forwarding, since monitoring usually requires more complicated processing logic than forwarding, which may further reduce the forwarding throughput.

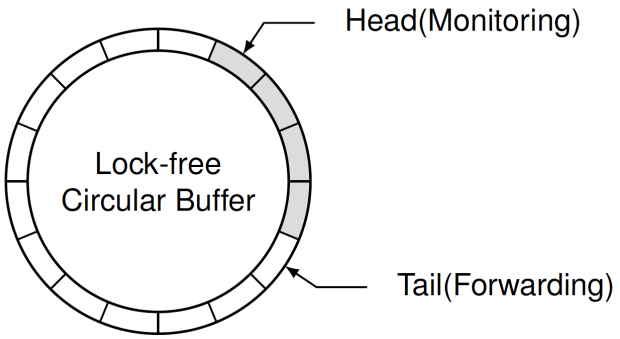

Figure 2.5: Lock-free single-producer single-consumer circular buffer.

To reduce the negative impact, we further embrace a buffering mechanism in order to take the monitoring function off the forwarding path. By using a ring buffer, we aim to decouple the monitoring functions from the forwarding path. We consider this mechanism for both FCAP and SMON, and thus design off-path FCAP and off-path SMON.

The overall architecture of off-path designs is demonstrated in Fig. 2.4. The ring buffer is conceptually a circular FIFO queue with pre-defined size. The ring buffer has two pointers, head pointer for the producer thread and tail pointer for the consumer thread, as depicted in Fig. 2.5. As long as the distance between the two pointers does not shrink to zero nor expand to the full buffer size, both the producer and the consumer could operate on the data in the queue. In our case, the forwarding process is the producer by making a copy of the incoming packet header and storing it to the buffer, while the monitoring thread, as the consumer, fetches the headers for further processing. Therefore, the ring buffer provides a communication channel for the asynchronous interactions between the two functions.

In our implementation, to minimize the performance overhead, we employ a lock-free mechanism to write to the ring buffer with a small probability of overwriting unprocessed data. In this way, forwarding will not be delayed if the monitoring thread cannot keep up with the forwarding process. While reading will be blocked if the buffer is empty, this aims to guarantee the high performance of the forwarding function by making constant-time operations. The lock-free nature helps to ensure that no waiting is involved in adding or deleting data in the buffer. Since we employ overwriting to handle full queues, the collected statistics might not be highly accurate. In order to achieve exactly accurate measurement results, the ring buffer has to be sufficiently large in order to keep up with the ever-increasing packet rates and flow bursts. These will be evaluated in our experiments.

In practice, with the built-in support for cpuset in Linux kernel, one can confine processes to certain processors and memory node subsets so that the monitoring thread and the forwarding thread do not compete for CPU resources. Such optimizations are feasible and practical for data center edge devices where abundant computing resources are available.

2.3.6 eBPF-based Monitoring

eBPF was originated from BPF, the Berkeley Packet Filter. BPF allows to capture and filter network packets that match specific rules, where filters can be implemented as programs and run on a register-based virtual machine. eBPF extends the support to 64-bit registers, among others, and represents an effort to make programmable Linux kernel. That is, one can run sandboxed programs in the Linux kernel without changing kernel source code or loading kernel modules, and thus can be leveraged for monitoring and security, etc.

The eBPF code is executed in an in-kernel virtual machine using a custom 64-bit RISC instruction set, with 11 64-bit registers, a program counter and a 512-byte stack space. eBPF supports running the code as Just-in-Time compiled bytecode, which is verified by an in-kernel verifier to guarantee security (e.g., forbidding loops to ensure program termination and type checks) before loading the code into the kernel. Internally, various mechanisms enable communication between in-kernel eBPF code and user space processes

asynchronously, such as eBPF maps and perf events (FIFO queues). eBPF maps are efficient key-value stores that allow data to be shared within the kernel (i.e., among multiple eBPF programs) or between the kernel and user space.

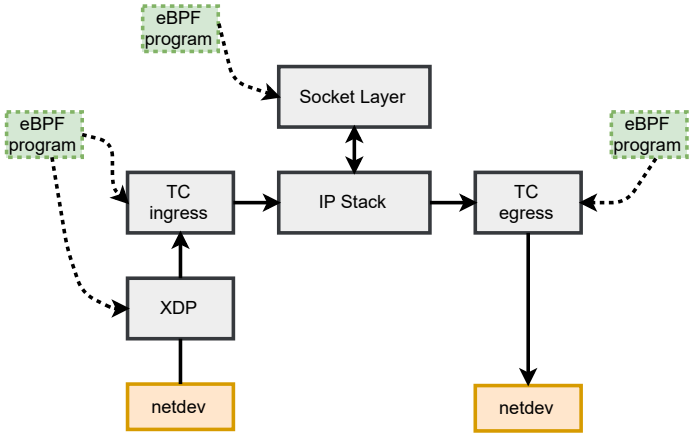

Figure 2.6: Locations of eBPF hooks where monitoring programs can be attached

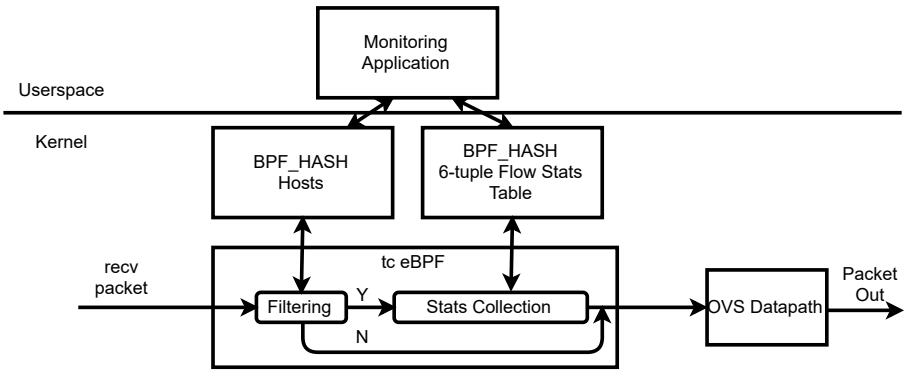

Figure 2.7: Design of eBPF-based monitoring framework.

Figure 2.6 illustrates how eBPF can be used for network traffic monitoring. Specifically, eBPF programs can be attached to different hook points in the networking data path, such as Traffic Control (TC) or eXpress DataPath (XDP) [39], thereby enabling flexible processing

on the intercepted packets. As shown in the figure, ingress traffic can be intercepted in XDP or TC ingress hooks, while the egress traffic can only be intercepted at the TC egress hook, as XDP is not available in egress. Furthermore, eBPF programs can also be attached in the socket layer. Unfortunately, this does not meet the need of monitoring both local and non-local traffic. Therefore, for our purpose the best hook point is the TC layer, where we are able to investigate both ingress and egress traffic. In the following, we discuss the specific design details of our eBPF-based monitoring framework.

eBPF-based Monitoring Framework. As illustrated in Figure 2.7, the framework consists of multiple components, including a monitoring pipeline in the kernel space and a monitoring application in the userspace. The communication between the kernel and user applications is facilitated through a shared eBPF map that maintains the real time traffic statistics. The userspace application retrieves the flow stats from the map and clears the entries at fixed time intervals. Similar to the FCAP/SMON designs, the interval is determined based on the users monitoring demands.

As discussed above, our tc program is attached at the ingress and the egress of a network interface, with full visibility of all inbound/outbound traffic. The eBPF program is executed and the flow stats are updated for each incoming packet. The monitoring workload varies along with the number of hosts to be monitored. For each monitored host, we need to track the number of packets for each 6-tuple flow associated with it. To filter out the hosts, each incoming packet has to go through a monitoring filter before it is counted towards the flow status hash table. More specifically, the monitoring filter examines the destination IP address of the packet and filters out packets that are not monitored (*line 12*). Only packets that find a match in the filter will be counted towards the following flow stats hash table (*line 13-14*). The workflow of eBPF monitoring is outlined in Algorithm 3. Intuitively, such processing may introduce extra performance penalty, which comes in the form of map lookups and updates. In this work, we will conduct in-depth investigation about the performance of the eBPF-based monitoring module and compare it to the aforementioned monitoring alternatives.

Algorithm 3. 1: **procedure** eBPF Userspace Component

2: LoadBPFProgram()

3: PopulateMonitorTable()

4: **while** true **do**

5: Sleep(T)

6: *flowStats* ← BPFMapRead()

7: BPFMapClear()

8: **end while**

9: **end procedure**

10: **procedure** eBPF Kernel Monitoring

11: *flowTuple* ← ParseHeaders(*packet*))

12: *isMonitored* ← BPFMapLookup(*monitorTable*, *flowTuple*)

13: **if** $isMonitored = True$ **then**

14: BPFMapUpdate(*flowStatsTable*, *flowTuple*)

15: **end if**

16: **end procedure**

To understand the root cause of the processing overhead, we first examine the underlying implementation of the eBPF maps. Currently, eBPF is featured with fifteen types of maps to maintain the states across the invocations of the eBPF program and share data among multiple programs or between the kernel and the user space. Two of the most commonly used types are hash maps and arrays, while the other variants serve more complex purposes. As aforementioned, our eBPF-based monitoring design involves multiple data structures, including a monitoring filter and a key-value store for recording flow stats. The design of the data structures for each functional component is critical since our eBPF kernel program lies in the packet processing path and the incurred overhead directly affects the network throughput/latency. In the following, we discuss the designs in more details.

Similar to FCAP/SMON, the monitoring workload is specified in terms of the set of the destination IP addresses of the monitored hosts. BPF_ARRAY and BPF_HASH can both be used for this purpose, while BPF_HASH achieves better performance due to its hash-based design. Therefore, our monitoring filter is implemented based on BPF_HASH,

which can be populated with the host IP addresses from the user space. Furthermore, it can be updated at runtime in accordance with changes in the monitoring workloads. Specifically, in our userspace program, the hash map is initialized with the IP addresses as keys whereas the value is set to 1, before it is loaded into kernel. For each incoming packet, if the destination IP address is present in the hash table, the corresponding flow stats will be updated; otherwise, control flow will follow the original packet processing path. In this way, the host filtering stage can be performed in O(1) time.

On the other hand, to maintain the 6-tuple flow stats information, a hash map (BPF_HASH) is used to store the key/value pairs, where a key represents the flow identifier (e.g., 6-tuple) and a value refers to the packet/byte counts per flow. Since eBPF maps are instantiated inside the kernel, it is critical to keep the size within a reasonable limit to avoid the exhaustion of kernel memory. In the meanwhile, to achieve the desired monitoring accuracy, the actually requested size should be determined based on an sensible estimation of the total number of flows in the monitored network. By default, BPF_HASH has 10240 entries. Since in our monitoring workload, the total number of flows far exceeds this value, the table size has to be explicitly specified during initialization. Unfortunately, eBPF map cannot be resized after it is created. In the latest kernel implementation, eBPF hash maps use pre-allocation by default. The maximum memory size is bounded by the *max_entries* defined by the userspace program during map initialization. Once the map is full, insertions of new keys will fail in order to make sure that the eBPF programs will not exhaust kernel memory. In other words, an underestimated flow count will result in inaccurate measurement results. Therefore, *max_entries* must be carefully chosen in order to accommodate all 6-tuple flows. The actual parameter setting is workload-dependant and will be discussed in detail in Section 2.4.

As a consequence, the performance penalty is mostly incurred by the hash map related operations, including hash computation and hash map updates. Moreover, the exact amount of overhead introduced by our monitoring module should be directly correlated with the actual monitoring workload. A closer scrutinization of the underlying implementation of

the eBPF hashmap APIs further reveals that the kernel hash table is consistently reused. In the eBPF hashmap implementation, linked lists are used to resolve hash collision. Due to this design, the measurement results of the flow stats are accurate as long as there is no packet loss. In the meanwhile, the underlying implementation is optimized for lookup speed. Given the *max_entries*, the hashmap size is always set to the next power of 2. The total memory allocation is *n_buckets * bucket_size + max_entries * element_size*, where *n_bucket* is actual hashmap size and *max_entries* is the maximum number of entries estimated by the user. We will conduct experiments to measure and compare the throughput/latency under various monitoring workloads. A detailed analysis and comparison with other monitoring designs will be presented in our evaluations.

2.4 Performance Evaluation

Our test-bed consists of three Lenovo ThinkServer machines equipped with Intel Xeon 4-Core 3.20GHz CPU and 4GB memory that run Ubuntu 14.04. One machine is dedicated to run the instrumented Open vSwitch (OVS). The second machine serves both as the packet generator and the data sink that receives the data from the OVS. These two machines are connected with two 10Gbps Ethernet cables. As shown in [40], the native OVS can achieve ∼3Gbps switching speed. Thus 10G NIC is sufficient to make it not the bottleneck. We host the packet generator and the data sink on the same machine to facilitate the delay and throughput measurement. The third machine serves as the SDN controller running Ryu [41]. We perform the trace driven evaluation using a CAIDA trace [42] that contains about 30 million packets. The packet trace is replayed using TCPReplay [43] and is fed into the instrumented OVS. The packets are routed and measured by the OVS, and received by the data sink.

2.4.1 FCAP vs. SMON vs. eBPF

In this section we compare the performance of all the monitoring designs. For FCAP and SMON, both on-path and off-path versions are considered. This measurement is conducted

under a packet rate of 160 Kpps with 1400 hosts being monitored. We examine the performance in terms of forwarding latency incurred in the kernel data forwarding path, kernel memory usage, and measurement accuracy. The results are averaged over 10 runs and reported in Table 2.1.

Table 2.1: Comparison of different monitoring designs.

Kernel Monitoring Design	*eBPF*	*On-Path*		*Off-Path*	
		SMON	*FCAP*	*SMON*	*FCAP*
Forwarding Latency	344ns	848ns	515ns	182ns	182ns
Measurement Accuracy	100%	$> 99\%$	100%	$> 99\%$	100%
Memory Usage	396KB	96.97KB	149.58KB	2.116MB	2.168MB

The kernel data-path forwarding latency measures the extra delay introduced by the monitoring modules in the kernel datapath. For off-path FCAP and SMON, we only measure the delay introduced by the ring buffer. The processing delay of the actual stats collection by the monitoring module is ignored since they work off-path. We also measure the overall performance in Section 2.4.3 which shall reflect off-path modules' impact.

As shown in Table 2.1, in general, on-path FCAP/SMON incur longer delays than their off-path counterparts. It takes $182ns$ to put each packet into the ring buffer for off-path monitoring, while on-path SMON and on-path FCAP incur $848ns$ and $515ns$ of delay, respectively. In addition, the eBPF processing takes $344ns$ for each packet. Apparently, the off-path design is the most efficient among all since it only involves a single memory copy operation. Comparatively, on-path designs consume a significantly amount of CPU cycles from hash computation and counter updates, resulting in much longer processing delays. Among the three on-path designs, eBPF outperforms the other two due to its highly performant underlying implementation. As discussed in Section 2.3.6, eBPF hashmap size is kept sufficiently large in order to minimize the length of the linked lists. Thereby, the average per-packet latency is considerably smaller than FCAP. Compared to eBPF and FCAP, on-path SMON incurs much long processing delay since sketch encoding requires

multiple hash computations and memory access to multiple counters for each incoming packet.

In terms of memory consumption, on-path SMON consumes less memory ($96.97KB$) than on-path FCAP ($149.58KB$) and eBPF ($396KB$). By utilizing sketches, SMON is the most memory-efficient by compressing multiple flow information into a single sketch counter at the cost of slight accuracy degradation. Between FCAP and eBPF, the latter requires more memory usage due to its large hash table size and the memory pre-allocation mechanism. Compared to the on-path designs, off-path FCAP and SMON consume the largest amount of memory since they require a ring buffer to store all incoming packets.

We next examine if the use of ring buffer and sketch reduces the measurement accuracy. As shown in Table 2.1, the results show that the off-path measurements achieve comparable accuracy as the on-path measurements as long as the ring buffer is sufficiently large to accommodate incoming packets. We find that in order to avoid packet losses, for a packet rate of 160 Kpps, the memory allocated for the ring buffer must be over 2MB. The size of the ring buffer can be configured from the user space through the Netlink interface according to the estimated packet rate and the desired accuracy. The measurement accuracy of SMON is over 99%, which suggests the use of sketches does not lead to large accuracy loss. Moreover, on-path/off-path FCAP and eBPF provide fully accurate measurement results since they both employ linked lists to resolve hash collisions in their implementation.

2.4.2 Impact of Monitoring Workloads

Here we examine the impact of monitoring workloads on the CPU utilization and memory usage of instrumented OVSes. We vary the number of monitored hosts, i.e. IP addresses, which directly leads to a varying number of monitored micro-flows, as listed in Table 2.2 and Table 2.3. We also experiment with two different packets rates, 80 Kpps and 160 Kpps, replayed by TCPReplay to represent different OVS switching workloads.

Figure 2.8 and Figure 2.9 depict the CPU utilization overhead caused by the monitoring activities against the number of monitored hosts at two packet rates. The reported results

Table 2.2: Memory usage (MB)(packet rate = 160 Kpps).

#hosts	200	400	600	800	1000	1200	1400	1600
#flows	346	703	1067	1402	1698	2082	2550	3043
off-path SMON	2.039	2.049	2.063	2.079	2.087	2.100	2.116	2.132
off-path FCAP	2.075	2.091	2.107	2.118	2.133	2.149	2.168	2.191
eBPF	0.105	0.105	0.160	0.211	0.262	0.324	0.387	0.449
UMON	4.556							

Table 2.3: Memory usage (MB)(packet rate = 80 Kpps).

#hosts	200	400	600	800	1000	1200	1400	1600
#flows	217	487	641	811	980	1219	1456	1712
off-path SMON	1.347	1.358	1.360	1.366	1.371	1.379	1.387	1.395
off-path FCAP	1.386	1.393	1.402	1.410	1.418	1.426	1.438	1.469
eBPF	0.043	0.063	0.121	0.152	0.184	0.215	0.246	0.281
UMON	1.589							

represent the total CPU utilization of all related threads including handlers, revalidators, and new threads created for monitoring purposes. In FCAP and SMON, we create a user-space thread called *collector* to collect the flow stats from the data structures exported from the kernel datapath at fixed time intervals. Moreover, for off-path FCAP/SMON, there is a kernel thread that retrieves packets from the ring buffer. Differently, eBPF monitoring threads are not incorporated into OVS, so we measured their CPU usage separately. In all experiments, the stats collection interval is set to 0.5 second.

In UMON, the flow aggregation functionality is integrated into the existing revalidator threads. CPU utilization of the two on-path designs is not shown in the figure since the implementation of the monitoring modules is similar to their off-path counterparts, resulting in similar CPU utilization.

As illustrated in Figure 2.8 and Figure 2.9, the CPU utilization overhead increases as

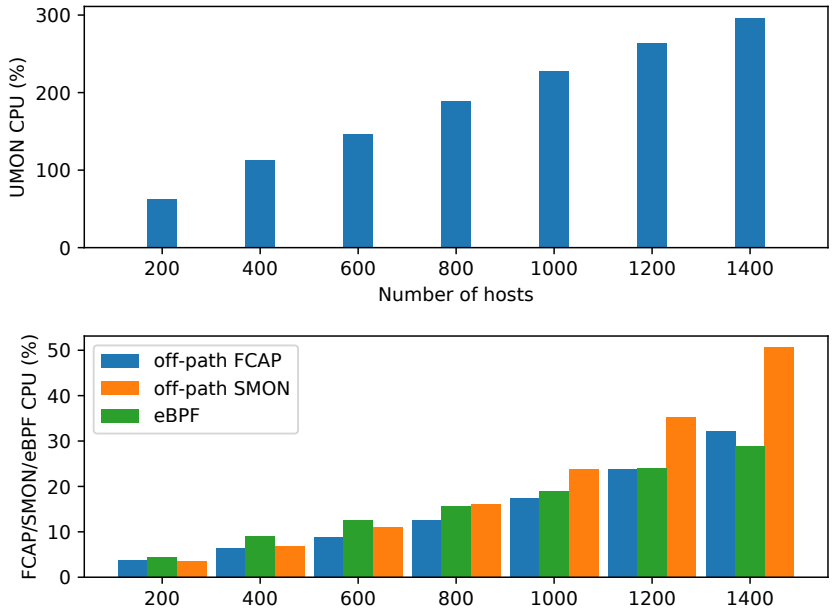

Figure 2.8: CPU overhead under various monitoring workloads with packet rate 160 Kpps.

the number of monitored hosts and the packet rate increase. In addition, off-path FCAP incurs the least amount of CPU utilization overhead, while eBPF incurs comparable but slightly larger CPU utilization overhead in the average case. The difference between the two is attributed to the different underlying implementation schemes for the communication between kernel and userspace. As discussed in Section 2.3, in FCAP and SMON, the kernel/userspace communication is facilitated via the Netlink socket interface, following the same communication mechanism as OVS. Comparatively, with eBPF, at fixed time intervals, the entries are accessed and cleared from the userspace via system calls. Compared to FCAP and eBPF, SMON requires complex sketch decoding operations performed by the *collector* thread, resulting higher CPU utilization. The overhead introduced by UMON is significantly larger than those of off-path FCAP/SMON, which is mainly due to two reasons. First, since UMON installs fine-grained forwarding rules in the kernel flow table, there are more frequent packet misses such that its userspace handler threads are busy with handling

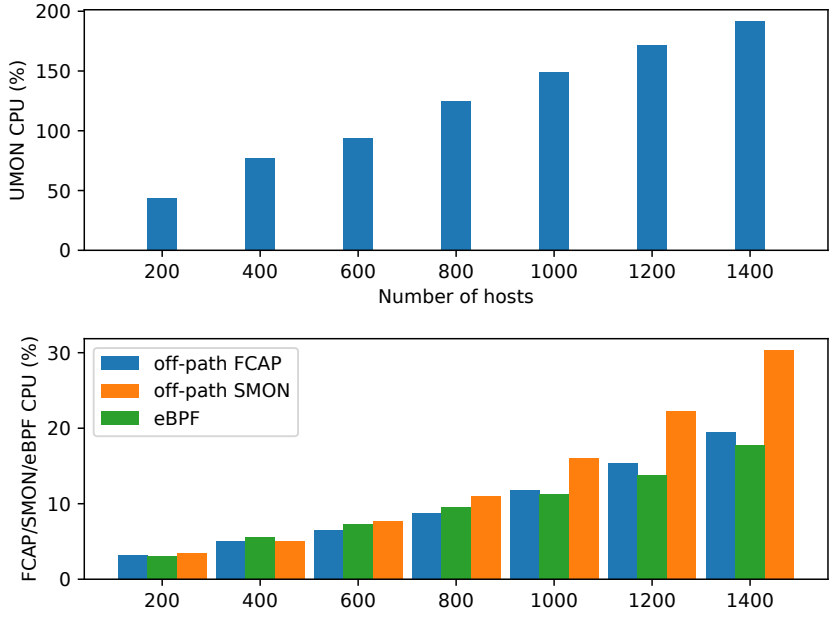

Figure 2.9: CPU overhead under various monitoring workloads with packet rate 80 Kpps.

upcalls. Second, with a large kernel flow table, the revalidator threads in UMON are also heavily loaded with updating the flow stats into the userspace monitoring table. On the contrary, in off-path FCAP/SMON, the kernel flow table only contains two flow rules in our experimental setup, thus imposing negligible CPU overhead to the userspace handler and revalidator threads. The CPU utilization in these two cases is mainly attributed to the flow stats aggregation performed by our custom *collector* thread.

We next evaluate the memory overhead for monitoring. Since on-path FCAP/SMON use less memory than their off-path counterparts, we focus on the two off-path schemes and compare them to UMON.

The results are shown in Table 2.2 and Table 2.3 for two different packet rates. The memory size in the table is the amount of memory used in the kernel for the monitoring purpose. As the memory usage dynamically changes with incoming flows for all schemes, we take the peak usage in comparison. The memory consumption in the off-path FCAP/SMON

is caused by two data structures: a kernel ring buffer that caches incoming packets, and the actual data structures (i.e., hash table in FCAP and sketch in SMON) to maintain the flow stats. As previously mentioned, sufficient amount of memory needs to be allocated to the ring buffer in order to avoid packet losses. The experimental results show that off-path FCAP consumes about 3% more memory than off-path SMON. For eBPF, the memory consumption is mostly incurred by the BPF map to record flow statistics. For the sake of memory pre-allocation, the userspace program needs to specify the approximate maximum number of flows in each time interval at map initialization. Since this number varies across workloads and packet rates, the memory consumption of eBPF also varies accordingly. In Table 2.1, it indicates that eBPF requires more memory than on-path FCAP, largely due to the underlying eBPF hashmap implementation. UMON uses much larger amount of memory than all other monitoring designs. This is because in order to support comparable measurement accuracy, the kernel flow table in UMON has to maintain an individual forwarding entry for each 6-tuple flow. Consequently, the memory usage greatly exceeds the other solutions.

2.4.3 Switching Throughput and Latency

To study the throughput and latency of our designs under high packet rates, we use DPDK based packet generator MoonGen for traffic generation and measure the maximum achievable throughput for each measurement framework when there is no packet loss and the monitoring stats are highly accurate. Since only those packets with a match in the monitoring table will be counted towards the hash table/sketch, the ratio of monitored packets to the total number of packets directly affects the throughput of the entire system. To study this impact, we conduct experiment by varying the number of hosts in the kernel space monitoring table. The throughput and latency results under different workloads are shown in Figure 2.10 and Figure 2.11, respectively.

First and foremost, the switching throughput of UMON is the lowest among all monitoring designs, as can be seen in Figure 2.10. This could be explained by the fact that UMON

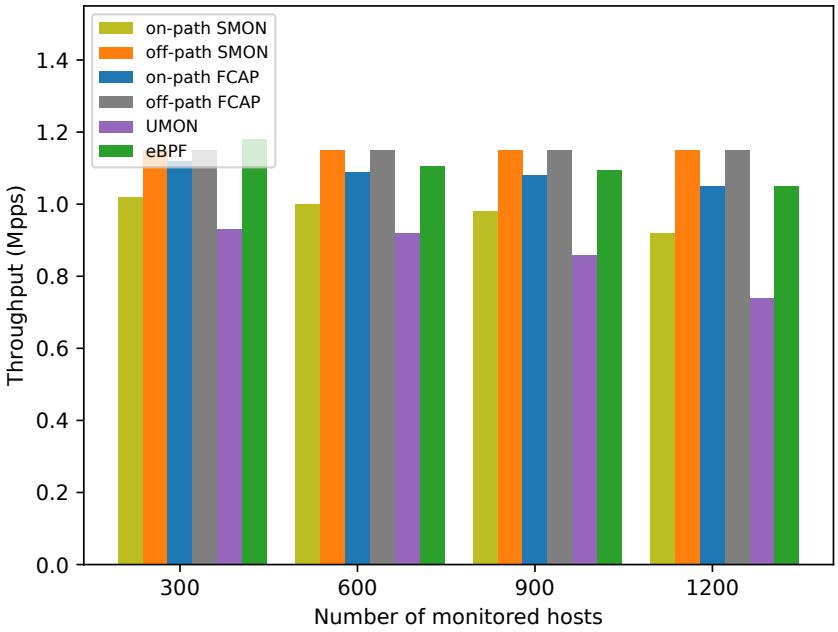

Figure 2.10: Throughput(Mpps) of different schemes under various monitoring workloads.

follows the traditional design of OVS kernel datapath, which requires the first packet of each new flow to traverse the slow path through the userspace. The userspace of UMON introduces an extra monitoring table in the forwarding pipeline. The forwarding rule and monitoring rule are combined to generate a more fine-grained flow that would be installed into the kernel cache table. Due to this design, an enormous amount of flow cache misses are introduced when we need to collect the flow statistics for each 6-tuple flow. On-path FCAP/SMON both outperform UMON but lag behind compared with eBPF and their off-path counterparts. As explained previously, this is because the monitoring modules (filtering, stats collection) are placed in the switch forwarding path. Regarding the on-path versions, FCAP achieves higher throughput than SMON, since the latter requires complex sketch encoding operations, while the former implements more light-weight hash tables. On the other hand, off-path FCAP/SMON achieves higher throughput because monitoring logic is decoupled from forwarding and the overhead only involves the memcpy from the

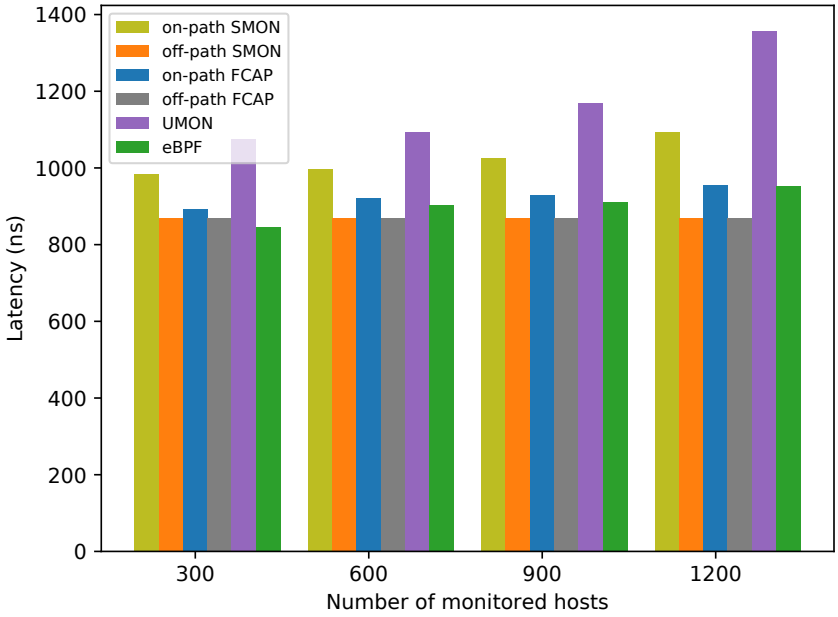

Figure 2.11: Average latency(ns) of different schemes under various monitoring workloads.

ring buffer. This also explains why FCAP and SMON achieve the same throughput in the off-path paradigm. Finally yet importantly, despite the fact that eBPF and on-path FCAP both are on-path and implemented based on hash tables, eBPF achieves slightly better performance than on-path FCAP as a result of its optimized hash table implementation within the Linux kernel. Due to the large size of the hash table, it has a shorter linked list for each bucket in the hash table in the average case, resulting in higher efficiency for hash lookups/updates. Overall, from the network performance perspective, off-path FCAP/SMON is a preferable solution among all the designs.

Following the same experimental methodology, we also measured the average switching latency and investigated the impact from various monitoring workloads. Figure 2.11 reveals that off-path designs incur minimum delay, while UMON significantly degrades the forwarding efficiency. Further, the switching performance worsens along the increase of the workloads. Likewise, eBPF yields smaller latency than on-path FCAP/SMON for the

same reason as throughput. Off-path options outperform all other alternatives with more memory consumption. More specifically, the memory required (15MB ring buffer) scaled linearly (7.5x) with the increase in throughput to 1.2 Mpps from our earlier experiments at 160 Kpps. We conclude roughly one MB is needed for each 80 Kpps of throughput.

2.5 Discussion

Table 2.4: Comparison of different frameworks.

Designs	eBPF	*On-Path*		*Off-Path*		UMON
		SMON	*FCAP*	*SMON*	*FCAP*	
CPU Overhead	low	moderate	low	moderate	low	high
Memory Consumption	low	low	low	moderate	moderate	high
Measurement Accuracy	precise	high	precise	high	precise	precise
Forwarding Latency	high	high	high	low	low	high
Implementation Complexity	low	high	high	high	high	low

Based on the evaluation results, it is clear that building monitoring capabilities into software entities on the edge servers, either software routers or independent monitoring modules, is feasible without sacrificing significant performance overhead. Nevertheless, each design bears its own pros and cons. Although it is a seemingly daunting task to determine which design achieves the overall best performance, we have attained several insights, as sketched in Table 2.4, into the design of software-based measurement framework.

First, UMON requires the least implementation efforts with no modifications to the OVS kernel datapath. However, it derives fine-grained forwarding rules by combining the forwarding and monitoring functionality, which leads to the heaviest CPU load in the user space. In addition, it necessitates significantly more memory consumption.

Second, FCAP outperforms both SMON and eBPF in all aspects except that it needs to instrument OVS kernel code. Particularly, on servers with sufficient memory resources, off-path FCAP is better suited than its on-path version, since it introduces minimal impact

on OVS throughput and latency, among all other alternatives.

Third, SMON is the most memory-efficient option, owing to the strong space-efficiency of bloom filter sketches. Although sketches have proved to be efficient in memory-constrained hardware devices, it turns out concerns have shifted away for building monitoring logic into the end hosts. Because of its high CPU utilization, it is not ideally suited for commodity servers with abundant memory resources.

Fourth, eBPF-based monitoring design achieves comparable performance with on-path FCAP from the perspective of CPU utilization, measurement accuracy and switching performance. In the meantime, it requires minimal maintenance efforts since it executes independently of OVS and can be configured and updated without interrupting the system operations. It necessitates more memory than on-path FCAP due to the difference in the underlying hash table implementation. Nevertheless, its overall performance falls behind the off-path designs since the eBPF program lies in the packet processing path.

Since all these schemes are designed for software defined measurement, we can see that:

In terms of the switching throughput and latency, off-path designs offer the best performance, regardless of the monitoring algorithms, since throughput and latency are only affected by the ring buffer write operations. Without ring buffers, on-path FCAP/eBPF achieve comparable throughput/latency, which demonstrate that hash tables could suffice in a software monitoring system;

In terms of implementation complexity and portability, eBPF is the best bet since monitoring programs could be loaded and updated at runtime, while OVS-embedded designs require to recompile and reinstall the OVS binary whenever there is an update.

By comparing the results across all the experiments, we observe that

- by decoupling the monitoring functionality from the kernel forwarding path, off-path schemes can achieve better switching performance than on-path schemes in terms of network throughput latency, while achieving the same measurement accuracy at the cost of higher memory consumption.

- in the design of flow stats collection module, our results demonstrate that hash table

is a more efficient solution compared to sketch due to its lower computational cost, which is a major factor in the evaluation of CPU utilization.

While there is no scheme that outperforms in all aspects, building monitoring frameworks into edge servers requires us to carefully examine the interplay of multiple key factors, including memory and CPU consumption, measurement accuracy, impact on switching throughput and latency, maintenance complexity, and so on so forth. Our empirical study demonstrates that hash tables are a better fit than sketches in a software monitoring framework despite of the strong memory-efficiency and wide utilisation of the sketches in hardware environment. The difference lies in the fact that hardware devices have much tighter memory constraints than commodity servers. In terms of the placement with respect to the forwarding pipeline, off-path outperforms on-path since the latter introduces noticeable latency to the traditional packet forwarding pipeline. Such impact becomes even more evident under high packet rate.

2.6 Chapter Summary

Software defined networking has provided more flexibility for network measurement and monitoring, and enabled software defined measurement. In this work, we have investigated various design options to implement software based measurement using Open vSwitches. However, enabling monitoring capability on the widely deployed OVSes in data centers has to take into a number of factors into consideration, including resource consumption, impact on the forwarding, measurement accuracy, implementation complexity, portability etc. We have empirically explored the various trade-offs among these factors by designing, implementing, and evaluating four different monitoring schemes and quantitatively shown their advantages and disadvantages. These results provide insightful guidelines for conducting network traffic measurement on the OVS as well as software defined measurement in general.

Chapter 3: BotSifter: A SDN-based Online Bot Detection Framework in Data Centers

3.1 Introduction

Driven by the great success of cloud computing, the last decade have seen the continuous migration of various services and applications to different cloud platforms. With the flexible pay-per-use model, more and more end users also rely more and more on the cloud platforms for their personal storage and computing need [44]. Under this trend, bots and botnets-as-a-service [45] are no exception: bots and botnets have been one of the most severe Internet threats underpinned by the economic motive, and recent years have witnessed the emergence of cloud-hosted botnets [46] largely due to its cost effectiveness and the long-term availability of the machines in the data centers. Compared to traditional bot machines which get switched on/off frequently by the end-users, data center hosted bot machines usually stay online for longer periods of time [46] and generate more attack traffic (and profit). It was previously reported [47] that cybercriminals have managed to install DDoS botnets in AWS by exploiting a vulnerability in Elasticsearch [48], an open source search engine that is often deployed in cloud environments such as Amazon EC2, Microsoft Azure, Google's Compute Engine. Likewise, botnet command and control software has previously been found to be hosted in Dropbox [49]. Apart from these particular cases, they show that cybercriminals are now breaching commodity data centers, seeking the values of cloud infrastructures to host the botnets.

Cloud-hosted bots (and thus truly botnets-as-a-service) are more harmful than their traditional counterpart, yet accurate and expeditious botnet detection schemes on cloud platforms are not on the horizon yet because of the following challenges. First, since the

cloud is a multi-tenant environment, the detection of bots should be fast (e.g., at runtime to prevent further damage) and as non-intrusive as possible so that the detection would have no or trivial impact on the normal data center applications. Second, given the large traffic volume in a data center, such a detection scheme must be scalable, capable of handling tens or hundred gigbit line rate. Third, the detection must be very accurate, since compared to the end user environment, the cost of misclassifying a bot process or connection (and subsequently closing/blocking the connection) in a data center could be much larger or even disastrous.

Some of the early-day bot detection schemes [50] [51] rely on deep packet inspection, which suffer from high resource demands and ineffectiveness against encrypted botnet traffic. Some others [52–56] focused on host traffic and bot behavior analysis. These schemes became less and less effective since contemporary botnets are constantly evolving to circumvent the advanced detection mechanisms [57–61]. For example, most botnets abondoned the traditional IRC based Command and Control (C&C) channels and embrace HTTP or P2P for communications.

To deal with such challenges, lately more schemes have been built by taking advantage of the machine learning (ML) techniques for detecting bot activity and/or command-and-control (C&C) channels [52,53,55,56,61,62]. Such schemes often demand intensive resources to capture the incoming and outgoing traffic information, e.g., a centralized traffic monitoring facility at the network gateway or the firewall, and then run the detection after the traffic features are extracted. These schemes may work for a small network with medium or low traffic volume, but they are hardly effective in detecting the cloud-based bots because (1) they demand a lot of extra resources for effective traffic monitoring, which is hardly scalable considering the huge traffic volume in data centers, (2) their accuracy varies with the used machine learning model and the chosen features, and they are often incapable of identifying unseen bots without understanding bot invariant characteristics (e.g., C&C channels), and (3) they provide little understanding on different phases of bot activities, which is essential for identifying the malicious intent of bots in their early stages. This is

however very desirable for data centers in order to prevent further damage.

More importantly, existing ML-based approaches focus on the detection of botnet flows [55, 56]. However, in the detection of cloud-hosted bots, accurate identification of the bots is more imperative compared to the detection of individual malicious flows. Only after the bots are identified, the infected VMs could be shut down to prevent future attacks. To our best knowledge, no prior works focused on the detection of VM-based bots in the cloud.

To this end, we propose to build BotSifter towards a scalable and accurate runtime bot detection framework in data centers. To be scalable, BotSifter integrates centralized learning (thus to have a global view of the traffic and centralized intelligence) with distributed edge-assisted detection by leveraging the software switches in data centers. Due to their widespread deployment in data centers, software switches (e.g., Open vSwitch) are being increasingly employed as monitoring devices as they typically reside in commodity servers with abundant hardware resources. Since they are usually deployed at the edge of the monitored network and located within close proximity to the end hosts, only relatively a small amount of traffic flows traverse the switch, rendering it a more scalable solution for anomaly detection in data centers. Implementing detection at the edge also enables our system to observe both directions of the traffic, which is an essential prerequisite for connection based anomaly detection. Moreover, our edge-based detection framework has the inherent capability of observing the internal attack traffic and the C&C traffic within the data center network.

To achieve high accuracy, BotSifter not only conducts neural network (NN) based bot activity detection, but also conducts the detection of C&C channels in parallel. These detections are further enhanced with local and network wide correlations to minimize false alarms. Not only detecting the existence of C&C channels, BotSifter is also able to differentiate different communication protocols (i.e., IRC, HTTP, P2P) utilized by the bots so that custom mitigation mechanisms could be quickly deployed to defend against specific types of bots. Since the majority of the detection operations in BotSifter are conducted within the software switches that are instrumented to collect connection features and states on

the fly, BotSifter is highly efficient in identifying bots without interrupting other network functions, thus making it a practical solution for the cloud and data center systems. A prototype of BotSifter is implemented, and the evaluations based on the real-world traces show BotSifter is highly accurate and efficient in detecting bots utilizing known protocols, but also bots with customized protocols.

The highlights of BotSifter lies in

- The design naturally utilizes the SDN's centralized structure to have a global visibility while distributing most of the detecting load to the network edge to be scalable.
- It builds the monitoring capability in the OVS via an off-path traffic collection design to minimize the intrusion of traffic monitoring to normal applications.
- It utilizes neural network to detect bot activities, and employs newly designed protocol specific mechanisms (e.g., self-correlation for P2P) to detect and differentiate different C&C protocols. The local and network wide correlations further enhance the accuracy and robustness of the detection.

The remainder of this chapter is organized as follows. Section 3.3 describes the BotSifter design and section 3.4 sketches its implementation. The evaluation is presented in section 3.5. We make concluding remarks in section 3.6.

3.2 Background

Open vSwitch Open vSwitch is software implementation of a multi-layer virtual switch designed to support networking in virtual environments, which are widely deployed in data centers nowadays. Its major components include a kernel space module and a userspace daemon. The userspace daemon serves as a full but slow path of forwarding while the kernel module acts as a forwarding cache. Incoming packets are firstly matched against the kernel forwarding cache before they traverse the userspace forwarding pipeline. Only packets without matches in the kernel forwarding table are forwarded to userspace.

Botnet Lifecycle Generaly, a botnet involves three parties: botmasters (or botherders), C&C servers, and bots. Botmasters are cybercriminals who are largely driven by economic profits to infect and recruit compromised hosts by exploiting known vulnerabilities of target systems. C&C servers are in charge of issuing commands to the compromised machines to coordinate their activities and receive reports back from them. Typically, the topology of botnets can be categorized into two types: centralized and decentralized. In centralized botnets, C&C severs are centralized computers while in decentralized botnets, the compromised machines serve as both bots and C&C servers. According to [63], the lifecycle of a botnet can be classified into four phases, including formation, C&C, attack and post-attack. In the formation phase, the botmaster infects the target machine, gains complete access to the system, runs scripts to install malicious binaries. In the C&C phase, bots communicate with C&C servers to retrieve new instructions. In attack phase, they launch attacks according to the instructions received from C&C servers. Post-attack phase is mainly for botnet maintainence. For example, they may download and update their software for more advanced evasion techniques.

The communication between bots and C&C servers are facilitated by a large variety of protocols. Among them, the most prominent ones include IRC, P2P and HTTP. Besides, it is also not a rare case that botnet developers customize their own protocols to leverage their capabilities of control, which will also be discussed in our work. IRC is the primary method for botnet control when botnets started emerging a few decades ago mostly because they are easy to deploy and they allow interactive control of the bots. Along with these advantages, IRC C&C becomes quite outdated nowadays since it is much easier to detect and taken down than web-based C&C as well as decentralized protocols. In more recent botnets, they are more likely to employ HTTP/HTTPS based C&C mostly because there are more potential methods to hide they command and control messages while traditional IDS systems are likely to regardd them as normal web traffic. Even worse, modern fast flux techniques make it even more difficult to track. In a nutshell, HTTP based C&C becomes much more prevalent for botnet control than the ealier IRC based C&C. Our analysis will

certainly not skip the IRC based botnet for completeness.

Compared to the above mentioned centralized control schemes, peer-to-peer botnets are hard to develop but they can provide more stable and robust control network. Our work mainly focuses on these three types of control protocols. Since the botnets are evolving and take advantage of stealthy communication techniques in order to evade detection, the detection of the C&C channels becomes really challenging nowadays.

Cloud-based Botnets. Cloud services refers to services made available to users by cloud service providers (CSP), ranging from software services, such as databases, to raw compute resources, such as storage and processing power. Cloud services provide numerous benefits, including reduced costs for hardware resources, enhanced security for users' data, simplified management and maintenance, and so on. Some of the most widely known CSP include Microsoft, Google, Amazon Web Services(AWS). Along with the benefits, much attention has been turned to the security of cloud services in recent years. Nonetheless, most of the research mainly focuses on how to protect the legitimate cloud users from external attackers. Until recently, there is continuous reports about botnets gradually stepping into the cloud due to the same reason of how Cloud provides benefits for normal applications. Compared to traditional botnets, Cloud-based botnets are a lot more cost-efficient and robust due to the high reliability as well as scalability of the cloud. Particularly, our work mainly concentrates on the detection of cloud-based botnets by incorporating edge-assisted detection enabled by the inherent distributed structure of the data centers.

3.3 BotSifter Design

The design of a cloud botnet detection framework that is capable of monitoring thousands of servers, tens of thousands of Virtual Machines (VMs) running on these servers, and terabit per second communication among these entities and between the data center and the outside Internet is extremely challenging. In this work, we explore a machine learning based distributed online cloud bot detection framework, called BotSifter, that monitors and detects the bots within a data center. BotSifter takes advantage of Software Defined Networking

(SDN) architecture widely adopted by the data center, and integrates the centralized Machine Learning (ML) training with distributed monitoring and detection. The ML based bot detector is trained at the central controller, and runs distributedly at the cloud servers with the ML configurations acquired from the central controller. SDN software switches, e.g., Open vSwitch (OVS), are instrumented with traffic monitoring capability. Both bot activity detector and bot C&C communication detector are implemented at servers using locally collected traffic stats. If necessary, the central detector is invoked to detect data-center wide botnets.

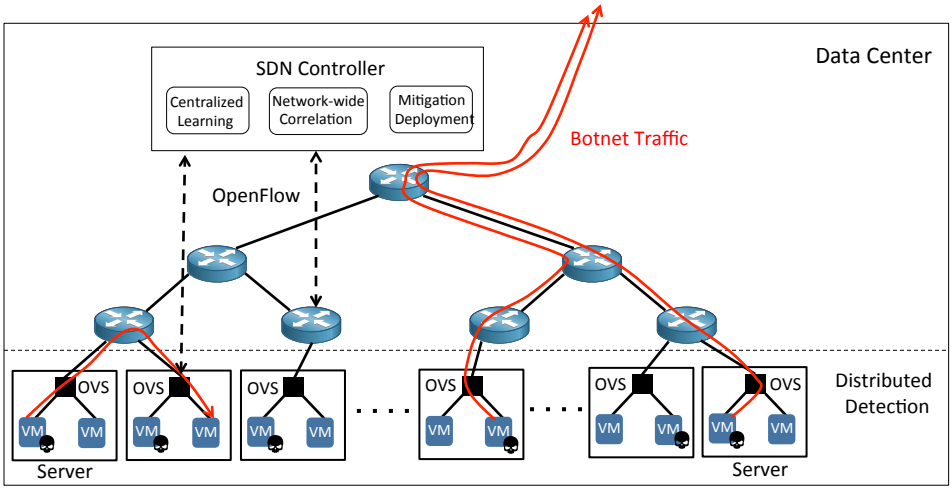

Figure 3.1: Overall architecture of BotSifter

The key feature of our design is to leverage the data-center servers and the software switches running on these servers. The contemporary servers have abundant computational and memory resources, and the software switches have the full access to the traffic originated from and destined to the end hosts residing on the servers. Our design philosophy is to place as many monitoring and detection functionality at the edge servers as possible, while judiciously utilize the central controller for data-center wide tasks when necessary.

The BotSifter architecture is depicted in Fig. 3.1. The central controller is responsible for ML training, network wide bot detection, and bot mitigation. At individual servers, traffic features are extracted and recorded by the instrumented software switches, which are then used by C&C channel detection modules and ML based bot activity detection module. Below we describe local traffic monitoring, C&C bot detection, ML based bot detection, and network wide bot detection and mitigation, respectively. The major components to accomplish these tasks are sketched in Figure 3.2.

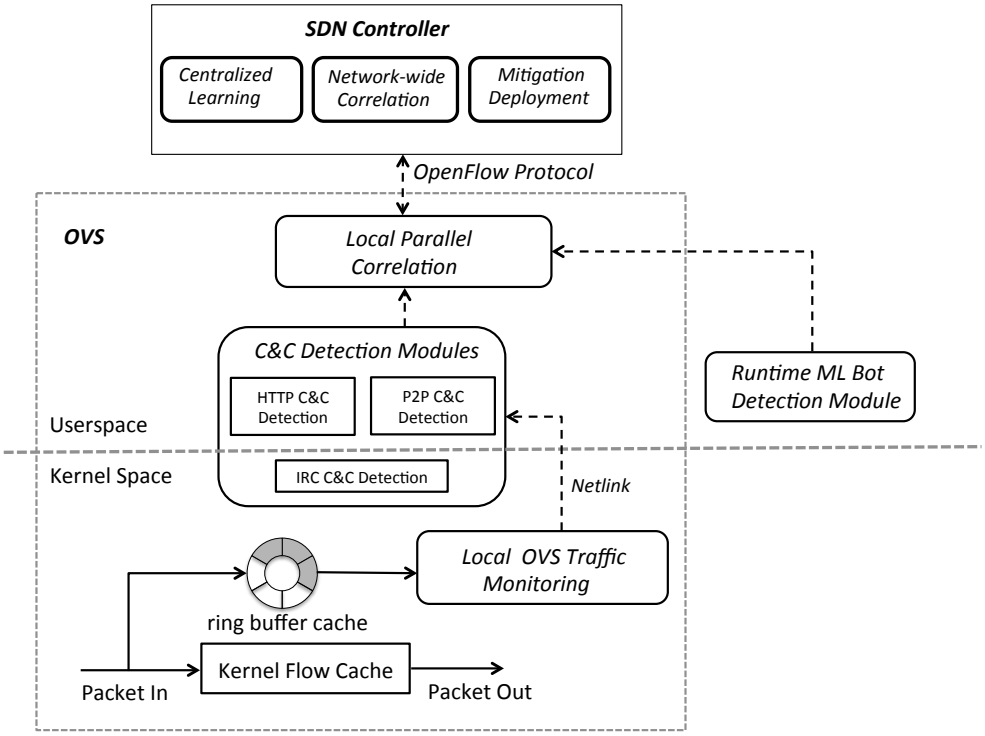

Figure 3.2: BotSifter design: major components

3.3.1 Local OVS Traffic Monitoring

To lessen the impact on software switch's forwarding speed while efficiently capturing the traffic stats required by ML bot detection module and C&C detection modules is the main

design challenge for local OVS based traffic monitoring. OVS maintains a user-space forwarding pipeline and a kernel space forwarding cache. The majority of the incoming packets are forwarded by the kernel module, with few packets that does not have the matches in the kernel being redirected to the user-space. To decouple the monitoring from the forwarding, a ring buffer cache, as shown in the bottom of Fig. 3.2, is introduced. Such a design significantly reduces the monitoring interference to the forwarding.

ML detection module and different C&C detection modules requires different traffic stats. For instance, ML module uses a traffic feature vector, while P2P C&C detection keeps track of DNS transactions for each source/destination pair. Multiple hash tables are employed for efficient lookup and update.

3.3.2 Parallel C&C and Bot Activity Detection

At edge server, BotSifter implements one ML based bot detection module and three C&C detection modules: HTTP C&C detection module, P2P C&C detection module, and IRC C&C detection module, respectively. We have the design choice of implementing them inside the software switch OVS, or on the server but outside the OVS. In addition, if a module is placed inside the OVS, we need to decide whether the module resides in the kernel or at user-space. Since P2P and HTTP C&C modules use the connection stats rather than individual packet info, placing them in the user-space at OVS is the right choice. In contrast, IRC based C&C module conducts keyword search over a packet, thus is impleneted in OVS kernel. ML based bot detector uses TensorFlow ML libraries, thus is implemented in the user-space outside OVS. We also place the local parallel correlation module in the OVS, as shown in Fig. 3.2.

3.3.3 C&C Detection

P2P C&C detection

Distinguishing P2P traffic used by bots from normal P2P traffic is not trivial. The study in [54] made the discovery that the sets of peers contacted by two different bots within the

same botnet typically have a much larger overlap compared to the peer sets contacted by two legitimate P2P clients within the same network. In practice, there is the possibility that only one bot resides in the data center.

We develop a client self-correlation approach to detect P2P bots. Specifically, the peer set of a P2P bot is more likely to remain stable over the time, while that of a normal P2P client is more likely to change due to the changing user behavior (e.g., downloading of disparate resources over time). We conduct extensive empirical experiments, and the results show that P2P bots do exhibit strong self-overlap patterns while normal P2P hosts do not. The new approach is more flexible and robust than the approach in [54].

HTTP C&C detection

Bots tend to communicate with the server periodically over the HTTP channel [64]. HTTP C&C detection module makes the detection using such periodic traffic pattern. The timing information of HTTP connections between a pair of end hosts is collected by the monitoring module. The number of HTTP connections within each time slot is counted. The time series of connection counts is fed into a Discrete Fourier Transform (DFT). If periodic pattern is discovered, the host becomes a bot candidate.

IRC based C&C detection

Although IRC based botnets are diminishing nowadays, we include the IRC C&C detection for the sake of completeness. To minimize system overhead, a combination of keyword matching and port numbers is leveraged to identify IRC connections. Furthermore, to determine whether they are IRC C&C channels, our detection relies on inspection of packet payloads by searching for attack relevant commands in IRC response messages. We implement the detection module at the kernel space of OVS.

We mainly focus on the systematic design of the detection framework rather than discovering every possible C&C communication pattern to detect all existing botnet variants. There are certainly cases where sophisticated botnet variants disguise their communication pattern through stealthy C&C to evade detection. Correspondingly, our detection framework is extensible in integrating new detection modules to account for such novel

C&C patterns. Alternatively, for botnets with unrevealed patterns, we can resort to large scale network wide correlation for detection. For example, for P2P C&C without sufficient self-overlap, flow stats for all P2P hosts can be exported to perform off-line network-wide clustering. This relies on the fact that normal users tend to have divergent usage patterns, resulting in non-overlapped peer sets among different users.

Runtime ML based bot detection

In parallel to the C&C channel detection, BotSifter also conducts bot activity detection using a deep learning Neural Network (NN). For each connection, NN is able to detect if it is potentially a bot activity connection. Instead of integrating this function into switches, it runs as a separate process on the sharing host of OVS. The motivations of such a design are two-folds: 1. Including non-standard library into OVS for composing such function diminishes portability of BotSifter. 2. Updating the model from the centralized controller becomes effortless without interrupting network functions of OVS.

In order to detect if a host is compromised and becomes a bot, BotSifter keeps track of the percentage of connections initiated by this host that are identified by the NN as the bot activity connections. If the percentage surpasses a preset threshold, the host is identified as a bot.

Algorithm 4. **Input:** $list_{p2p}, list_{http}, list_{irc}, list_{ml}$

Output: $BlackList, GreyList_{http}, GreyList_{ml}$

1: $BlackList \leftarrow \emptyset$

2: $GreyList_{http} \leftarrow \emptyset$

3: $GreyList_{ml} \leftarrow \emptyset$

4: $list_{c\&c} \leftarrow list_{p2p} \cup list_{http} \cup list_{irc}$

5: **foreach** $h \in (list_{c\&c} \cup list_{ml})$ **do**

6: **if** $h \in list_{c\&c} \cap list_{ml}$ **then**

7: $BlackList \leftarrow BlackList \cup h$

8: **else**

```
    if h ∈ (list_p2p ∪ list_irc) then
      BlackList ← BlackList ∪ h
    else
      if h ∈ list_http then
        GreyList_http ← GreyList_http ∪ h
      else
        GreyList_ml ← Greylist_ml ∪ h
      end if
    end if
  end if
end for
```

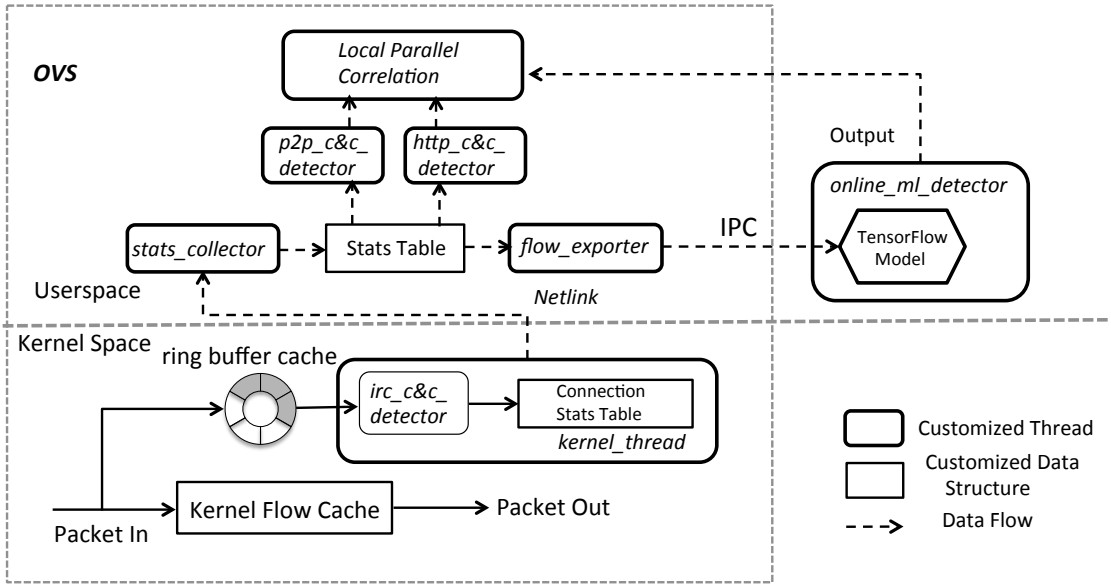

Figure 3.3: BotSifter system implementation

Local parallel correlation

While C&C based detection detects bots via monitoring C&C communication, ML based detection detects the bots via monitoring bot activity traffic. BotSifter introduces a local detection correlation module that combines the results from these two types of detection modules. The consistent detection by both C&C modules and ML based module is a strong indication that a host has been compromised and will be placed on a blacklist. On the other hand, a single positive identification may not be conclusive. For example, the P2P C&C detection module detects that a host may potentially be a bot. However, the bot may still be dormant and have not launched any attacks yet. As another example, if a host is identified by the online ML module as an attacker, it is not clear if the identified host is a bot with a customized C&C protocol that is not recognized by our C&C detection modules, or an ordinary attacker without any C&C channels. In any case, such bots are placed on a so-called local grey list and will be under persistent monitoring/scrutiny. The blacklists and grey lists are sent to the central controller for network-wide correlation and identification.

Algorithm 5.

1: **global** $blacklist, greylist_{http}, greylist_{ml}$ ▷ Global blacklist and greylist
2: **procedure** NETWORKWIDECORRELATION($BlackList$, $GreyList_{http}$, $GreyList_{ml}$)
3: **foreach** $h \in BlackList$ **do**
4: UPDATEBLACKLIST($blacklist, h$)
5: INSTALLRULE(h) ▷ Block traffic from host h
6: **end for**
7: **foreach** $h \in GreyList_{http}$ **do**
8: $g =$ FINDHTTPGROUP(h) ▷ Find the group based on destination IP address
9: **if** $\exists h' \in g$ **and** ISHTTPBOT($blacklist, h'$) **then**
10: UPDATEBLACKLIST($blacklist, h$)
11: **else**
12: **if** $h \in greylist_{http}$ **and** GETELAPSEDTIME(h) $> timeout$ **then**
13: $greylist_{http} \leftarrow greylist_{http} - h$

```
14:       else
15:         if h ∉ greylist_http then
16:           UpdateHttpGreylist(greylist_http, h)
17:         end if
18:       end if
19:     end if
20:   end for
21:   foreach h ∈ GreyList_ml do
22:     if h ∈ greylist_http then
23:       UpdateBlacklist(blacklist, h)
24:       InstallRule(h)
25:     else
26:       UpdateMLGreylist(greylist_ml, h)
27:       ContinueMonitor(h)
28:     end if
29:   end for
30: end procedure
```

3.3.4 Network-wide Correlation and Mitigation

While local detection is beneficial for the system scalability, the lack of global view can deter the bot detection. Hence BotSifter introduces a centralized correlation module that examines the blacklists and greylists received from the edge servers, and performs network-wide correlation to detect bots. In addition, a mitigation module is implemented to mitigate bots' damages. For each bot on the blacklist, the controller installs flow rules into the OVS to block C&C traffic and/or attacking traffic.

For hosts on a local C&C greylist, the controller performs network-wide grouping analysis to determine if it is a bot. Each group consists of the hosts on blacklists or C&C greylists communicating with the same destination host. If any host in the same group has already

been positively identified as a bot, the greylist hosts in the same group are marked as a bot and will be placed on the blacklist. Otherwise, the controller places the host on the global C&C greylist, and continues to monitor the host until a timeout occurs. For hosts on the local ML greylist, the controller looks up the global C&C greylist to check if there is a match. If so, this host is included in the global blacklist. Otherwise, it is placed onto the global ML greylist and continues to be monitored for future signs of C&C communication.

3.4 BotSifter Implementation

We implement a prototype BotSifter following the design as laid out in the Section 3.3. Fig. 3.3 highlights the major parts implemented for BotSifter. More implementation details are described as below.

3.4.1 Local OVS Traffic Collection Implementation

The local traffic monitoring function is implemented in a kernel thread called *kernel_collector*. We also modify the kernel forwarding thread so that the packet headers are pushed into the ring buffer cache upon arrival. The *kernel_collector* thread takes the packet headers off the ring buffer, and process them to generate the required stats that are stored in hash tables. The hash tables with connection stats are periodically pulled by the user-level stats collection thread *stats_collector*, as shown in Fig. 3.3.

A more involved task is to detect P2P connections required by the P2P C&C detection module. To identify a P2P connection, the kernel thread intercepts the DNS requests, parses them, and records the hosts who have conducted look-up for a specific IP address. For each new connection between any two hosts *src_ip* and *dst_ip*, we check whether the *src_ip* host has conducted a DNS lookup for the destination *dst_ip*. If yes, it is a normal connection. If not, this new connection is marked as a P2P connection, which will be further examined by the P2P C&C detection module.

3.4.2 Parallel C&C and Bot Activity Detection Module Implementation

Three threads, *irc_c&c_detector*, *p2p_c&c_detector* and *http_c&c_detector* are implemented to realize the IRC C&C detection, P2P C&C detection, and HTTP C&C detection, respectively. Thread *irc_c&c_detector* runs in the kernel, as discussed in the Section 3.3, while *p2p_c&c_detector* and *http_c&c_detector* run in the user space, as in Fig. 3.3.

To identify IRC connections, *irc_c&c_detector* performs lightweight payload inspection against connections associated with standard IRC ports (6660-6669). The first few packets of an IRC connection typically contain certain keywords, such as "JOIN","USER", "NICK" and "PRIVMSG". To further identify IRC C&C channels, *irc_c&c_detector* examines the IRC messages by searching for attack relevant commands (e.g., "scan", "flood"). If such commands are recognized, the *irc_flag* for the connections will be set and exported to the userspace.

As mentioned in Section 3.3, the differentiation of P2P C&C traffic and normal P2P traffic relies on self-correlation behavior of peer sets. For the self-correlation, we conduct overlap analysis over multiple time windows of each P2P connection. Time window is a fixed length period of time during the connection. For each time window, we calculate the per-host peer sets in the current and N subsequent time windows. Then, we calculate the number of re-appearing peers in both the current time window and the $i^{th}(i \leq N)$ time window. We use the average of the N overlap values to represent how the peer sets evolve over time. For runtime botnet detection, we calculate the moving average overlaps to differentiate the P2P bots from the normal hosts.

Thread *online_ml_detector* implements a NN based bot detection module. The NN model is implemented using TensorFlow 1.8.0 [65], an open source NN library. Since the thread runs outside of the OVS, a user-space thread inside OVS, called *flow_exporter*, is created to allow *online_ml_detector* to retrieve the connection feature vectors collected by OVS using Linux IPC calls, as depicted in Fig. 3.3.

Table 3.1 summarizes the features used in our NN model. Notably, different from previous flow based machine learning models, *src/dst* IP addresses are excluded from our

Table 3.1: Description of connection features.

Feature	Description
Source Port	Source port.
Destination Port	Destination port.
Protocol	IP protocol.
Forward packet count	Total number of packets in the forward direction.
Backward packet count	Total number of packets in the backward direction.
Forward byte count	Total number of bytes in the forward direction.
Backward byte count	Total number of bytes in the backward direction.
Duration	Connection duration.
Forward packet rate	Packet rate in the forward direction.
Backward packet rate	Packet rate in the backward direction.
Connection State	whether the connection is successfully established.
Forward inter-arrival time	Packet inter-arrival time in the forward direction.
Backward inter-arrival time	Packet inter-arrival time in the backward direction.
Forward average packet size	Average packet size in the forward direction.
Backward average packet size	Average packet size in the backward direction.
Forward byte rate	Byte rate in the forward direction.
Backward byte rate	Byte rate in the backward direction.

feature set. We believe the src/dst IP address are unique to specific data centers and thus shall be excluded. More importantly, the exclusion can avoid over-fitting the NN model. Other features are chosen so that they can help distinguish botnet activity traffic from legitimate traffic. We experimented with different features and incorporate the most discriminative ones in our feature set. For example, forwardbackward average packet size of normal flows tend to be larger than botnet traffic since they contain realistic payloads. Meanwhile, we introduce a novel feature, i.e., *conn_stat* to represent whether the connection has been successfully established. Since bots usually maintain a large number of half open connections, this feature is effective in identifying malicious botnet traffic such as direct DoS flooding attack and the amplification attack traffic.

The *online_ml_detector* thread fetches NN configuration parameters from the central controller, and initiates the NN model. The *flow-exporter* thread in the OVS continuously exports connection feature vectors to the NN model, as shown in Fig. 3.3. To facilitate the communication between *flow_exporter* and *online_ml_detector*, a shared memory pool is created. Linux semaphores are used to synchronize the access to the shared memory.

3.4.3 Network-wide Correlation and Mitigation Implementation

Bot detection applications are programmed at the central controller to perform network-wide correlation analysis. Once a bot is detected, the controller installs customized flow rules into the corresponding OVS switch to prevent future attacks. To facilitate the communication between the threads at servers and controller, traditional OpenFlow protocol is extended to facilitate the collection of detection results from edge servers.

3.5 Evaluation

Our testbed consists of three Lenovo ThinkServer machines running Ubuntu 14.04. Each machine is equipped with Intel Xeon 4-Core 3.20GHz CPU and 4GB RAM. One machine is installed with Open vSwitch 2.3.90. Another machine runs the Ryu SDN controller. A third machine serves as both packet generator and data sink, which is connected to the OVS machine via two 10Gbps Ethernet cables. Packet traces are replayed using TCPReplay at the original speed.

3.5.1 Datasets

Table 3.2: Description of botnet datasets for training and testing.

Datasets	***Underlying C&C Protocol***			
	HTTP	***P2P***	***IRC***	***Custom***
Training Dataset	Neris, Virut	Storm, Waledac	Rbot	-
Testing Dataset	Neris, Virut, Sogou	Storm, Waledac, NSIS	Rbot	Menti

For our experiments, we are able to obtain several third-party traces, including a variety of botnet traces composed of different botnet types and normal network traces. CTU-13 dataset [66] contains traffic of botnet samples which were captured in the CTU University, Czech Republic, in 2011. In each botnet sample, bots use specific protocols for C&C

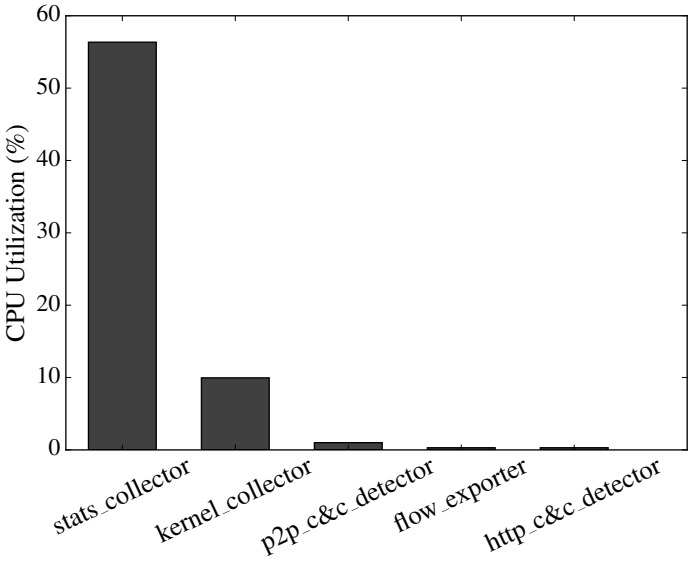

Figure 3.4: CPU utilization of detection related threads.

communication and perform diverse malicious tasks, including SPAM, port scanning, DDoS, click fraud, etc. Due to the diversity of botnet samples, this dataset is appropriate for evaluating the performance of our framework. For the normal network traces, we use the UNB ISCX IDS 2012 dataset [67] containing normal traces with full packet payloads. This dataset captures typical user daily activities (e.g., HTTP, DNS, SSH, FTP) at UNB university and was made public for scientific research.

As previously discussed, our runtime bot activity detection relies on a pre-trained model. Table 3.2 summarizes the composition of the botnets in the training and testing datasets. For the training dataset, we merged several traces, including P2P botnets (Storm, Waledac), IRC Botnets (Rbot) and HTTP botnets (Neris, Virut). For runtime detection, we include not only the same types of botnets as the training dataset, but also novel botnet variants, such as NSIS (P2P C&C), Menti (Proprietary C&C) and Sogou (HTTP C&C). Among them, Menti uses a custom protocol for C&C communication. By evaluating how BotSifter performs when facing novel botnet variants, this experimental strategy aims to demonstrate

the effectiveness of our design in real-world bot detection.

3.5.2 Impact on Normal Applications

Since the botnet detection accuracy of our system strongly relies on the accurate timing information of the packet sequences, the botnet traces are replayed at the original speed. To evaluate the overall system performance, we perform a stress test using a CAIDA trace and replay the trace towards our monitoring system at the highest achievable rate.

Considering that no modification is made for native OVS threads (e.g., handlers and revalidators), the CPU usage of these threads is not shown here. We only report the CPU utilizations of the customized threads in Fig. 3.4, which shows the peak CPU usage of each detection-related thread under various detection tasks.

We can see that the *stats_collector* in the userspace incurs the highest CPU utilization while the threads regarding C&C detection and flow exportation to the online machine learning detector only introduce negligible overhead. Since *stats_collector* mainly manages the collection of connection stats from the kernel space through Netlink and the maintenance of the connection hash table in the userspace, we infer that its CPU utilization is mainly attributed to the Netlink communications.

In realistic scenarios, these overheads are acceptable as OVS typically resides in commodity servers with abundant CPU resources. The detector threads could be pinned to different CPUs to avoid interfering with CPUs dedicated to the forwarding functions in OVS.

To estimate the network overhead, we use DPDK based packet generator MoonGen to generate high speed traffic and measure the maximally achievable throughput such that no packet loss occurs on the forwarding path. The experiment is repeated 10 times. Compared to the throughput of the native OVS (1.44Mpps), BotSifter could achieve 1.15Mpps throughput. This overhead is acceptable and further analysis shows this overhead is mainly incurred by the *memcpy* operations on the ring buffer cache.

3.5.3 Detection Performance

Table 3.3: Detection results for 8 botnet variants and normal trace.

Botnet Variant	*C&C Protocol*	*#Bots or #Hosts*	*Duration*	*P2P Overlap*	*HTTP Periodicity*	*IRC C&C*	*C&C Accuracy*	*ML Detection Accuracy*	*Local Parallel Accuracy*
Neris	*HTTP*	1	4.8h	-	*Strong*	-	1/1	1/1(99.2%)	1/1
Virut	*HTTP*	1	16.36h	-	*Strong*	-	1/1	1/1(99.7%)	1/1
Sogou	*HTTP*	1	0.38h	-	*Weak*	-	1/1	1/1(95.5%)	1/1
Storm	*P2P*	13	3.1h	*Yes*	-	-	13/13	13/13(90.1%)	13/13
Waledac	*P2P*	3	3.45h	*Yes*	-	-	3/3	0/3(50.5%)	3/3
NSIS	*P2P*	3	1.21h	-	-	-	0/3	1/1(93.7%)	3/3
Rbot	*IRC*	1	5h	-	-	Yes	1/1	1/1(99%)	1/1
Menti	*Custom*	1	2.18h	-	-	-	0/1	1/1(99.4%)	1/1
Normal	*n/a*	36	10h	-	*Yes(4/36)*	-	4/36 on greylist	FPR:0/36(See Fig. 3.6)	FPR: 0/36

Evaluation metrics

To evaluate the performance of Botsifter, we use three metrics. *C&C Accuracy* is calculated as the ratio between the number of hosts which have triggered alarms of C&C detectors and the total number of hosts. *ML Accuracy* represents the detection rate of the bots based on suspicious ratios predicted by the runtime NN model. Since our final detection result relies on a parallel correlation between C&C detection and runtime ML detection, we define a novel metric *Local Parallel Accuracy* to represent the overall accuracy.

As discussed in Section 3.3, the hosts triggering alarms from both *http_c&c_detector* and *irc_c&c_detector* will be put onto a greylist instead of blacklist since they need further network-wide correlation. In contrast, alarms from *p2p_c&c_detector* indicate the associated hosts must be bots and thus they are included in a blacklist. Therefore, we claim that if the host appears on either the C&C blacklist or the ML greylist (e.g., is an attacker, but needs further monitoring to find out C&C channels), it is regarded as a true positive , which also implies that it has been successfully detected by BotSifter. This design principle further demonstrates the robustness of BotSifter compared to methods solely based on single stage detection (either C&C or machine learning).

Detection Accuracy

The measurement results are shown in the C&C accuracy column in Table 3.3. Note that, in ML Accuracy, the percentage values in the brackets represent the suspicious ratios for the bots in the current trace. We can see that, parallel detection scheme in BotSifter successfully detects all bots with zero false positives, although some bots trigger alerts from only one stage (either C&C or NN model), such as NSIS and Menti. In the normal trace, there are no false positives due to the following two reasons. First, the hosts are included in HTTP greylist instead of the blacklist, due to the periodicity patterns of software updates. Besides, the suspicious ratios of all hosts are far below the threshold. In a nutshell, BotSifter is able to detect all bots which may get missed by any single stage of detection. Since local detection accomplishes the tasks, no computation is needed from the central controller, which demonstrates that BotSifter is a distributed scalable solution. In the following, we analyze the results in more detail.

HTTP C&C detection. Based on our empirical experiments, we find that the C&C connections of certain HTTP botnets may not exhibit periodicity patterns as strong as other botnets. This may be due to unknown factors such as network delay and congestion which introduce noises to the connection timing. We use *strong* and *weak* to represent whether strong periodicity is observed in each botnet trace. Among the used HTTP botnet samples, Neris and Virut exhibit strong periodicity since the bots connect to the HTTP C&C server at regular intervals with negligible timing variations. Since periodicity patterns are observed for each 3-tuple *(src_ip, dst_ip, dst_port)*, the suspicious client and server associated with the 3-tuple could be accurately pinpointed and placed onto a greylist for further examination. Aside from this, we also observed that certain botnets contain more than one HTTP C&C channels, some of which exhibit no periodical pattern. These C&C communications are not typical and cannot represent the botnet C&C behaviors. For those C&C channels exhibiting periodical connection patterns, our online *http_c&c_detector* can accurately identify the C&C channels.

Table 3.4: Description of P2P traces.

Trace	***Normal***			***Botnet***	
	Emule	***FrostWire***	***uTorrent***	***Storm***	***Waledac***
Number of Packets	3.6M	1.35M	6.5M	6M	2.5M
Duration	$3.25h$	$10h$	$8h$	$3.16h$	$8.23h$

During the test for the normal trace, the *http_c&c_detector* triggers an alert unexpectedly, which implies that periodic http connections are observed between two normal hosts. Further examination reveals that those hosts are running legitimate applications with periodic HTTP connections.

P2P C&C detection. The detection results for the three P2P based botnets are also quite promising. Storm and Waledac both show strong self-overlap in their C&C communications. One exceptional case is NSIS, for which no P2P overlap pattern is observed. This is reasonable as the trace is reported to be incomplete with only one P2P C&C channel, which is insufficient for P2P C&C analysis. In the following, we demonstrate that our detection scheme allows to accurately distinguish P2P botnet C&C traffic from normal P2P traffic.

As previously discussed, the key characteristic differentiating botnet P2P C&C traffic from normal P2P traffic is that the peer sets of a P2P bot have large overlaps across consecutive time windows; while the overlaps for a normal P2P host are noticeably smaller. To verify this, we use several third-party P2P traces [68] of both normal P2P applications and P2P botnets. Table 3.4 gives a brief summary of our P2P traces. The Storm and Waldec P2P botnet contain 13 and 3 P2P bots respectively. Our experiments show that the online *p2p_c&c_detector* could successfully detect all bots.

In our experiment, the average overlap values are calculated across 5 consecutive time windows with respect to the current time window. The length of the time window is set to be 10 minutes. We have tested with various window parameters and acquired similar detection results. However, choosing a relatively small window and short window sequence results

in more prompt detection. Otherwise, it causes longer detection delay if more subsequent time windows are used to calculate the average overlaps. To demonstrate the effectiveness of the self-correlation scheme, the moving average overlaps for 10 initial time windows for each P2P host in the trace are shown in Figure 3.5. For each trace, the result for only one P2P host is depicted in the figure. Other hosts in the same trace all demonstrate similar patterns as the one reported here.

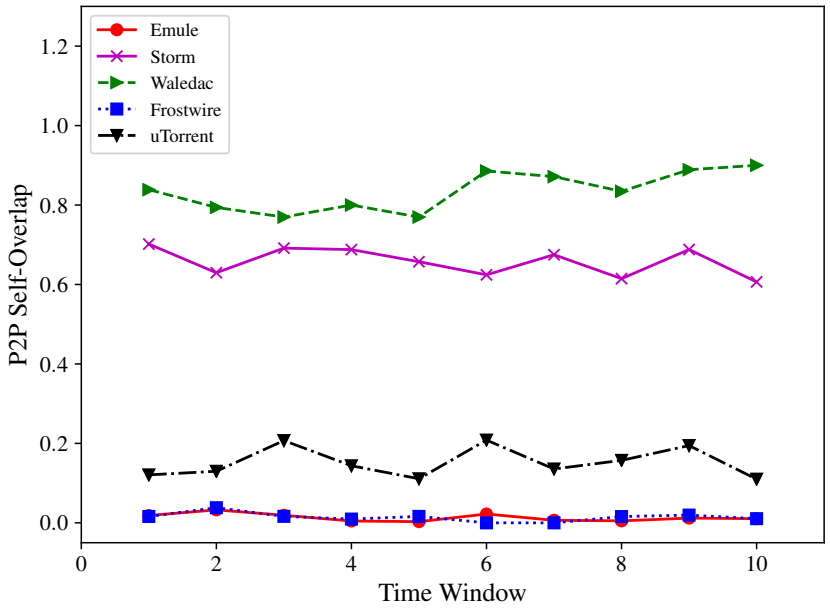

Figure 3.5: Peer overlaps for botnet/normal P2P applications.

From Figure 3.5, we can find that there is significant difference between the peer overlap for P2P bots and normal P2P hosts. In real world detection, a straightforward threshold based measure (e.g., 0.6) would suffice to distinguish between P2P bots or normal hosts. Based on the reported overlaps, our *p2p_c&c_detector* manages to identify all P2P bots in both traces. This demonstrates the effectiveness of the self-correlation scheme for detecting P2P C&C channels.

IRC C&C detection. In the Rbot botnet, the bot communicates with the C&C

server through an established IRC channel. Via this channel the server commands the bot to perform port scanning against certain IPs in the network and the bot continuously reports scanning results back to the C&C server. Our *irc_c&c_detector* can accurately detect the IRC C&C channel using keyword matching.

Different from other botnets, Menti adopts a custom protocol for an unencrypted C&C channel and performs port scans. Since its C&C communication does not rely on the three well-known C&C protocols, the *c&c_detector* discovers no suspicious C&C pattern. However, our experiment shows that *online_ml_detector* raises an alert since suspicious activities are discovered in the trace, which will be further explained in the following analysis.

Runtime ML based bot detection. From Table 3.3, we can see that the suspicious ratios for the majority bots fall between 95% to 99%. For the remaining bots, the ratios still exceed 90%. The result for Waledac is relatively low since the majority of its traffic is C&C related. Since the goal of ML detection module is to detect suspicious bots instead of individual connections, the *online_ml_detector* raises alerts based on the suspicious ratio for each individual host. As defined in Section 3.3, it represents the ratio of the number of suspicious connections with respect to the total number of connections for each host. If the ratio exceeds a pre-specified threshold, the host will be included in a greylist. Obviously, the choice of this threshold has a direct impact on the detection accuracy. A higher threshold leads to more false negatives while a lower threshold incurs more false positives. With a threshold of 10%, in the normal trace, 4 out of 36 are false positives. Instead, with a higher threshold, for example, 85%, all bots are accurately identified, with zero false positives. By excluding those hosts with a limited number of connections, the ratios of all other hosts are shown in Figure 3.6.

For novel botnets, such as Menti, no C&C patterns are discovered since it uses a custom C&C protocol. However, as shown in Table 3.3 the *online_ml_detector* reports high suspicious ratio for the bot in this trace and raises alarms for further correlation. This parallel design is extremely effective in detecting bots in real-world, since it increases the difficulty of novel botnet variants evading both stages of detection.

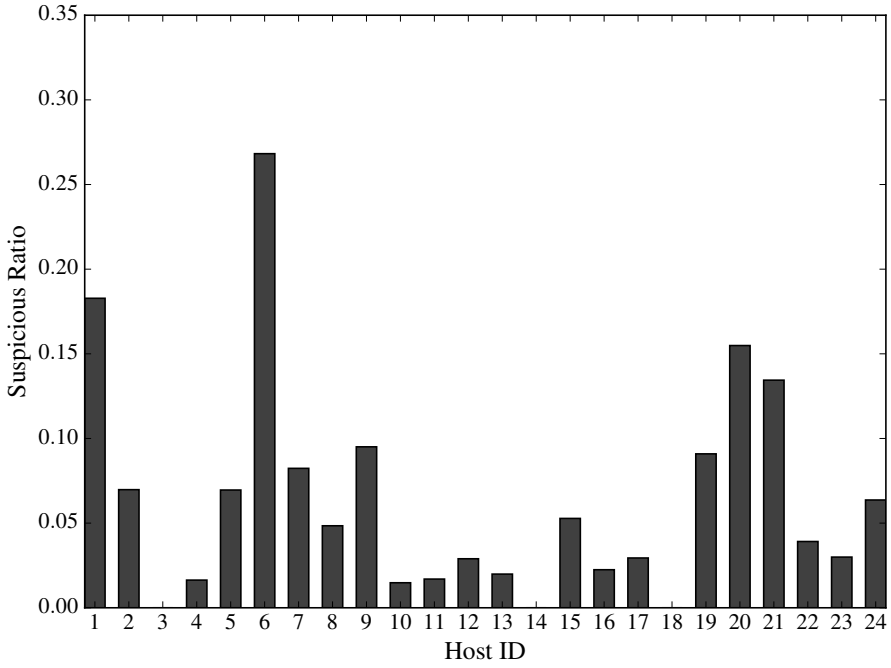

Figure 3.6: Suspicious ratios for hosts in the normal trace.

Performance comparisons.

We compare our approach to previous works by evaluating them using the same datasets in our experiment. Since our detection relies on parallel correlation of C&C detection and bot activity detection, the comparison is two-fold. First, we compare our ML detection accuracy to a recent ML based approach [56]. Different from ours, their ML model is based on per-flow feature vectors and only achieves an overall accuracy of 75% on a testing dataset containing multiple novel botnet variants not embraced in the training dataset. By contrast, in our evaluation for Menti (a variant with a custom C&C protocol), the ML model reports a fairly high accuracy ($\sim$99.4%). Indeed our detection method achieves satisfactory accuracy for a majority of the test traces.

Moreover, another similar work [54] focuses on the detection of stealthy P2P botnets based on cross-bot overlap analysis, which shows that the P2P bots could be identified

based on cross-bot correlation. Our experiments also evaluate the detection accuracy on the same P2P botnets. As shown in Table 3.3, all P2P bots are detected without any false negatives, which indicates that Botsifter can achieve comparable detection performance by solely using single-bot patterns. However, our approach is more robust in the scenario of a small number of bots.

3.6 Chapter Summary

The cloud-based bots pose an imperative threat to the applications and services running on various cloud platforms, yet highly accurate and scalable detection solutions are not available. In this study, we have designed and implemented BotSifter, a SDN based scalable and accurate runtime bot detection framework in data centers. BotSifter utilizes a centralized learning and distributed detection model that is effectively supported by SDN. Furthermore, BotSifter adopts parallel detection of both the botnet C&C communication patterns and the machine learning based attack activities. The evaluations show the effectiveness of BotSifter based on real-world traces.

Chapter 4: EZPath: Expediting Container Network Traffic via Programmable Switches in Data Centers

4.1 Introduction

In recent years, containerization has been increasingly adopted to deploy large-scale distributed applications (e.g., content providers [69, 70], eCommerce [71, 72], and in-memory key-value stores [73]) in clouds, such as AWS Lambda [74], Google Compute Platform [75], and Microsoft Azure [76]. However, container networking has been suffering in such a multi-tenant environment, particular with the increasing deployment scale due to the following reasons. First, the sharing nature of multi-tenant cloud networks requires tenant isolation and quality of service (QoS) through the enforcement of thousands of control plane policies (e.g., access control) and data plane policies (e.g., tunneling, QoS and rate limiting), resulting in significant host computing resource consumption. The sheer density of the container deployment and its short-livedness further exacerbate the problem [77, 78]. Second, the container networking should be able to provide portability and flexibility for container placement and migration, obviating the need for the application developers to coordinate the assignment of port and IP addresses.

Existing container orchestration solutions mostly employ virtual overlay networks to achieve portability and flexibility. Essentially, overlays employ various tunneling technologies, e.g., VXLAN, GRE, to implement virtual networking among the containers owned by a single tenant, providing tenant isolation and network virtualization. One example of such software entities is Open vSwitch [79], which is widely deployed in data center servers to enable network virtualization on end hosts [80–83]. However, as shown in previous works [84–86], implementing the tunneling and other network virtualization functionalities in the software switch causes significant networking performance degradation.

In practice, the poor performance of overlay networking solutions has remained to be a pressing issue ever since virtualization came into play. In VM-based virtualization environments, Single Root I/O Virtualization (SR-IOV) has proved to be an effective technique to improve the networking performance [84]. Nonetheless, the number of VMs that can be accommodated on a commodity server is far less than that of containers in contemporary containerized data centers. More recently, some other designs [77, 85, 87] have been proposed to improve container network performance, while aiming to preserve its portability. However, they are difficult to deploy due to the requirement of customized software and extra maintenance [85], or specialized hardware support [77, 84].

In this work, we propose to develop an efficient and application-transparent framework, called *EZPath*, to expedite the container network traffic, by leveraging the programmable data planes of the prevalent Software Defined Networking (SDN) switches in data centers. We achieve this goal by offloading heavy-weight network traffic to in-network programmable switches to relieve the system resource pressure on the servers. The performance of the containerized applications is significantly improved in benefit of the programmability and line rate processing speed of modern switches (e.g., P4).

Nonetheless, due to the specific requirements and constraints in the containerized environment, migrating network functions and traffic to the programmable hardware poses several challenges. First, due to the sheer scale and density of containers in deployment, the amount of memory for accommodating the metadata (i.e., the tunnelling mappings) required by performing the container tunnelling functions can be substantial. Therefore, simply offloading the tunnelling operations for all container traffic is not practical, given the resource constraints imposed by programmable switch ASICs.

Second, containers usually have much shorter lifespans than virtual machines in many application scenarios, such as microservice deployment and serverless computing. Blindly offloading all traffic may cause constant update of offloading selections and tunnel mappings, which could potentially degrade the overall network performance. Therefore, we need an

optimal strategy that strikes the balance between network performance and resource consumption.

To address these challenges, we develop *EZPath*, a holistic framework that optimizes the performance of container network virtualization through hardware-assisted acceleration. *EZPath* is featured with a software-hardware codesign that incorporates the control plane software and the dataplane hardware. The control plane determines the offloading strategies in an adaptive manner based on the monitoring results from the data plane. The offloading strategy is then translated into P4 instructions executed in the programmable switches. By seamlessly integrating heavy flow monitoring with adaptive offloading, *EZPath* can flexibly migrate the network virtualization functionalities of the most performance-critical flows to the ToR (Top-of-Rack) switches.

To evaluate *EZPath*, we conduct extensive experiments with some typical container workloads, e.g., containerized key-value stores, web servers, and message queuing applications. While details are to be presented later in the chapter, the highlights of *EZPath* include:

- We quantitatively evaluate the overhead of the overlay approach for container networking, and show such overhead contributes significantly to the network bottleneck.
- Taking a software-hardware co-design approach, we design and implement *EZPath* that can adaptively monitor and offload heavy network traffic by leveraging the programmable data planes of modern SDN switches.
- *EZPath* is application transparent, preserving compatibility with legacy containerized applications. It does not require changes in host kernel, making it compatible with all existing monitoring tools. The evaluation results show that *EZPath* can expedite container traffic significantly, e.g., with a 35% improvement on throughput and a 42% tail latency reduction for Memcached.

The reminder of this chapter is organized as follows. We present the background of container networking in section 4.2. Via experiments, we reveals the breakdowns of the

network performance in section 4.3. The design and implementation of *EZPath* are presented in section 4.4 and section 4.5, respectively. *EZPath* is evaluated in section 4.6. We discuss related works in section 4.7 and make concluding remarks in section 4.8.

4.2 Background

To understand the packet processing overhead in the network stack, in this section, we first present the generic background on how Linux networking stack processes packets. We then introduce the principles of the container overlay networking and analyze the corresponding packet process path. Finally, we quantify the overhead of container overlay network and provide a detailed root cause analysis.

4.2.1 Linux Packet Processing

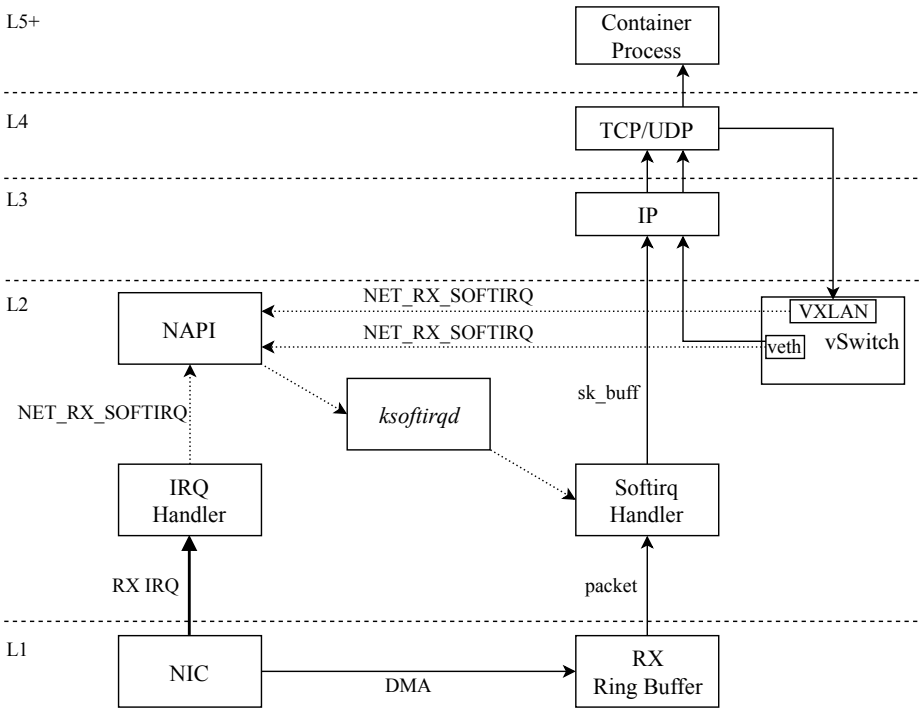

Figure 4.1: Packet ingress path in container overlay network.

As shown in Figure 4.1, Linux kernel packet processing generally consists of two phases: a top half for hardware interrupts and a bottom half for software interrupts. The top half quickly services the hardware IRQ and schedules the bottom half to service the software interrupts. Most of the packet processing actions are deferred into the bottom half. When a packet is received by the NIC from the network, the NIC uses DMA to copy the data to the ring buffer in kernel memory and raises an IRQ to the CPU. The top half executes the interrupt handler registered by the device driver to serve the IRQ and raise the NET_RX_SOFTIRQ softirq. A per-CPU *ksoftirqd* kernel thread is responsible for the bottom half processing, which runs the corresponding softirq handler to pass the data frame through the kernel network protocol stack and finally to the user space process. Thus, the majority of the packet processing is executed in the softirq context. In Section 4.3, we will analyze and break down the packet's processing overhead in a container's overlay network by following the packet processing procedure in the Linux kernel.

4.2.2 Container Overlay Networking

The most popular way to virtualize container networks is through overlay networks, which provide Layer 2 connectivity among distributed multi-host containers. Container overlay networks typically employ a tunneling technique (e.g., VXLAN) to transport the overlay container frames through the underlay network. For instance, VXLAN performs a MAC-in-UDP encapsulation that encapsulates container Layer 2 frames inside an underlay IP/UDP header. The overlay consists of stateless VXLAN tunnels among the participating hosts, while the host IP/UDP header provides the connectivity between hosts on the underlay network. Since each container overlay network has its own IP address space and network configuration, the overlay is independent of the underlay topology, thus making the applications portable.

To provision an overlay network, a virtual switch (e.g., Open vSwitch [79]) in the kernel is leveraged to perform tunneling-related packet transformations. Specifically, the container is attached to the virtual switch through a pair of virtual Ethernet (veth) devices, and the

virtual switch bridges the containers on the same host. To provide external connectivity, the virtual switch creates a VXLAN interface for tunneling packets through encapsulation and decapsulation. Compared to the native host network, a container overlay network involves more complex packet processing and prolonged data path. The packet has to traverse the overlay/host kernel stack in different network namespaces and additional devices such as the virtual switch, the tunneling interface, and the veth interfaces. Even worse, the extra traversal of these virtual devices incurs a high rate of softirqs. Given that software interrupts run in the process context, this inevitably leads to high packet processing overhead and interference with the performance of userspace processes [88].

4.3 Motivation

In this section, we first investigate the overhead of the overlay networking approach, and discuss other alternative options for improvement, which motivates the design of *EZPath*.

4.3.1 Overhead of Overlay Networking

To understand the performance issues of virtual switch based network virtualization, we perform an overhead breakdown analysis of a popular container network solution: Docker Overlay [89]. Docker overlay utilizes a VXLAN data plane that decouples the container network from the physical underlay network. A virtual Linux bridge is created per overlay along with its associated VXLAN interfaces. As depicted in Figure 4.2, our testbed consists of two KVM virtual machines (VMs) on a single physical host. The VMs are used to simulate two physical hosts. Each VM is configured with 2 vCPUs, 2 GB memory and a virtio NIC. The overhead remains the same in a physical machine environment as long as the underlay host network is not the bottleneck. We create a Docker container inside each VM, which runs network performance benchmarks Netperf [90] and iperf3 [91]. We then deploy a Swarm mode overlay network to connect the containers. To quantify the overhead, we compare the performance of the container overlay to the host mode network, where the container network stack is not isolated from the host stack and the host IP is directly

allocated to the container. We measure the TCP/UDP throughput when sending data as fast as possible from the client to the server.

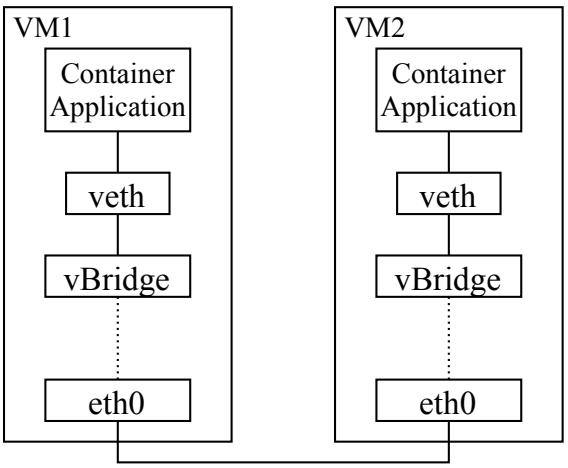

Figure 4.2: Experiment setup

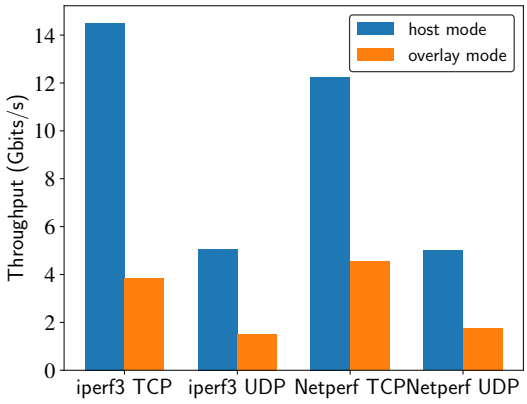

Figure 4.3: Throughput overhead of container overlay

Figure 4.3 shows the results. The performance of iperf3 is shown on the left half while those of NetPerf is shown on the right half. Each experiment is repeated five times and the average results are reported here. As shown in Figure 4.3, the TCP throughput of the container overlay network is only 26.4% and 36.2%, respectively, when compared to the native host networking for iperf3 and Netperf. The trend of overhead for the UDP throughput is similar. UDP does not have congestion control, which results in lower throughput than TCP in iperf.

Table 4.1: Packet Latency

Networking Mode	Latency (μs)
Host mode	26.19
Overlay mode	38.48

We further use NPtcp [92] to measure the latency of TCP packets. As shown in Table 4.3, the virtual switch based tunneling incurs 46.7% more latency on average (26.19 μs and 38.48 μs for the host and overlay modes, respectively).

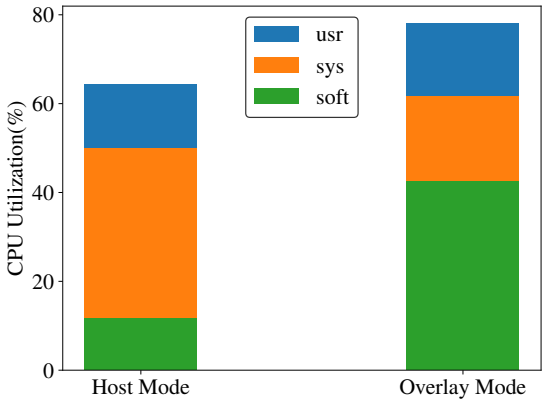

Figure 4.4: CPU utilization: host mode vs. overlay mode

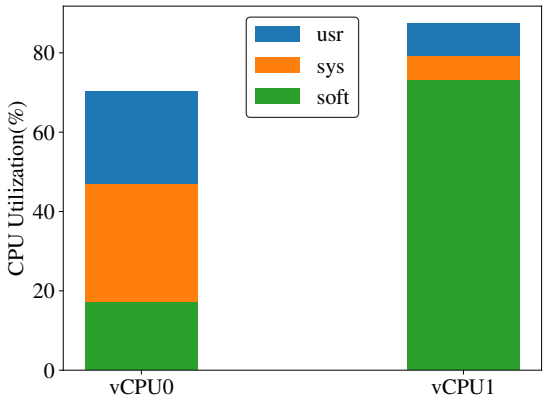

Figure 4.5: Per-CPU utilization in overlay mode

To understand the bottleneck of the tunneling handling, we use mpstat [93] to measure the CPU utilization of the system when stress testing the overlay network using iperf3. Specifically, we run iperf3 to generate traffic from the client container for 100 seconds. Figure 4.4 shows a breakdown of the average CPU utilization on the iperf3 server. For the host mode networking, 38.15% of CPU utilization is due to executing the kernel code (sys); while only 11.7% of CPU is spent on servicing softirqs (soft). For the container overlay networking, the softirq processing accounts for 42.48% of CPU utilization and the kernel code execution takes 19.14%.

To quantify where CPU cycles are spent among the software interrupts, we use ebpf [94] to collect the timing statistics of the softirq events. Specifically, it works by probing the softirq tracepoints softirq_entry and softirq_exit, which are called immediately before the softirq handler and after the handler returns. We use iperf to transmit 100GB of data through TCP in this measurement. Table 4.2 summarizes the CPU cycles (in μs) spent

on each type of softirq on both the iperf client and the server. As shown in the table, we can see that over 99.8% of the softirq processing is devoted to the networking softirqs NET_RX_SOFTIRQ and NET_TX_SOFTIRQ.

Table 4.2: Softirq event processing time breakdown

Softirq	Total time (μs)	
	iperf client	iperf server
`tasklet`	35	16
`block`	23439	18911
`rcu`	103120	72321
`timer`	132485	117597
`net_tx`	30256505	560
`net_rx`	117351671	185016254

The experiment results show that the degradation of the overlay network performance is attributed to the execution of extra kernel code (for tunneling-related packet transformations) and the servicing of an increased number of softirqs. The overlay network entails the traversal of additional virtual network devices (i.e., the virtual bridge and the VXLAN interface), which leads to an explosive growth of softirqs. We further look into the per-CPU utilization for the overlay networking. As shown in Figure 4.5, the majority of the softirqs is served by the ksoftirqd thread on vCPU1. This concentration of softirqs is determined by the IRQ affinity configuration of the system, which pins a type of interrupts to a particular set of CPU cores. This further confirms our analysis.

4.3.2 Alternatives for Container Networking

To optimize the container network performance, we first discuss different implementation strategies, ranging from hardware based acceleration to pure software solution. Pros and

cons of different solutions are discussed in detail, which sheds light on the design considerations adopted by *EZPath*.

Hardware Accelerations.

Since overlay networking requires packets to traverse both the guest network stack and host network stack, it introduces significant performance overhead to containerized applications. One natural solution is to assign physical network devices to selected containers and grant them exclusive access. Macvlan [95] and SR-IOV [96] are two such solutions.

With Macvlan, virtual interfaces are created, configured with host routable IP address, and assigned into the container namespaces. In contrast, SR-IOV requires hardware support that simulates a single PCI NIC as multiple virtual functions (VF), each with its own MAC address and functions as a physical NIC from the view of containers. However, the number of VFs supported by the hardware is quite limited (e.g., 64) and does not keep up with container network scales. Furthermore, to build applications that span across multiple hosts, both technologies require configuration of routable IP addresses on the host network. This may be a feasible solution in VM-based virtualization environment, but it is not well suited for container networks. Different from VMs, containers are often short-lived and may be migrated in real time, making frequent network reconfiguration a nightmare. In addition, for intra-host communication where containers reside on the same physical host, SR-IOV imposes significantly higher overhead than Macvlan, since packets from one container must be sent through PCIe bus to the NIC before being forwarded to the other container [97].

Compared to above networking modes, the overlay networking aims to build a virtual logical L2 network over an existing L3 host network. This provides much greater flexibility for multi-host networking in terms of easier configuration and management. Production container applications often span multiple hosts or even across multiple Data Center (DC) clusters to guarantee service scalability and reliability. For example, in a real world multi-tier web service, multiple fleets of containers would be deployed for web servers and backend services, respectively. Both the web tier and backend service tier are highly replicated so

that it could elastically scale and also cope with infrastructure failures. However, it is quite challenging for the hardware-based solutions (i.e., Macvlan and SR-IOV) to configure and manage such large scale production networks, especially considering the volatile nature of container applications. For this purpose, popular container orchestration frameworks (e.g., Docker Overlay and Flannel for Kubernetes) or in-house built management solutions usually adopt overlay as their networking solution in real-world production management.

SmartNIC offloading. To cope with above concerns, hardware offloading is a viable solution with great potential to retain both the flexibilities of overlays and the performance of bare-metal. Intelligent or smart NICs have emerged recently to bridge the gap between the constantly increased network speed and the limited CPU processing power at the host machine. They are equipped with a variety of functional blocks (storage, security, networking, etc) to perform computation tasks on behalf of server CPUs, which not only accelerates network application performance, but also frees up CPU cycles for application workloads. Nonetheless, hardware offloading in virtualized setups with OVS (e.g., VxLAN, connection tracking) is realized through SR-IOV or *virtio* that are not suitable for container environments. The aforementioned scalability limitation of SR-IOV remains in the container environment. Further, SR-IOV necessitates the installations of dedicated NIC drivers within the containers, and the installation of SmartNIC cards at individual host machines incurs extra cost and management overhead. Finally, as in the context of VM migration, SR-IOV makes the container live migration infeasible.

Software Accelerations.

Slim [85] proposed a pure software-based optimization approach to improve the container overlay network performance. It relies on extra pieces of software, including a shim layer to intercept socket related system calls, and a userspace router that creates connections on behalf of containers and maps host namespace file descriptors to the container namespace. This effectively shortens the packet path and improves the container network performance. However, Slim is not transparent to containers, which is a major limitation. Application

binaries are required to dynamically linked to the shim layer and extra care needs to be taken in the deployment and maintenance of the software components. Moreover, after connections are established, Slim allows local containers to directly talk to the remote containers via the host namespace file descriptors, bypassing the container network interface and the virtual switch. As a consequence, it lacks support for conventional low-level network monitoring and debugging tools, such as *tcpdump*, as packets are not going through the virtual network interface and thus cannot be captured.

4.4 *EZPath* Design

To overcome the aforementioned limitations of current container networking designs, in this section, we propose the design of *EZPath*, to improve the performance of container networking by leveraging recent advances in programmable hardware.

At a high level, *EZPath* takes a software and hardware codesign approach. Specifically, it leverages the software for centralized controller while utilizing the programmable data planes in modern switches for traffic offloading. As an example, Figure 4.7 shows the change of ingress traffic path with *EZPath*, when compared to the default overlay approach 4.6. From this figure, we can clearly see that an overlay network is realized by creating multiple virtual network devices that are connected through OVS. These include a VXLAN port, and a *veth* pair with one end assigned to the container namespace while the other end attached to OVS. Overlay packets follow this prolonged processing path, incurring much more software interrupts and corresponding context switches. Comparatively, in figure 4.7, *EZPath* reduces the number of network devices that a packet has to traverse in the overlay. *EZPath* not only shortens the transmission path that a packet has to traverse, but also saves a lot of system processing overhead caused by explosive interrupt handling.

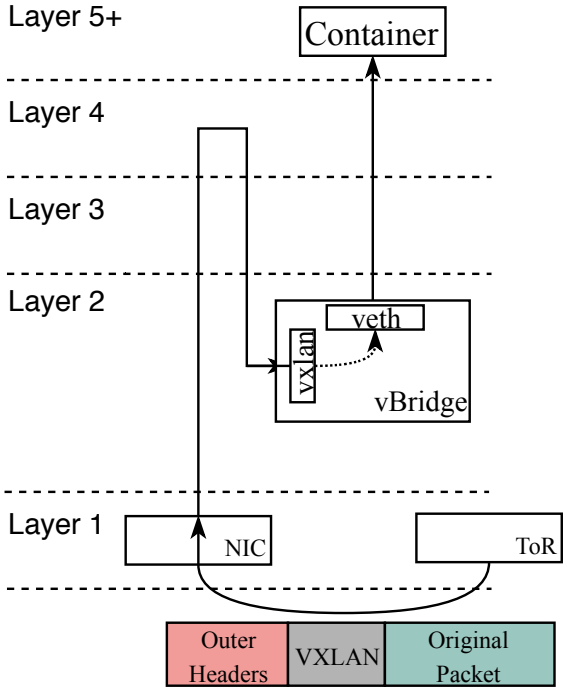

Figure 4.6: Overlay: vBridge tunneling

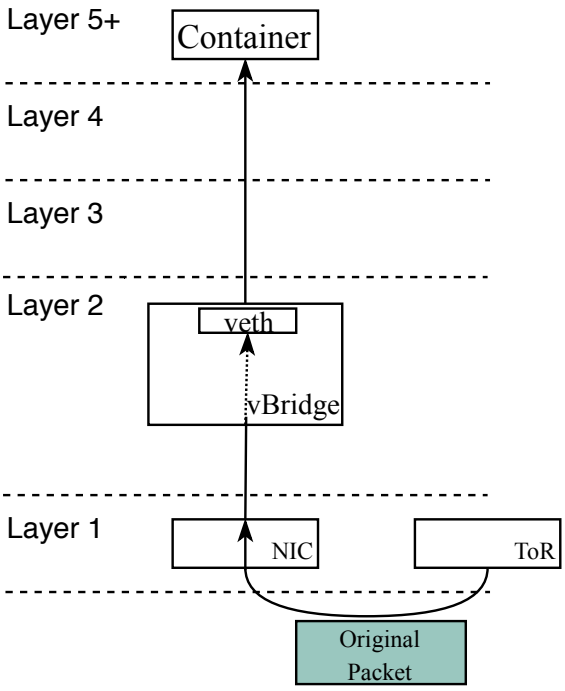

Figure 4.7: *EZPath*: ToR tunneling

Figure 4.8: Network stack view: Overlay vs. *EZPath* offloading. Ingress data path of (a) host virtual bridge based overlay tunneling and (b) *EZPath* offloading tunneling to ToR switch. Through *EZPath* offloading, the in-host packet data path is shortened, the number of traversed network devices is reduced, the kernel processing for softirq is saved.

4.4.1 Key Principles and Challenges

Resource Constraints While *EZPath* can offload all traffic through the hardware in an ideal situation and thus completely eliminate the network bottleneck in the container network, this is not feasible in practice, mostly due to the constraint of limited hardware resources. Despite the high-speed forwarding performance, switching hardware has highly constrained on-chip memory. For example, in a typical Tofino switch, there is only 528 Kb TCAM and 10 Mb SRAM per Match-Action Unit (MAU) used for various purposes. The TCAM memory can be used to implement ternary matches, longest prefix matching and range matches. The SRAM is typically for storing exact match tables, action data and

stateful externs, such as counters, register arrays and meters. Since the overlay operations require the mapping information for the tunnel endpoints, the naive offloading strategy that offloads all container traffic is clearly impractical considering the extensive scale and short lifespans of containers.

Transparency Requirement *EZPath* is designed for multi-tenant environments, aiming to be application and kernel transparent so that it can work with existing systems and applications. *EZPath* does not require any modifications to user applications or the host kernel, including the virtual switches. This brings additional challenges to the system design. For example, in a containerized cloud shared by multiple tenants, the IP addresses of containers in different tenant networks are assigned locally and independently and thus can be overlapped or even the same, e.g., all of the IP addresses of different containers can be assigned in the range of 10.0.0.1/24. If *EZPath* decides to offload flows from different container networks with overlapped IP addresses, the ToR switch must be able to distinguish them.

Adaptive and Transparent Offloading As discussed earlier, the amount of entries needed for tunneling depends upon the total number of tunnels used by containers in the rack. If the total number is less than the number of available entries at ToR switch, no adaptive offloading is needed and the performance can be maximized. In *EZPath*, we aim to minimize the occupation of SRAM in P4 switches in order to leave sufficient space for accommodating other networking functions. Therefore, we choose to selectively offload the tunneling operations of performance critical flows, e.g., heavy flows and long term flows. That is, we design *EZPath* to adaptively offload resource intensive overlay network operations to the programmable switches at the DC network edge, so that we can effectively relieve the performance bottleneck in the host network stack.

4.4.2 Overview of *EZPath* Design

To this end, Figure 4.9 depicts the major components of *EZPath*. As shown in the figure, it mainly consists of the centralized controller, the in-host software switch and the

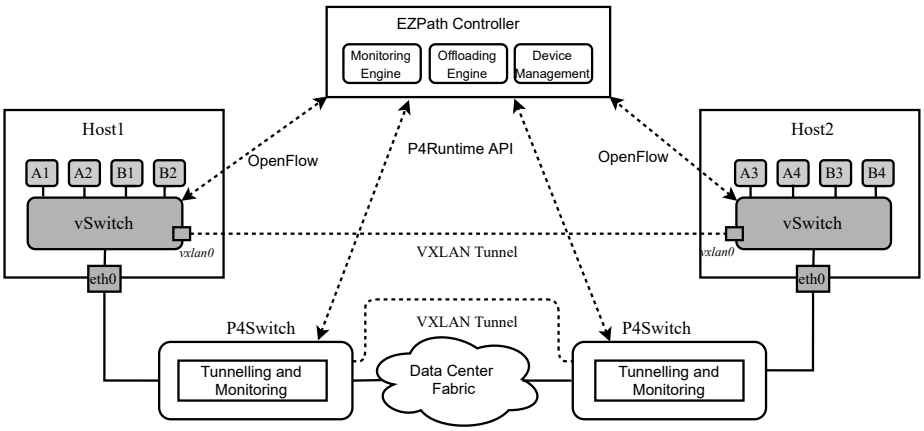

Figure 4.9: *EZPath* design: major components.

programmable ToR switch. The centralized controller monitors the traffic communications between containers and collects the heavy flow information from the ToR switch in real time. Moreover, it is responsible for coordinating the offloading operations between the software switches and the ToR switch. In particular, on the end host, we leverage the widely used OVS in our design. Unoffloaded flows follow the traditional path and are tunneled through the host network stack while ToR switches support heavy flow monitoring and perform the heavy-lifting tunneling operations for offloaded flows. In *EZPath* we use P4 switch as the ToR switch, which provides programmable control for packet monitoring and forwarding.

To maintain application and kernel transparency, ToR switches must be able to distinguish the flows from different tenants. For this purpose, in our design, OVSes leverages the VLAN headers to carry tenant information to ToR switches. For an outgoing packet, the directly attached ToR switch would first strip off the VLAN header, look up tunnelling related information in match-action tables, and perform packet encapsulations. On the other hand, for an incoming packet, the ToR switch would examine if it is the end of the tunnel. If it is the case, it would decapsulate the packet, re-tag the packet with VLAN header and forward it to the connected OVS. Otherwise, the packet is processed similarly as in normal case. VLAN only involves L2 processing and the logic is much simpler by

following a shortened processing path, which greatly reduces the CPU cycles on the end hosts. The detailed steps and data structures involved will be discussed in Section 4.5.

To facilitate heavy flow detection for adaptive offloading, *EZPath* relies on a flow monitoring module. In *EZPath*, we integrate this flow monitoring module within the P4 pipeline. This design choice will be further discussed in Section 4.4.3. The monitoring module keeps track of heavy flow information (i.e., 3-tuple flow identifiers) in stateful P4 object, e.g., Register Externs, which will be retrieved by the control plane through P4Runtime interface. The control plane makes real-time offloading decisions based on multiple factors (subsection 4.4.3). In correspondence to the updated set of offloaded flows, the controller modifies flow rules in the OVSes on the end hosts where the containers reside on and updates the tunnelling mappings in the P4 switches. Furthermore, the controller monitors the hardware resource utilization in real-time and fine-tunes the heavy flow thresholds to maximize the number of offloaded flows within the resource constraint.

Another key design option worth noting is that, in Figure 4.9, although a centralized controller is shown to take full control of the entire DC network or even across the DCs, this is not a hard requirement. Instead, to make *EZPath* more scalable, we can use local controllers, which only manage the software/hardware entities in a single rack. In this way, the ToR P4 switches only serve as overlay proxies that emulate the host virtual tunnel endpoints. To realize this, P4 switches use the L2/L3 addresses of host virtual tunnel endpoints to encapsulate/decapsulate the traffic. On the remote end of the connection, no matter whether the flow is offloaded or not, traffic can still get through and reach the destination containers. With this design, we would anticipate slightly less performance gains than double-end offloading. For realistic deployment, we need to achieve a balanced trade-off between scalability and performance.

4.4.3 Adaptive Offloading Strategy

As the core of the *EZPath*, next we discuss the details of our adaptive offloading strategy that is optimized under the resource constraints of the hardware.

Application-aware offloading

The containerized environment presents unique characteristics due to the diversity of both the deployed applications and the specific usage scenarios in containerized clouds. This inevitably results in more complicated traffic patterns in container data center network. Despite the research advances, monitoring all flows to identify heavy flows in such an environment is very resource consuming. Therefore, we want to minimize such overhead by narrowing down our target flows.

For this purpose, we aim to exclude short flows first. As the previous report [98] shows, containers usually have much shorter lifespans than traditional virtual machines. In particular, around 74% of the containers in production have lifespan shorter than an hour, while 85% live less than a day. Furthermore, containers are also widely used in dev/test environment (e.g., when developing microservice applications), where they only stay up for a really short period of time of up to the order of seconds. Given this observation, we do not foresee significant gains by offloading those flows associated with such short-lived containers. First, it is unusual for these containers to generate heavy/long-lived flows. Second, offloading flows of short-lived containers cause frequent mutation of tunnel mappings and waste switch hardware memory.

To identify such short flows, we turn to the orchestration system. Currently, the deployment and scheduling of containers is managed by specialized container orchestrators (e.g., Kubernetes, Amazon ECS, Mesos and Marathon), which can provide relevant information about container lifespans (e.g., in its configuration file). Therefore, to efficiently leverage the constrained hardware resources, we design an application-aware offloading strategy that considers various factors including real-time packet rate, the projected container lifespans and latency sensitivity of the applications. Following this strategy, the control plane cooperates with the container orchestration tools and makes holistic offloading decisions.

Monitoring placement and optimization

As explained above, a critical part of *EZPath* is adaptive heavy flow monitoring. In *EZPath*, the monitoring module monitors the traffic demands and flow statistics of the containers in the overlay networking. To collect the flow statistics, we can integrate the traffic monitoring capability into OVS on the end hosts, which operates as the host machine edge-router in the virtualized data center. With edge-based monitoring, there are essentially two design options: (1) the SDN controller proactively polls the vSwitch for the flow statistics using the dedicated OpenFlow API; (2) The vSwitch periodically exports specific flow records to the user space using sFlow [17], Netflow [16] or IPFIX [99]. While conducting traffic measurement at OVS is possible, it incurs significant computational overhead and delay. For example, OVS flow forwarding rules may specify that packets are forwarded based on Layer-3 destination address, while the heavy hitters are defined as more fine-grained flows. The discrepancy between the forwarding and monitoring requirements forces us to install much more fine-grained flow rules in OVS that negatively affect the OVS forwarding performance [33]. To capitalize the VxLAN offloading benefits, we keep the local OVS unchanged and place all offloading related functionalities at ToR switch. By leveraging its programmability and line-rate monitoring/forwarding capability, we expect such an option can significantly reduce the overhead incurred by monitoring.

Furthermore, constrained by the limited resource available on the hardware switches, it is infeasible to monitor the flow statistics of all 6-tuple flows (srcIP, dstIP, srcPort, dstPort, protocol, tenant ID) considering the scale of a DC network, the request volume and the variety of container applications. As a workaround, in *EZPath*, we choose to capture only the heavy flows that may vary with the creation, destruction, and migration of containers. As aforementioned, with offloading, Virtual Tunnel End Point (VTEP) mapping information needs to be stored into highly constrained SRAM on the hardware switch. Different from virtual machines, containers are usually lightweight and provisioned as single purpose services. In our monitoring design, a network flow is instead defined as a 3-tuple, including the source and destination container IP address, and VXLAN Network Identifier

(VNI). VNI represents a 24 bit segment ID, which can uniquely identify a single overlay segment. This design further reduces the SRAM space for storing heavy flow information.

Monitoring algorithm

By excluding the short-lived flows and using the 3-tuple for monitoring, we have narrowed down the candidate heavy flows for more efficient usage of the switch memory. For heavy flow detection, we develop a monitoring scheme based on the classical coupon-collector algorithm, a variant of BeauCoup [100] that handles measurement queries using limited hardware resources in the dataplane. Our heavy flow detection can be considered as a distinct counting problem by treating the 3-tuple flow ID as the key and packet timestamp as attribute. The algorithm is depicted in Algorithm 6.

Algorithm 6.

```
1:  procedure FlowOffloadingModule                                   ▷ ControlPlane
2:      CandidateFlows ← dict()
3:      FlowHashTable ← dict()
4:      OffloadedFlows ← dict()
5:      epoch ← 0
6:      while true do
7:          Sleep(T)
8:          epoch ← epoch + 1
9:          HHKeys ← SendRegisterReadRequest(P4Switch)
10:         foreach key ∈ HHKeys do
11:             if key ∈ OffloadedFlows then
12:                 OffloadedFlows[key] ← clock()
13:             else
14:                 UpdateFlowHashTable(key, epoch, FlowHashTable)
15:                 if IsLongLivedHeavyFlow(key, FlowHashTable) then
16:                     AssignPriorityAndAddCandidate(key, priority, CandidateFlows)
17:                 end if
18:             end if
19:         end for
```

```
20:     while CheckP4Unitilization() do
21:       NextKey ← GetHighestPriorityFlow(CandidateFlows)
22:       OffloadedFlows[NextKey] ← clock()
23:       SendTunnelOffloadingRequest(P4Switch, NextKey)
24:       UpdateOvsRules(OVS, NextKey)
25:     end while
26:   end while
27: end procedure
28: procedure HeavyHitterDetection                                  ▷ Dataplane
29:   key ← ExtractKey(packet)
30:   timestamp ← GetTimestamp(packet)
31:   hash ← Hash1(timestamp)
32:   coupon ← FindCoupon(hash)
33:   index ← Hash2(key)
34:   UpdateCouponRegister(index, coupon)
35:   if Count(coupon) > m then                    ▷ m is derived from BeauCoup
36:     report key
37:   end if
38: end procedure
```

During each polling epoch (T), the controller retrieves the registers from the dataplane where heavy flow IDs are recorded. Besides, it maintains a hash table that records all epochs at which each flow is labelled as a heavy flow (line 14). To determine the long-livedness of the heavy flows, we propose a heuristic approach, which labels a flow and adds it into the candidate flow set if it stays heavy for N consecutive polling intervals (line 15). Meanwhile, the configuration parameters relevant to container lifespans and QoS requirements are fetched from the orchestration tools. Collectively, it assigns higher priorities to those heavy flows with stricter latency requirements in order to be application-aware (line 16), as described in Algorithm 6. In addition, another custom thread periodically scans and finds out all inactive offloaded flows, which will be migrated back to host. Details are omitted in the algorithm due to limited space. Meanwhile, the controller monitors real-time

SRAM utilization in P4 switches and offloads the flows based on the assigned priorities(line 20-26). We design this adaptive priority-based offloading strategy due to the following reasons. First, considering the scale and density of containers in microservice deployment, the amount of traffic can be substantial. Even worse, it is normal for containers within a local rack to have both intra-DC and across-DC communications. Simply offloading all flows demands immense switch memory to accommodate the metadata (i.e., the tunnelling mappings). Moreover, realistic network virtualization setup requires switch SRAM for other functions (e.g., security ACL enforcement).

Seamless offloading

Real-time migration is another key component of *EZPath*, which aims to seamlessly offload the flows from the host to P4 switch without disrupting the existing connections. Once the candidate flows for offloading are determined, the controller updates tables on both OVSes and P4 switches according to the following steps: (1) It installs the tunnel mapping for the offloaded containers into the P4 tables on both ToR switches; (2) It modifies OVS flow rules on both end hosts to bypass the host network stack (line 23-24). This is a critical step to guarantee that there is no interruption to the existing connections. Reversing the steps could cause packet drop since there is no tunnel mapping entries in P4 to handle the offloaded packets. On the contrary, flows that become inactive should be migrated back to the host. On the control plane, *EZPath* keeps track of the latest timestamp when each flow stays active. Specifically, in each polling interval, the timestamps in the data structure are updated to reflect the current time when the heavy flows have been detected as active. Flows that have been identified as inactive for a specified duration would be migrated back to host. The implementation details are discussed in Section 4.5.

4.5 Implementation

We have implemented a prototype of *EZPath*. The implementation is about 1000 LOC of P4 in the dataplane and 350 LOC in the control plane. Since the packet processing in the dataplane involves multiple entities along the path between each pair of containers, in the section, we present their implementation details.

In *EZPath*, the OVSes communicate with the controller through OpenFlow protocol, whereas the P4 devices are managed through the P4Runtime interface with gRPC [101] as the underlying communication protocol. Each container is connected to OVS through a veth pair as in the traditional architecture. Traffic originated from or sent to containers goes through the traditional path by following OVS rules that direct traffic through the VXLAN port. The outgoing/incoming packets are encapsulated/decapsulated within the host kernel. For offloaded flows, the rules are updated to add VLAN headers and redirect packets out of the physical interface. The P4 switches maintain an exact match table with tunnelling mapping entries. Specifically, each entry maps the VNI and the destination MAC address to the IP address of the remote VTEP. Each VTEP represents an endpoint of the logical tunnel between the communicating containers. These are managed by the centralized controller in a unified manner. In the P4 switches, for outgoing packets, if the associated flow is offloaded, the switch first maps VLAN ID to VNI and looks up the tunnelling mapping table using the combination of VNI and destination MAC. The offloaded packets are encapsulated and forwarded to the next hop. For incoming packets, the switch performs the decapsulation if the tunnel terminates, by examining the destination L3 addresses. Otherwise, packets are treated and processed as normal. In the meanwhile, flow IDs, $(srcIP, dstIP, VNI)$, are extracted as packets go through the switch pipeline. In *EZPath*, flows are bi-directional since the topology is symmetrical in terms of the tunnelling endpoints. Besides, the associated registers for heavy flow detection are updated accordingly. Once an incoming packet triggers heavy flow detector, it will be injected into control plane via a packet-in message by forwarding the packet to the CPU port (e.g., 64) in the P4 program. In our implementation, we record the heavy flow IDs in a separate set of Register

externs, which can be proactively retrieved by the control plane. The information we record includes the IP addresses of the two communicating endpoints and their tenant ID.

Our control plane is built upon Barefoot Runtime Interface (BRI) that comes with Barefoot SDE (9.1.1). It provides APIs for the control plane to configure and manipulate the dataplane pipeline and objects, such as the match-action tables and stateful objects. On each P4 target, a gRPC server runs and listens for the requests from the control plane, which will be further parsed into target-specific actions. The controller periodically sends a Register read request to ToR P4 switches every T seconds, defined as a polling interval. The collected heavy flows in each interval are constructed into time-series and are analyzed as discussed in Section 4.4.3. Besides, a list is used to record the latest timestamp of each identified heavy flow. We create another custom thread that periodically scans the list to find out flows that have become inactive. For flows that have triggered migration, the rules are updated according to the algorithm discussed in Section 4.4.3.

4.6 Evaluation

In this section, we present our evaluation of *EZPath* following the experiment setup. To quantify the performance improvement, we first use microbenchmarks to study the performance of *EZPath* for tunnel offloading with respect to the normal container overlay networking. Then we evaluate the performance of some typical real-world containerized applications, including an in-memory key value store Memcached [102], ZeroMQ [103] for large scale distributed message library, Nginx [104] for web servers, when adopting *EZPath*.

4.6.1 Experiment Setup

Our testbed consists of two STORDIS BF2556X-1T tofino switches with P4 programmability support and two host servers. Each server is equipped with an Intel Xeon Silver 4110 2.10 GHz CPU, 32 GB DDR4 RAM and NetXtreme-E RDMA 25 Gbps NICs. They are both running Ubuntu Bionic 18.04 LTS OS with Linux kernel 4.15.0. On each host, we leverage Docker 19.03 to create/manage containers and deploy containerized applications.

The switches and servers are directly connected through 25 Gbps cables.

4.6.2 Network Throughput and Latency

First, we compare the performance of different networking modes, i.e., with and without offloading. The former means the default overlay networking, while the latter represents *EZPath*. Specifically, we measure the network performance in terms of both throughput and latency. As we discussed earlier, in offloading mode in *EZPath* the traffic egressing from the containers is directly forwarded to the physical NIC on the host server. The tunnelling-related operations (e.g., packet encapsulation and decapsulation) are performed by the P4 switches. In contrast, in non-offloading mode, the container traffic follows the normal path by traversing both the container and host kernel network stacks.

We use iperf3 to measure the network throughput of a TCP flow in both modes. Each run takes 90 seconds with the messages sizes varied across 128B, 256B, 512B, 1024B and 1440B. The results averaged out of 5 runs under each setting are shown in Figure 4.10.

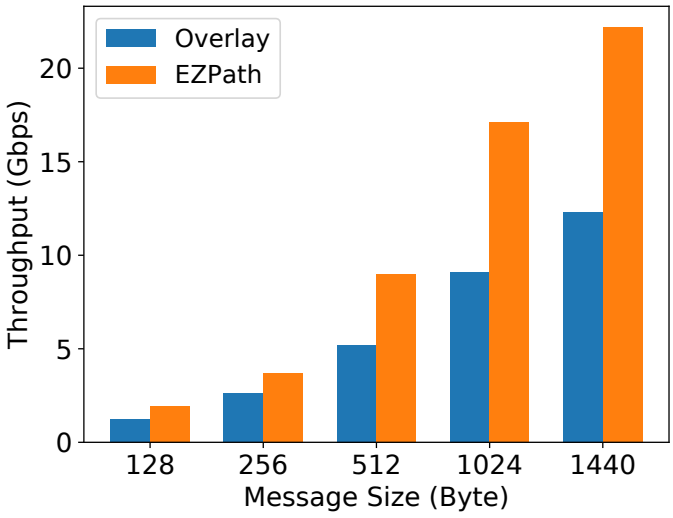

Figure 4.10: Network throughput comparison

Table 4.3: Packet Latency

Networking Mode	RTT (μs)
Overlay	45.0
EZPath	32.8

As we can see, by offloading to the hardware P4 switches, we can significantly improve the throughput performance of the container overlay networking. On average, we see 68% throughput improvement. When the message size is 1440B, the throughput is improved by 80%.

We use sockperf-3.6 to measure the packet latency. The tests are performed in its ping-pong mode, where the latency of single packet is measured without waiting for the reply before sending the subsequent packet on time. As shown in Table 4.3, Without offloading, the measured round trip latency, averaged on 5 runs, is approximately 45 μs for a packet of moderate size 350B. In contrast, with *EZPath* offloading the latency is significantly reduced to 32.8 μs, an improvement of 27%. Similar results are observed in other tests.

4.6.3 Application Performance

In this section, we evaluate the improvement to application performance brought by *EZPath*. For this purpose, we evaluate the performance with popular containerized applications. Considering that there is a wide range of overlay networking solutions, we choose the host networking mode as the baseline in our evaluation. In the host mode, all containers on the same host share the host network namespace. Therefore, they have direct access to all the host's network interfaces. In real-world applications, the host mode is rarely employed due to its inflexibility in supporting multi-tenant cloud applications. In particular, the ports cannot be reused by the same type of applications co-located on the same physical host. For example, once port 80 is assigned to one containerized web server, the other containers would

have to use different ports for their web services to avoid conflicts. Nonetheless, compared to the other networking solutions, the host mode achieves the highest performance despite its inflexibility. Our evaluation adopts similar comparison approaches as used in the existing work by comparing *EZPath* to the baseline host mode and the legacy overlay mode. The relative numbers of the performance improvements can demonstrate how *EZPath* compares with other approaches that are evaluated in different hardware setups.

Memcached Benchmarking

Memcached has been widely used to deploy distributed key-value services in commodity data centers (e.g., Facebook, Google, AWS, Netflix) to improve web service performance.

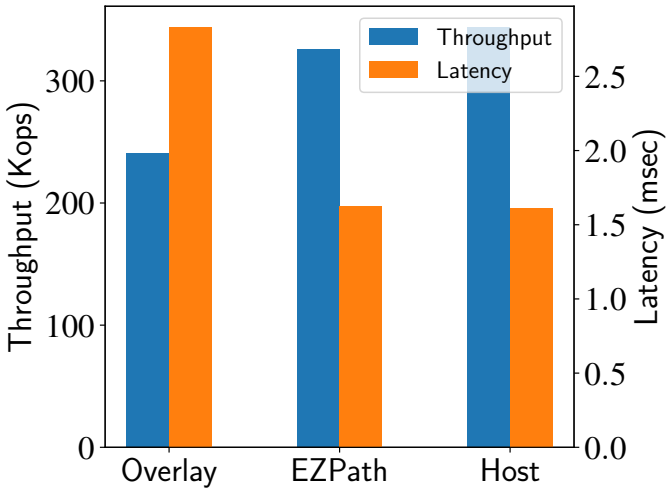

Figure 4.11: Memcached: throughput & latency

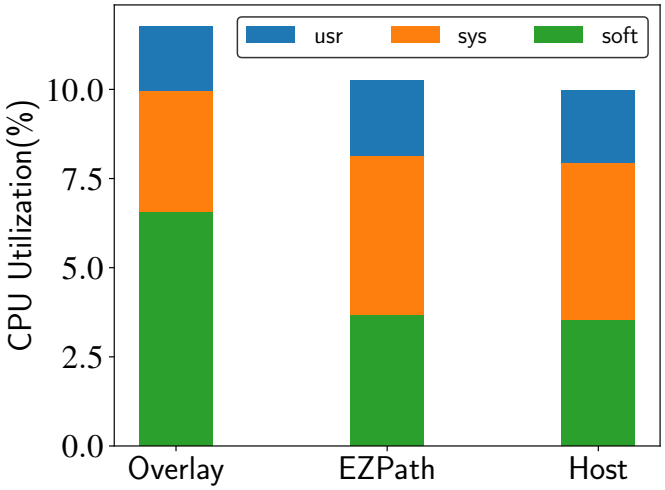

Figure 4.12: Memcached: CPU utilization

In our experiments, we create one Docker container on each physical server. The Memcached server is deployed in one container, and the Memcached benchmarking client runs in the other container. We measure the throughput and latency of the Memcached service in the host mode (baseline), the overlay mode (common practice), and *EZPath*, respectively.

The benchmark tool we use is memtier_benchmark [105] developed by Redis Labs. In our experiments, we use the default settings with four client threads and 50 connections. Each experiment sends 100000 requests and the SET:GET request ratio is set to 10. We repeat the same set of experiments under different modes. The results are averaged over five runs.

Figure 4.11 shows the throughput results in the number of total completed Memcached operations per-second. As we can see, the offloading mode by *EZPath* achieves the throughput comparable to the host mode, and outperforms the overlay mode by 35% on average.

EZPath also reduces Memcached request latency. Figure 4.11 also shows the 99.9^{th} percentile latency to complete a Memcached request. We find the latency through the *EZPath*'s offloading mode is the same as the host mode. Compared to the overlay mode,

EZPath reduces the latency by 42%.

ZeroMQ Benchmarking

ZeroMQ is an asynchronous network messaging library widely deployed in large scale distributed/concurrent systems. Different from brokered message queues (e.g., Apache Kafka [106], ActiveMQ [107], RabbitMQ [108]), ZeroMQ does not rely on brokers and thus achieves much higher throughput. In our experiments, we again measure two key performance metrics, throughput and latency, of ZeroMQ under different modes as before. Each measurement is performed across a wide range of message sizes. On each host, we create a Docker container with ZeroMQ-4.2.2 installed. One container serves as the sender and the other processes the requests as the receiver. The throughput is measured in terms of the number of messages per second; while the latency is measured as the average time it takes to transfer a single message between the two endpoints. The message size varies from 64B to 128KB. For each measurement, the experiment is repeated three times and the average is reported.

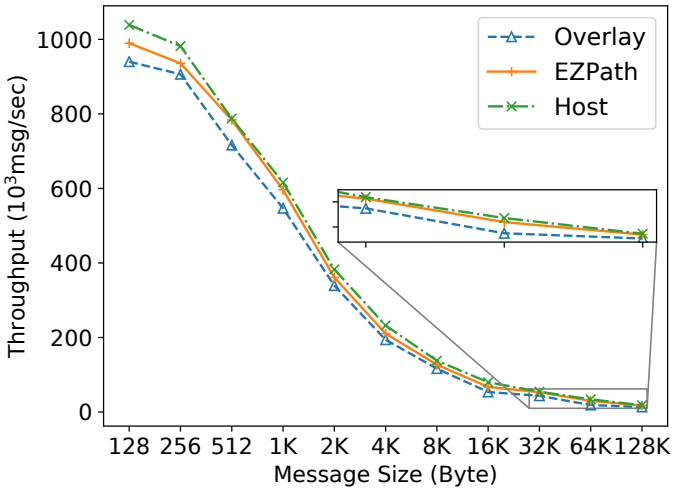

Figure 4.13: ZeroMQ: throughput

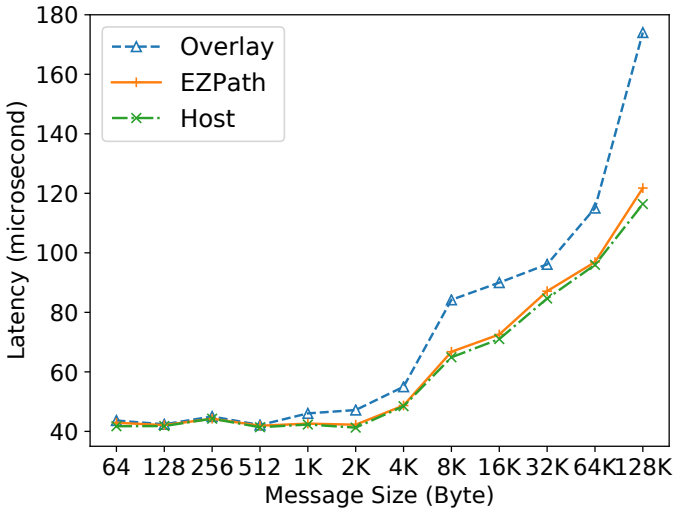

Figure 4.14: ZeroMQ: latency

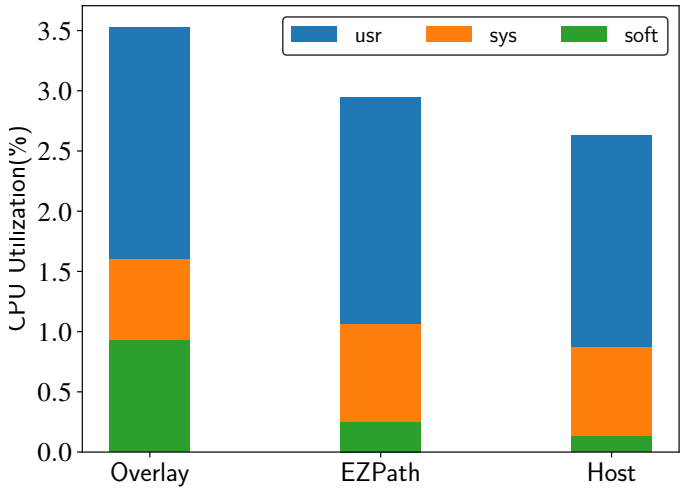

Figure 4.15: ZeroMQ: CPU utilization

Figure 4.13 and Figure 4.14 depict the resulted throughput and latency, respectively. From the figures, we can observe that in all settings, *EZPath* offloading greatly outperforms

the non-offloading counterpart in the overlay mode. Moreover, the improvement becomes more pronounced with the increase of the message size. In particular, for larger message sizes, e.g., 128KB, the throughput is increased by around 42% from 13K msg/sec to 18.5K msg/sec (which is not clearly visible due to the large scale of y-axis in Figure 4.13), while the latency is reduced by 30%. When compared to the host mode, *EZPath* offloading achieves comparable performance with merely slight degradation in both throughput and latency. We believe that this result is acceptable considering the various advantages of offloading over the host mode.

In addition to the application performance, we also examine the variation in the composition of CPU consumption under various modes, mainly including sys, usr, and soft, which represent CPU utilization for executing at user level, system level and service software interrupts, respectively. In our testbed, each server has 16 physical cores with hyperthreading enabled (equally, 32 virtual cores). Therefore, when measuring the CPU overhead breakdown, the CPU utilization can be conveniently converted to the amount of virtual cores. The result are shown in Figure 4.15. As clearly shown in the figure, *EZPath* offloading also reduces CPU cycles significantly. In particular, the CPU cycles spent on serving the software interrupts are reduced by 73%.

Web Server Benchmarking

To study how various networking modes could affect the performance of popular web applications, we run a Nginx container on one host and a client container on the other host. The benchmark software we used is wrk2 [109], which takes throughput as an input argument. The throughput is specified in terms of the total requests per second combined across all connections. Specifically, we create 2 threads in wrk2 and each thread establishes 100 HTTP connections concurrently to the Nginx server. For test purpose, we also randomly generate files with sizes 1KB and 1MB on the web server. In our experiment, the request throughput is set to 10000 and 2500 per second for 1KB and 1MB, respectively. In this way, we can keep the average latency within the order of milliseconds. We measure the average

latency and report the result with its standard deviations computed across multiple runs. Figure 4.16 and Figure 4.17 show the results. As shown in the figure, when requesting 1KB files with 10K requests/sec throughput, the average latency in *EZPath* offloading mode is 1.09ms, which is noticeably smaller than the latency in non-offloading mode (1.16ms). The difference is even larger when requesting 1MB files with 2.5K requests/sec throughput. The average latency in *EZPath* offloading mode is 9.92ms, while the latency in non-offloading mode is 21.68ms, an improvement more than 54%. In both cases, the request processing latency in the offloading mode roughly amounts to that in the host mode.

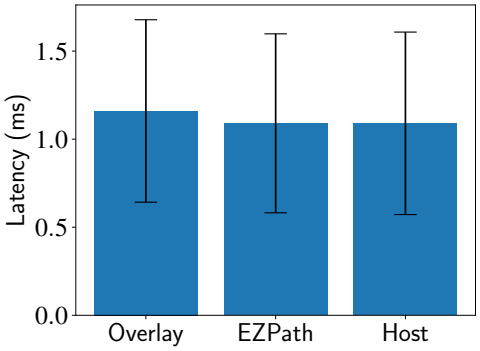

Figure 4.16: Nginx: 1KB files with 10k requests/sec

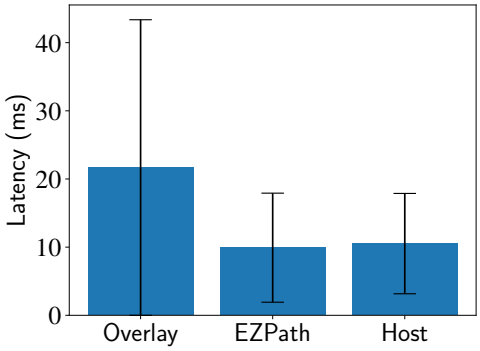

Figure 4.17: Nginx: 1MB files with 2.5k requests/sec

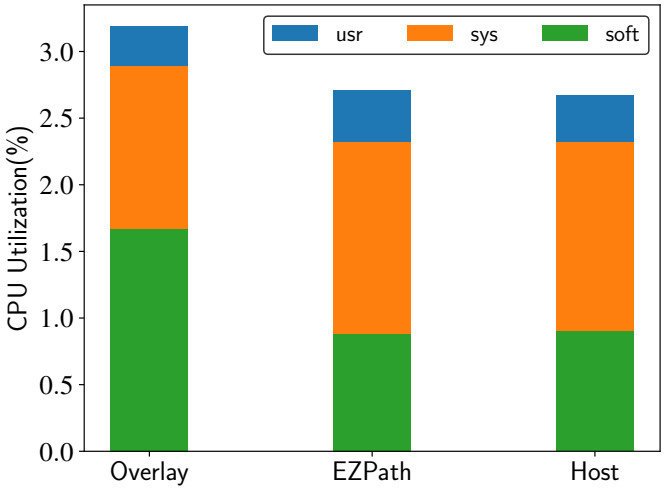

Figure 4.18: Nginx CPU utilization under various modes.

Figure 4.18 further shows that the CPU cycles resulted from serving software IRQs are significantly reduced by approximately 50% in the *EZPath* offloading mode. On the other hand, we observe negligible difference between the CPU cycles incurred in the *EZPath* offloading mode and the host mode. Therefore, the offloading in *EZPath* to the programmable hardware achieves substantially better web performance, with low CPU overhead similar to that of the native host mode. This potentially preserves server resources and opens up further opportunities for overall performance improvement.

4.6.4 Adaptive Offloading Strategy Evaluation

The setup to examine *EZPath*'s adaptive offloading performance is as follows. On host A, we run a Memcached server and a Nginx server in two separate containers. On host B, we create eleven Docker containers, among which one runs memtier_benchmark tool that generates high volume traffic and the other ten containers serve as HTTP clients to generate low-rate background web traffic. In the memtier_benchmark container, we create 4 threads which establish 50 connections with the Memcached server on host A. All HTTP clients

request a 1KB randomly generated file from the Nginx server every 5 seconds. All traffic traverses the ToR switches and is monitored by the dataplane HH (heavy hitter) detection module. We aim to evaluate the performance of our real-time heavy flow identification and offloading. Ideally, our monitoring module should be able to detect the flow between Memcached containers and seamlessly offload the flow to the in-network hardware. In the experiment, the HH detection time interval is set to 100ms and the controller retrieves the heavy flows in each polling interval, which is adjustable and set to 0.5s to avoid too much communication cost. These parameters only affect the delay in offloading the flows.

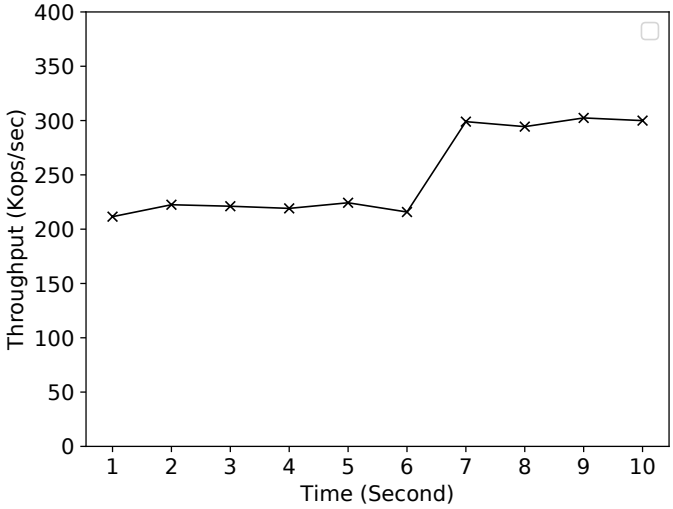

Figure 4.19: Heavy flow identification and seamless offloading for Memcached.

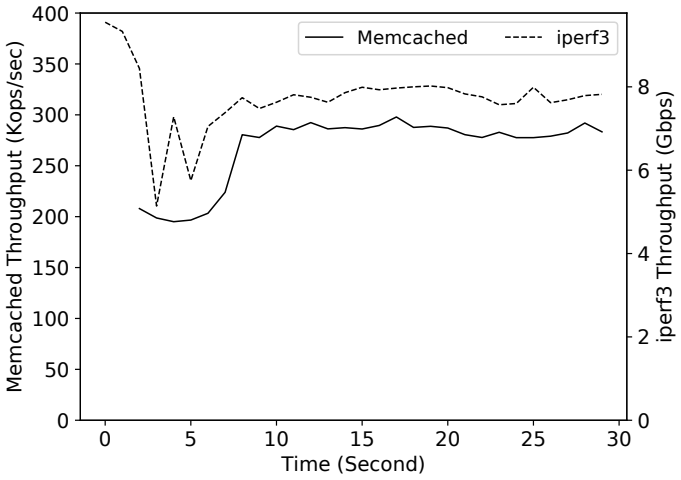

Figure 4.20: Application-aware offloading: multiple applications + background traffic.

At the beginning, flow rules are installed into both OVSes that direct all traffic through the normal path (e.g., the host kernel stack). All HTTP clients start to send background traffic simultaneously before memtier_benchmark is launched. In the first 0.5 second, the controller detects the flow between Memcached and memtier_benchmark container as a HH flow. The controller starts the offloading after the flow remains as a HH flow for 10 polling intervals, which amounts to 5s in our current threshold configuration. In the 6th second, the offloading is triggered. The controller adds tunnelling mapping entries into ToR switches and updates the OVS rules following the process discussed in Section 4.5. Figure 4.19 shows that the Memcached traffic is seamlessly offloaded without any interruption. Furthermore, as also shown in Figure 4.19, in the 7th second after the offloading is completed, the throughput of Memcached is increased by 30%, from around 225Kops to 300Kops (operations per second), which further demonstrates the performance enhancement of *EZPath*.

To demonstrate the application-awareness of *EZPath* offloading strategy, we examine the performance of *EZPath* in-depth with multiple high-rate benchmarking applications executing concurrently, namely Memcached and iperf3, mixed with ten low-rate background HTTP flows. Background traffic is injected similarly as before. Another pair of containers

are added to run iperf3 server and client separately on the two hosts, with packet size of 1024B. Besides, iperf3 server and client are launched 2 seconds before Memcached server and memtier_benchmark threads are launched. In memtier_benchmark, we create 4 thread, 50 connections per thread with 50000 requests per client. In the controller, Memcached is assigned a higher priority according to Kubernetes. During the experiment, we observed that both high-rate flows (Memcached and iperf3) can be captured by our HH detection module and controller. With the priorities, the controller only migrates the Memcached flow, leaving the iperf3 flow untouched. The dynamics of the throughput for both Memcached and iperf3 are illustrated in Figure 4.20. As is shown in the figure, with Memcached traffic started at the 2nd second, iperf3 throughput suffers from moderate drops, competing the bandwidth with the other flow. In the 8th second, the Memcached flow is stealthily migrated to the P4 hardware, with the throughput increased from around 210Kops to 285Kops or so. Starting from 2nd second, due to the newly joined Memcached flow, iperf3 goes through TCP congestion control stage. Around 8th second, it reaches the stable state. This efficaciously demonstrates *EZPath* is application-aware and could significantly improve the average network performance.

As aforementioned, the major constraint for offloading is the available amount of SRAM in the switch. In our implementation, each tunnel mapping entry takes up 104 bits (48b MAC + 24b vni + 32b IP) of switch memory. In the switch we used for the experiments, there are 10Mb SRAM per MAU. If 30% SRAM is used for offloading, it can accommodate approximately 30K entries. Thus, depending on the application context, it may necessitate the offloading priority as we discussed in section 4.4 and demonstrated here.

4.6.5 Discussion

To achieve full optimization, *EZPath* requires a centralized control plane, which controls and manages the software/hardware network devices over the entire DC and across DCs. If containers are deployed and replicated across data centers (e.g., to enhance service reliability and availability), it will place burden on the centralized control plane to manage such large

scaled networks. One workaround to mitigate this is to use local controllers that make offloading decisions on their own, without coordination between the endpoints. As pointed out in Section 4.4, this may not fully optimize the application performance due to in-network resource under-utilization. We plan to study this in our future research.

On the other hand, the offloading function offered by *EZPath* does not have to be tied to any particular offloading algorithm. The adaptive offloading strategy we have demonstrated can be replaced based on the application's need. Essentially, regardless of how the target flow is identified, *EZPath* can offload that seamlessly.

4.7 Related Work

Efficiently and flexibly managing containerized clouds poses a myriad of challenges from different aspects [77, 84, 85, 87, 88, 110]. Among them, the network performance degradation due to the software based overlay networking has attracted a lot of attention [84–86]. This is mainly due to the overhead when implementing the tunneling and other network virtualization functionalities in the software switch, such as Open vSwitch, a key component in Weave [80], one of the most widely used container network interface (CNI) plugins for production container orchestration platforms such as Kubernetes [81], Apache Mesos [82], and Amazon ECS [83]. Approaches like SR-IOV [84] and Macvlan [95] aim to assign physical network devices to selected containers, but they are not suitable for containerized clouds. For example, SR-IOV is limited by the simulated 64 virtual functions (VF) while container networks often have massive scales [111, 112], and is also incurs extra overhead [113] in networking among intra-host containers as discussed in section 4.3.2.

Recent designs [77, 85, 87], on the other hand, either demand hardware support or fail to support legacy monitoring and debugging tools in data centers. For example, Slim [85] can achieve promising performance boosts by redesigning container overlay networks with a dedicated user-space router to reduce the packet traversal of OS-kernel network stack. But it is impractical for real world deployment due to the customized software demand. Compared to them, *EZPath* is both application and system kernel transparent, and is compatible with

any existing tools, thus offering a transparent alternative.

Due to resource constraints, *EZPath* adaptively monitors and offload heavy flows. Heavy hitter detection algorithms have been extensively studied [114], [23], [115]. However, they do not work well in highly resource constrained hardware switches. To address the resource constraint, some recent work has re-designed the algorithms and implementation in order to work around the limitations, such as HashPipe [7] and Precision [8]. However, it has been demonstrated that HashPipe is hard to realize in RMT hardware models [24] because the implementation requires consecutive pipeline stages to access the same stateful memory block. Precison eliminates the requirement by recirculating a small portion of packets so that recirculated packet only accesses the same memory in different passes. This inevitably degrades the performance and complicates the system design. Therefore, in *EZPath*, we use multiple heuristics to improve the detection efficiency while reducing the runtime overhead.

4.8 Chapter Summary

Cloud computing is increasingly adopting containers, evidenced by the popularity of microservices offered by all major cloud service providers. However, containerized applications often suffer from the degraded networking performance in the common practice due to the extra processing overhead induced by the container overlay. Previous solutions addressing this problem are either not compatible with the legacy monitoring and debugging tools, or demanding customized hardware support. In this work, we have instead designed and implemented *EZPath* to expedite the container traffic, by leveraging the high speed of the programmable switches in data centers. *EZPath* is transparent to applications or the underlying system kernel, compatible with all existing tools. Considering the resource constraints in the hardware switches, *EZPath* can carefully identify heavy-hitters and adaptively offload them. Our evaluation shows it can significantly expedite container traffic by improving the throughput and reducing the latency of typical containerized applications.

Chapter 5: Conclusion and Future Work

Cloud computing platforms heavily rely on the underlying data center systems. Among various components, the data center networks (DCNs) play a critical role in response to the ever-increasing demand of applications and attacks. Therefore, efficient monitoring and management of data center traffic is essential.

5.1 Conclusion

In this dissertation, we investigate novel solutions to address several challenges posed to the data center networks with the support of modern networking technology, e.g., SDN. First, we design and implement flexible and efficient solutions to address DCN monitoring and management challenges. For this, we have empirically explored the various trade-offs among these factors by designing, implementing, and evaluating five different monitoring schemes. Among them, four are built into the OVS kernel datapath, while the other one is based on eBPF that leaves OVS intact. They differ in various aspects, including the placement of monitoring modules, the data structures for maintaining traffic statistics, and the interaction with OVS. Based on extensive experiments, we have quantitatively showed their advantages and disadvantages. These results provide insightful guidelines for conducting network traffic measurement on the OVS as well as software defined measurement in general.

With these insights on how to conduct efficient and accurate measurement at the network edge, e.g., edge-based software switches, we have further delved into how to leverage edge assisted measurement to enhance data center security. Accordingly, we proposed BotSifter, an accurate, scalable and runtime framework for bot detection in data centers by incorporating state-of-the-art machine learning techniques. Particularly, we focused on edge-based detection since it is inherently scalable through distributing the monitoring load across the

edge nodes. In the meantime, it enables more expeditious detection due to the feasibility of migrating the security applications to the edge, which not only supports more timely detection but also reduces the communication bandwidth between the monitoring nodes and the centralized monitoring server (e.g., the SDN controller).

Contemporary DCNs are often faced with performance bottlenecks with the increasing network scales and traffic speeds, due to legitimate or attacking traffic. To mitigate such bottlenecks, we have designed *EZPath*, which focuses on the performance optimization of container overlay networks by leveraging the emerging programmable dataplanes. *EZPath* requires no modifications to either the applications or the system kernel and thus easy to maintain. Besides, *EZPath* identifies performance-critical flows and intelligently offloads them, making the best use out of the limited resources within the hardware devices. Our evaluations based on a wide range of microbenchmarks and application benchmarks have demonstrated that *EZPath* can significantly expedite container traffic by improving the throughput and reducing the latency of typical containerized applications.

5.2 Future Work

BotSifter takes the first step towards automating DC network security defenses by applying ML models to perform runtime botnet detection. However, since it relies on supervised ML models to learn from past traffic samples, it becomes less effective against new forms of attacks. Furthermore, upon detection, it still requires non-trivial human interventions to deploy corresponding security policies. In recent years, reinforcement learning (RL) techniques have been studied to learn control policies that automatically map environment states to actions. Examples include data center traffic optimization [116], cache replacement in storage systems [117], TCP congestion control [118] and packet classification [119].

In the future, we will explore how RL could be leveraged to build a fully automated DCN defense system. Our system consists of several key building blocks, including a scalable data collection module to collect fine-grained information from the end hosts and in-network devices and a RL enabled learning agent that interacts with the network environment to

optimize the RL model in an online manner. More specifically, the DCN defense system will be formulated as a RL model, where the collected information is defined as the network environment state and the defense strategies as the actions. Our goal is to automate the defense policies by learning the optimized mapping between the network state and the defense actions.

Bibliography

Bibliography

[1] "Openflow version 1.5.1," http://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, 2018.

[2] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[3] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch." in *Usenix NSDI*, 2013.

[4] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Dream: dynamic resource allocation for software-defined measurement," in *ACM SIGCOMM CCR*, 2014.

[5] ——, "Scream: Sketch resource allocation for software-defined measurement," in *ACM CoNEXT*, 2015.

[6] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers." in *Usenix NSDI*, 2016.

[7] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*. ACM, 2017, pp. 164–176.

[8] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient measurement on programmable switches using probabilistic recirculation," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 313–323.

[9] R. B. Basat, G. Einziger, I. Keslassy, A. Orda, S. Vargaftik, and E. Waisbard, "Memento: Making sliding windows efficient for heavy hitters," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, 2018, pp. 254–266.

[10] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen, "Umon: Flexible and fine grained traffic monitoring in open vswitch," in *ACM CoNEXT*, 2015.

[11] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 129–143.

[12] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. 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.

[13] "Open vSwitch," http://openvswitch.org/, 2017.

[14] M. T. Goodrich and M. Mitzenmacher, "Invertible bloom lookup tables," in *Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2011.

[15] M. Fleming, "A thorough introduction to ebpf," 2017.

[16] B. Claise, "Cisco systems netflow services export version 9," 2004.

[17] S. Panchen, P. Phaal, and N. McKee, "Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks," 2001.

[18] B. Claise, "Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information," 2008.

[19] Juniper, "Juniper flow monitoring," 2011.

[20] Alcatel-Lucent, "Cflowd," https://infoproducts.alcatel-lucent.com/html/0_add-h-f/93-0073-HTML/7750_SR_OS_Router_Configuration_Guide/Cflowd-Intro.html, 2017.

[21] HP, "Hp netstream monitoring module," 2012.

[22] 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 SIGCOMM Conference*. ACM, 2016, pp. 101–114.

[23] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International Conference on Database Theory*. Springer, 2005, pp. 398–412.

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

[25] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM journal on computing*, vol. 31, no. 6, pp. 1794–1813, 2002.

[26] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows." in *INFOCOM*, 2016, pp. 1–9.

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

[28] 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.

[29] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 561–575.

[30] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 334–350.

[31] E. J. Candès, J. Romberg, and T. Tao, "Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information," *IEEE Transactions on information theory*, vol. 52, no. 2, pp. 489–509, 2006.

[32] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[33] 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. 1–7.

[34] 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 *Usenix NSDI*, 2015.

[35] X. Li, F. Bian, M. Crovella, C. Diot, R. Govindan, G. Iannaccone, and A. Lakhina, "Detection and identification of network anomalies using sketch subspaces," in *ACM IMC*, 2006.

[36] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *ACM IMC*, 2004.

[37] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in data streams," in *VLDB*, 2003.

[38] Y. Zhang, "An adaptive flow counting method for anomaly detection in sdn," in *ACM CoNEXT*, 2013.

[39] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. ACM, 2018, pp. 54–66.

[40] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*. IEEE, 2014, pp. 120–125.

[41] "Ryu SDN controller," https://osrg.github.io/ryu/, 2017.

[42] "Caida internet traces 2012," https://www.caida.org/data/passive/passive_2012_dataset.xml, 2012.

[43] "Tcpreplay," http://tcpreplay.appneta.com/, 2017.

[44] L. Columbus, "83% Of Enterprise Workloads Will Be In The Cloud By 2020," 2018. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/#32e7847e6261

[45] P. McDougall, "Microsoft: Kelihos Ring Sold 'Botnet-As-A-Service'," 2011-09-30. [Online]. Available: https://www.darkreading.com/risk-management/microsoft-kelihos-ring-sold-botnet-as-a-service/d/d-id/1100470?piddl_msgorder=thrd

[46] K. Clark, M. Warnier, and F. M. Brazier, "Botclouds-the future of cloud-based botnets," in *in CLOSER*. Citeseer, 2011.

[47] [Online]. Available: https://securelist.com/elasticsearch-vuln-abuse-on-amazon-cloud-and-more-for-ddos-and-profit/65192/

[48] [Online]. Available: https://www.elastic.co/

[49] B. Butler, "Hackers found controlling malware and botnets from the cloud," 2014-06-26. [Online]. Available: https://www.networkworld.com/article/2369887/cloud-security/hackers-found-controlling-malware-and-botnets-from-the-cloud.html

[50] J. Goebel and T. Holz, "Rishi: Identify bot contaminated hosts by irc nickname evaluation." *HotBots*, vol. 7, pp. 8–8, 2007.

[51] G. Gu, P. A. Porras, V. Yegneswaran, M. W. Fong, and W. Lee, "Bothunter: Detecting malware infection through ids-driven dialog correlation." in *USENIX Security Symposium*, vol. 7, 2007, pp. 1–16.

[52] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection," 2008.

[53] G. Gu, J. Zhang, and W. Lee, "Botsniffer: Detecting botnet command and control channels in network traffic," 2008.

[54] J. Zhang, R. Perdisci, W. Lee, U. Sarfraz, and X. Luo, "Detecting stealthy p2p botnets using statistical traffic fingerprints," in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. IEEE, 2011, pp. 121–132.

[55] D. Zhao, I. Traore, B. Sayed, W. Lu, S. Saad, A. Ghorbani, and D. Garant, "Botnet detection based on traffic behavior analysis and flow intervals," *Computers & Security*, vol. 39, pp. 2–16, 2013.

[56] E. B. Beigi, H. H. Jazi, N. Stakhanova, and A. A. Ghorbani, "Towards effective feature selection in machine learning-based botnet detection approaches," in *Communications and Network Security (CNS), 2014 IEEE Conference on*. IEEE, 2014, pp. 247–255.

[57] T. Holz, M. Steiner, F. Dahl, E. Biersack, F. C. Freiling *et al.*, "Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm." *LEET*, vol. 8, no. 1, pp. 1–9, 2008.

[58] B. B. Kang, E. Chan-Tin, C. P. Lee, J. Tyra, H. J. Kang, C. Nunnery, Z. Wadler, G. Sinclair, N. Hopper, D. Dagon *et al.*, "Towards complete node enumeration in a peer-to-peer botnet," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. ACM, 2009, pp. 23–34.

[59] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich, "Analysis of the storm and nugache trojans: P2p is here," *USENIX; login*, vol. 32, no. 6, pp. 18–27, 2007.

[60] G. K. Venkatesh and R. A. Nadarajan, "Http botnet detection using adaptive learning rate multilayer feed-forward neural network," in *IFIP International Workshop on Information Security Theory and Practice*. Springer, 2012, pp. 38–48.

[61] T. Cai and F. Zou, "Detecting http botnet with clustering network traffic," in *Wireless Communications, Networking and Mobile Computing (WiCOM), 2012 8th International Conference on*. IEEE, 2012, pp. 1–7.

[62] L. Carl *et al.*, "Using machine learning technliques to identify botnet traffic," in *Local Computer Networks, Proceedings 2006 31st IEEE Conference on. IEEE*, 2006.

[63] J. Leonard, S. Xu, and R. Sandhu, "A framework for understanding botnets," in *Availability, Reliability and Security, 2009. ARES'09. International Conference on*. IEEE, 2009, pp. 917–922.

[64] S. Garcıa, "Identifying, modeling and detecting botnet behaviors in the network," *Unpublished doctoral dissertation, Universidad Nacional del Centro de la Provincia de Buenos Aires*, 2014.

[65] [Online]. Available: https://www.tensorflow.org/

[66] S. Garcia, M. Grill, J. Stiborek, and A. Zunino, "An empirical comparison of botnet detection methods," *computers & security*, vol. 45, pp. 100–123, 2014.

[67] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection," *computers & security*, vol. 31, no. 3, pp. 357–374, 2012.

[68] B. Rahbarinia, R. Perdisci, A. Lanzi, and K. Li, "Peerrush: Mining for unwanted p2p traffic," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013, pp. 62–82.

[69] "Building products at soundcloud —part i: Dealing with the monolith — soundcloud backstage blog," https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith.

[70] "Building netflix's distributed tracing infrastructure — by netflix technology blog — netflix techblog," https://netflixtechblog.com/building-netflixs-distributed-tracing-infrastructure-bb856c319304?gi=24ea49f7eb2c.

[71] "What led amazon to its own microservices architecture – the new stack," https://thenewstack.io/led-amazon-microservices-architecture/.

[72] "Monolith to microservices: Transforming a web-scale, real-world e-commerce platform using the strangler pattern — runscope blog," https://blog.runscope.com/posts/monolith-microservices-transforming-real-world-ecommerce-platform-using-strangler-pattern?author=58ec33a515d5db8a63ffa523.

[73] "Apache ignite as an inter-microservice transactional in-memory data store — by dishan metihakwala — the startup — medium," https://medium.com/swlh/apache-ignite-as-an-inter-microservice-transactional-in-memory-data-store-6f599f7b5792.

[74] "Aws lambda – serverless compute - amazon web services," https://aws.amazon.com/lambda/.

[75] "Cloud computing services — google cloud," https://cloud.google.com/.

[76] "Azure service fabric—building microservices — microsoft azure," https://azure.microsoft.com/en-us/services/service-fabric/?&ef_id=Cj0KCQiA3NX_BRDQARIsALA3fII8DVilRzKU4ZRvOYTJ-k1PokLIORFBBL4M5bpIMjcko8lDNb6vnKUaAsicEALw_wcB:G:s&OCID=AID2100131_SEM_Cj0KCQiA3NX_BRDQARIsALA3fII8DVilRzKU4ZRvOYTJ-k1PokLIORFBBL4M5bpIMjcko8lDNb6vnKUaAsicEALw_wcB:G:s&gclid=Cj0KCQiA3NX_BRDQARIsALA3fII8DVilRzKU4ZRvOYTJ-k1PokLIORFBBL4M5bpIMjcko8lDNb6vnKUaAsicEALw_wcB.

[77] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar, "Freeflow: High performance container networking," in *Proceedings of the 15th ACM workshop on hot topics in networks*, 2016, pp. 43–49.

[78] S. Thomas, L. Ao, G. M. Voelker, and G. Porter, "Particle: ephemeral endpoints for serverless networking," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 16–29.

[79] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 117–130.

[80] "Introducing weave net," https://www.weave.works/docs/net/latest/overview/.

[81] "Kubernetes," https://kubernetes.io/.

[82] "Apache Mesos," http://mesos.apache.org/.

[83] "Amazon Elastic Container Service," https://aws.amazon.com/ecs/features/.

[84] R. Niranjan Mysore, G. Porter, and A. Vahdat, "Fastrak: enabling express lanes in multi-tenant data centers," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, 2013, pp. 139–150.

[85] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim:{OS} kernel support for a low-overhead container overlay network," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 331–344.

[86] J. Lei, K. Suo, H. Lu, and J. Rao, "Tackling parallelization challenges of kernel network stack for container overlay networks," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[87] R. Nakamura, Y. Sekiya, and H. Tazaki, "Grafting sockets for fast container networking," in *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, 2018, pp. 15–27.

[88] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, "Iron: Isolating network-based {CPU} in container environments," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 313–328.

[89] Docker, "Docker container networking," https://docs.docker.com/v17.09/engine/userguide/networking/, 2019.

[90] R. Jones *et al.*, "Netperf: a network performance benchmark," *Information Networks Division, Hewlett-Packard Company*, 1996.

[91] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu, "iperf3, tool for active measurements of the maximum achievable bandwidth on ip networks," *URL: https://github. com/esnet/iperf*.

[92] Q. O. Snell, A. R. Mikler, and J. L. Gustafson, "Netpipe: A network protocol independent performance evaluator," in *IASTED international conference on intelligent information management and systems*, vol. 6, 1996, p. 49.

[93] "mpstat(1): Report processors related statistics - linux man page," https://linux.die.net/man/1/mpstat.

[94] Brendan Gregg, "Linux Extended BPF (eBPF) Tracing Tools," http://www.brendangregg.com/ebpf.html, 2009.

[95] "Use macvlan networks — docker documentation," https://docs.docker.com/network/macvlan/.

[96] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, 2012.

[97] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *2016 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2016, pp. 1–7.

[98] "Why are containers so short-lived?" https://www.infoworld.com/article/3293260/why-are-containers-so-short-lived.html.

[99] B. Claise, B. Trammell, and P. Aitken, "Specification of the ip flow information export (ipfix) protocol for the exchange of flow information," in *RFC 7011 (INTERNET STANDARD), Internet Engineering Task Force*, 2013.

[100] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "Beaucoup: Answering many network traffic queries, one memory update at a time," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 226–239.

[101] Google, "GRPC," https://github.com/grpc/grpc.

[102] "Memcached," http://memcached.org/, 2019.

[103] zeromq.org, "ZeroMQ: An open-source universal messaging library," https://zeromq.org/.

[104] "Nginx," https://www.nginx.com/.

[105] RedisLabs, "memtier_benchmark," https://github.com/RedisLabs/memtier_benchmark, 2019.

[106] "Apache Kafka," https://kafka.apache.org/.

[107] "Apache ActiveMQ," http://activemq.apache.org/.

[108] "RabbitMQ," https://www.rabbitmq.com/.

[109] "wrk2," https://github.com/giltene/wrk2, 2019.

[110] C. Xu, K. Rajamani, and W. Felter, "Nbwguard: Realizing network qos for kubernetes," in *Proceedings of the 19th International Middleware Conference Industry*, 2018, pp. 32–38.

[111] IBM, "Docker at insane scale on IBM Power Systems," https://www.ibm.com/cloud/blog/docker-insane-scale-on-ibm-power-systems, 2015.

[112] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained nfs for flexible per-flow customization," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 2016, pp. 3–17.

[113] "Chapter 8. performance and optimization reference architectures 2017 — red hat customer portal," https://access.redhat.com/documentation/en-us/reference_architectures/2017/html/deploying_mobile_networks_using_network_functions_virtualization/performance_and_optimization.

[114] X. Dimitropoulos, P. Hurley, and A. Kind, "Probabilistic lossy counting: an efficient algorithm for finding heavy hitters," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 1, pp. 5–5, 2008.

[115] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Transactions on Database Systems (TODS)*, vol. 28, no. 1, pp. 51–55, 2003.

[116] L. Chen, J. Lingys, K. Chen, and F. Liu, "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proceedings of the 2018 conference of the ACM special interest group on data communication*, 2018, pp. 191–205.

[117] L. V. Rodriguez, F. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan, "Learning cache replacement with {CACHEUS}," in *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, 2021, pp. 341–354.

[118] W. Li, F. Zhou, K. R. Chowdhury, and W. Meleis, "Qtcp: Adaptive congestion control with reinforcement learning," *IEEE Transactions on Network Science and Engineering*, vol. 6, no. 3, pp. 445–458, 2018.

[119] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 256–269.

Curriculum Vitae

Zili Zha received her Bachelor of Science degree in Computer Science from University of Science and Technology of China in 2009. She received her Master of Science degree in Applied Science from College of William and Mary in 2013.