Paving the Way for NFV

By

Junaid Khalid

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2018

Date of final oral examination: 07/18/2018

The dissertation is approved by the following members of the Final Oral Committee:

Aditya Akella, Professor, Computer Sciences
Remzi Arpaci-Dusseau, Professor, Computer Sciences
Kassem Fawaz, Assistant Professor, Electrical & Computer Engineering
Eric Rozner, Research Staff Member, IBM Research
Michael Swift, Professor, Computer Sciences

Abstract

Network functions (NF) or middleboxes play a vital role in improving the performance and ensuring the security of networks, along with providing other network functionalities. Recent years have seen an increased trend towards network functions virtualization (NFV) by replacing traditional middleboxes with virtualized instances of NFs. Such NFs run on top of generic compute resources. NFV enables network operators to realize custom network policies by traversing network traffic through sequence of NFs (NF chains). Despite the fact that NFV promises a greater flexibility to operators for network handling, its potential has not been fully realized. In this dissertation, we designed systems to overcome some of the major hindrances in NFV adoption.

One of our major contributions is showing that containers do not provide hard enough isolation needed to deploy network intensive applications such as NFs. To mitigate this issue we designed a scheme named Iron, this scheme ensures hard isolation of network-based CPUs in containerized environment. Iron accounts for the time spent in the networking stack on behalf of a container and ensures this processing cannot adversely impact co-located containers through novel enforcement mechanisms.

As our second contribution, we designed CHC, a ground-up framework to ensure chain-wide correctness and provide good performance. To ensure fault tolerance and hide the complexity of elastic scaling from the NF developers, CHC relies on manging state, external to NFs. It couples

state externalization with novel caching and state updating mechanisms to ensure high performance and correctness.

Our third contribution is to simplify the process of modifying NF codes to support NFV frameworks. To realize this goal we designed a system termed as StateAlyzr, which automates the process of modifying existing NFs to support new frameworks with minimal manual effort. StateAlyzr leverages the tools from program analysis to design new algorithms that can provably and automatically identify all the states that must be cloned/migrated to ensure consistent NF output during redistribution of network traffic.

To my parents,
Ruby, Ruby, Ruby, and Khalid

Acknowledgments

This dissertation would have not been possible without the support and encouragement from a lot of people, to whom I wish to thank.

First and foremost I would like to thank my advisor and my mentor, Aditya Akella for his dedication and support. I am extremely grateful for all the freedom he gave me in pursuing my research interests. His feedback on my research work was always extremely insightful and it helped me shape and develop my research ideas into a mature work. He also guided me on creating simple but elegant solutions for complex problems

I am deeply indebted to Eric Rozner for the constant support and encouragement that he provided not only during our collaboration but even after that. He not only helped me with polishing research ideas but also spent a tremendous amount of time in debugging the code with me and other minor details to make a solid paper.

Aaron Gember-Jacobson has not only been an awesome collaborator and fun person to work with, he also taught me how to do good research in the early years of my PhD.

I am grateful to all my peers and collaborators who contributed their time and insights: Sharad Aggarwal, Mark Coatsworth, Sourav Das, Wesley Felter, Alexandre Ferreira, Keqiang He, Roney Michael, Anubhav Nidhi, Jitu Padhye, Karthick Rajamani, Arjun Singhvi and Cong Xu.

I have been lucky enough to have found a lot of wonderful friends and colleagues who were always there to share the ups and downs of PhD life: Osama Khalid, my brother and my best friend was always there

to provide his support in all possible ways. His support has played a key role in the success of my PhD life. Aimal Khan, provided constant support, despite being remote. Robert Grandl was not just my officemate but also an amazing friend, who was always there to encourage and push me. Shoban Preeth, who shared my love for food, was always there to provide his insightful and unbiased opinions over dinner. Rohit Bhat and Jyotsna Negi who were always ready to go on roadtrips with me. Despite the distance, Rohit was always able to take time out of his schedules to give me advice whenever I needed it and Jyotsna was always there with the bad relationship advice. Raajay Viswanathan for procrastinating with me during our PhDs, and barely managing to meet the deadlines. I would also like to thank Theo Benson for being there in the difficult moments and providing his support remotely. And a similar thanks for all my other friends who were always there to provide their support Shoaib Bin Altaf, Shaleen Deep, Ram Durairajan, Mobin Javed, Yanfang Le, Kshiteej Mahajan, Haroon Raja, Brent Stephens, Kaushik Subramanian, and Wenfei Wu.

I appreciate the valuable feedback provided by thesis committee members: Remzi Arpaci-Dusseau, Kassem Fawaz, Eric Rozner and Michael Swift.

I am extremely grateful to Syed Ali Khayam and Syed Akbar Mehdi, who helped discover and hone my passion for research.

I am extremely grateful for my brother Talha and my parents. Despite been in a different continent, they were always able to provide encouragement. My PhD journey would have not been possible without their continuous support and guidance.

Contents

List of Figures and Tables

1 Introduction

Earlier networks were designed to just provide connectivity between different devices by forwarding and routing packets. The growth of networks gave rise to new services, devices, applications and use cases. These applications evolved to provide more than just forwarding.

Nowadays, network applications range from ensuring security by inspecting and blocking malicious traffic to improving performance by caching content close to applications or improving data transfers by compressing data, enabling new protocols by acting as proxies to provide translations or just as a point to account and monitor traffic to support billing and diagnosis.

Network operators provide these services by using specialized hardware devices called "middleboxes" or "network functions". These devices sit on-path or off-path in the network and perform sophisticated analysis on the traffic that flows through them. Some example of middleboxes that commonly exists are:

- **Intrusion detection/prevention systems (IDS/IPS)**: These devices inspect the packet headers and payloads to detect anomalous behaviour. On identifying anomalous behaivour they can raise alarms for the administrator and may also drop the malicious packets.
- **Caching proxies**: They are used for improving the response time of a user's request by caching the hot content close to the user.

- **Load balancers**: They are used to distribute load between different backend servers. Load balancers can operate from L3 to L7, depending upon their requirements.

Recently, there has been a growing interest in replacing hardware based middleboxes with software middleboxes, also known as network functions. Unlike dedicated hardware, these NFs run on generic compute resources. This trend of running NFs as software on generic compute resources, rather than dedicated hardware, is known as network function virtualization (NFV) [11].

Network function virtualization promises to offer networks a great flexibility in handling middlebox load spikes by spinning up new virtual instances and dynamically redistributing traffic, among these instances. NFV promises to revolutionized the deployment of NFs by:

1. Reducing the operational cost of running NFs by consolidating resources and dynamically allocating new instances and route traffic through them as the load increases.

2. Simplifying the task of deploying, configuring and managing of NFs. Unlike hardware middleboxes, NFs are faster to upgrade by adding new functionality and capabilities as the network requirements change.

3. Simplifying deployment of richer policies to improve network management, access control and security, by routing the packet from a sequence of NFs. This traversal of a packet from multiple NFs to implement a policy is also called “service chaining”.

4. Reducing the deployment overhead by running NFs in virtual environment over generic compute resources. This enables efficient resource usage by sharing the underlying hardware resources between different tenant.

Central to realizing these benefits of NFV is the ability to run these NFs in multi-tenant enviroment *without causing interference* to other applications. In addition to this, handling *internal NF state* during traffic redistribution is critical to enforce network wide policies. As the NF state is dynamic (it can be updated for each incoming packet) and critical (its current value determines NF actions), the relevant internal state must be made available when traffic is rerouted to a different NF instance [53, 98, 106].

Both the academia and the industry have been pushing NFV for over a half decade to replace traditional middleboxes. Several frameworks [25, 31, 48, 53, 69, 86, 93, 96, 98, 104, 106, 117] exist to provide elastic scaling, ensure fault tolerance and support service chaining. These frameworks attempt to realize the full potential of virutalization which has been envisioned by NFVs. However, these frameworks still have not been able to achieve their full potential. In this dissertation, the main focus is to look at three major challenges that are causing hurdles in the faster adoption of NFV and in achieving its full potential.

1.1 Challenges with NFV adoption

In this section, we describe three challenges associated with the adoption of NFV.

1.1.1 Isolating network based CPU

Containers have been rapidly adopted for deployment of NFs in multi-tenant environments because containers are light-weight as compared to VMs. Furthermore, NFs like memcached [6] and load balancers [122] are the fundamental part of the micro services architecture. Multiple separate instances of such NFs are deployed in the cloud as sharing these NFs between different services is hard because of the security implications. Though, containers reduces the overhead of running NFs in the multi-

tenant environments, it is crucial to ensure strong resource isolation. As containers share underlying components of the host server's operating system (OS), it is critical for the OS to provide strong resource isolation to the container's assigned resources, such as CPU, disk, network bandwidth, and memory to avoid overcharging and ensure predictable performance. Strong isolation is critical for not only NFs which are part of micro service architecture but also other NFs which use the network stack of the OS (e.g. Bro [88], Squid [20] or Prads [16]).

1.1.2 Ensuring correctness and performance for stateful chained NFs

To enforce any network policy correctly, the NF chains are required to provide *chain output equivalence* (COE): given an input packet stream, at any point in time, the collective action taken by all NF instances in an NFV chain (Figure 1.1b.) must match that taken by an hypothetical equivalent chain with infinite capacity always available single NFs (Figure 1.1a). COE must hold under dynamics: under NF instance failures/slowdowns, traffic reallocation for load balancing/elastic scaling, etc. Given that NFV is targeted for cloud and ISP deployments, COE should not come at the cost of performance. NFs' statefulness makes this a challenging problem. As most NFs maintain detailed internal state that can be updated as often as per packet and may be shared across instances. Most existing frameworks ignore shared states or imposes high overhead on state maintenance.

1.1.3 Simplifying NF modification

For an NF to work with all these proposed frameworks [25, 31, 48, 53, 69, 86, 93, 96, 98, 104, 106, 117], NF developers have to modify, or at least annotate, their code to perform custom state allocation, track updates to state, and (de)serialize state objects. Presently three factors make such

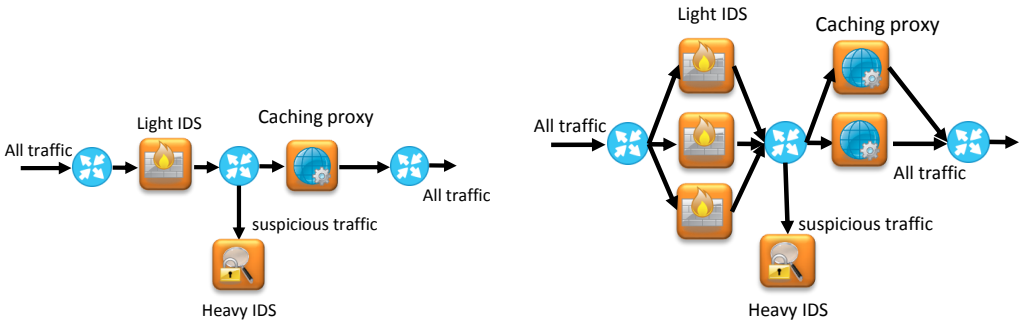


Figure 1.1: (a) logical view with infinite capacity NFs/links for COE (b) Example NFV chain with multiple instances per NF.

modifications difficult: (i) middlebox software is extremely complex, and the logic to update/create different pieces of state can be intricate; (ii) there may be 10s-100s of object types that correspond to state that needs explicit handling; and (iii) middleboxes are extremely diverse. Factors i and ii make it difficult to reason about the completeness or correctness of manual modifications. And, iii means manual techniques that apply to one middlebox may not extend to another. Our own experience in modifying middleboxes to work with OpenNF [53] underscores these problems. Making even a simple monitoring appliance PRADS [16], with 10K LOC) OpenNF-compliant took over 120 man-hours. We had to iterate over multiple code changes and corresponding unit tests to ascertain completeness of our modifications; moreover, the process we used for modifying this middlebox could not be easily adapted to other more complex ones.

1.2 Contributions

In summary, this dissertation present three novel systems which help in paving the way for the adoption of NFV.

1.2.1 Iron [74]

We show how the standard techniques used for years to provide isolation by Linux break when large amount of network traffic is handled. We developed a scheme called Iron that monitors, charges, and enforces CPU usage to process network traffic. Iron relies on a careful set of kernel instrumentations in order to obtain the cost of processing packets at a fine-grained level, while maintaining the efficiency and responsiveness of interrupt processing in the kernel. Iron integrates with the Linux scheduler to charge the appropriate container for its traffic. Charging alone cannot provide hardened isolation because processing traffic received by a container after it consumes its CPU allocation will break isolation. As a result, Iron implements a novel hardware-based scheme to drop the packets destined for a container that has exhausted its resource allocation.

1.2.2 CHC [72]

Providing support for high performance service chaining with COE is tedious, and is slowing down the wider adoption of NFV. We designed CHC to make it easier for developers to build scalable and highly available NFs. CHC relies on managing state external to NFs, but couples that with several novel caching and state update algorithms to ensure low latency and high throughput. In addition, it leverage simple metadata to ensure various correctness properties are maintained even under traffic reallocation, NF failures, as well as failures of key CHC framework components.

1.2.3 StateAlyzr [73]

To reduce manual effort and ease adoption, we develop StateAlyzr, a system that relies on *data and control-flow analysis* to automate identification of state objects that need explicit handling. Using StateAlyzr's output,

developers can easily make framework-compliant changes to arbitrary NFs. We shows that the novel state characterization algorithms of StateAlyzr provide high precision while ensuring soundness.

2

Iron

Our first step towards paving the way for NFVs is to look at containerized virtualization environment to run NFs. Containers provide an ability to share the underlying resources with minimum overhead. In order to provide consistent and reliable performance, containerized environments must ensure that NFs cannot adversely interfere with each other.

Nowadays container are widely used for the deployment of applications in the cloud such as IBM, Google, Microsoft and Amazon, in this chapter, we do not limit our scope to just NFs. Any application which is deployed in a container requires strong isolation of resources for predictable performance. When resource availability is compromised due to overprovisioning or ineffective resource isolation, latency-sensitive applications can suffer from performance degradation, which can ultimately impact revenue [22, 32, 41, 79]. In serverless computing, billing is time-based [61] and insufficient resource isolation can cause users to be needlessly overcharged. Cloud providers also rely on resource isolation to employ efficient container orchestration schemes [4, 7, 113] that enable hyper-dense container deployments per server. However, without hardened bounds on container resource consumption, providers are faced with a trade-off: either underprovision dedicated container resources on each server (and thus waste potential revenue by selling spare compute to lower priority jobs) or allow loose isolation that may hurt customer performance on their cloud.

we show containers are able to utilize more CPU than allocated by their respective cgroup when sending or receiving network traffic, effectively

breaking isolation. Modern OS kernels process traffic via interrupts, and the time spent handling interrupts is often not charged to the container sending or receiving traffic. Without accurately charging containers for network processing, the kernel cannot provide hardened resource isolation. In fact, our measurements indicate the problem can be quite severe: containers with high traffic rates can cause colocated compute-driven containers to suffer an almost **6X** slowdown. The interference can be high because kernels are responsible for a significant amount of network processing: from servicing interrupts, to protocol handling, to implementing network function virtualizations (*e.g.*, , switches, firewalls, rate limiters, etc). Modern datacenter line rates are very fast (10-100 Gbps), and studies have shown network processing can incur significant computational overhead [58, 60, 66, 95].

Interference in multi-tenant datacenters is a known problem [79, 82, 105], and researchers have developed schemes to isolate CPU [28, 33, 111] and network bandwidth [26, 56, 67, 85, 90, 91, 100, 107]. In contrast, the recent study of isolating network-based processing has been limited. Prior schemes cannot be applied to modern containerized ecosystems [57] or alter the network subsystem in such a way that interrupt processing becomes less efficient [27, 47].

We present Iron (Isolating Resource Overhead from Networking), a system that monitors, charges, and enforces CPU usage for processing network traffic. Iron relies on a careful set of kernel instrumentations in order to obtain the cost of processing packets at a fine-grained level, while maintaining the efficiency and responsiveness of interrupt processing in the kernel. Iron integrates with the Linux scheduler to charge the appropriate container for its traffic. Charging alone cannot provide hardened isolation because processing traffic received by a container after it consumes its CPU allocation will break isolation. As a result, Iron implements a novel hardware-based scheme to drop the packets destined for a container that

has exhausted its resource allocation.

Providing isolation is challenging in containerized systems for many reasons. A container's traffic traverses the whole network stack on the server OS and thus charging should be performed in a fine-grained manner to capture variations in processing different types of packets. A given solution must be computationally light-weight because linerate per-packet operations are prone to high overhead and keeping state across cores can lead to inefficient locking. Finally, limiting interference due to packet receptions is difficult because administrators may not have control over traffic sources.

Iron addresses these challenges to effectively enforce isolation for network-based processing. In short, the contributions are as follows:

- A case study that shows the computational burden of processing network traffic can be quite high. Current cgroup scheduling mechanisms do not account this burden properly, which can cause an 6X slowdown for some container workloads.

- A system called Iron to provide hardened isolation. Iron consists of a charging mechanism that integrates with the Linux cgroup scheduler in order to ensure containers are properly charged or credited for network-based processing. Iron also provides a novel packet dropping mechanism to limit the effect, with minimal overhead, of a noisy neighbor that has exhausted its resource allocation.

- An evaluation showing MapReduce jobs can experience over 50% slowdown competing with trace-driven network loads and compute-driven jobs can experience a 6X slowdown in controlled settings. Iron effectively isolates and enforces network-based processing to reduce these slowdowns to less than 5%.

2.1 Background and Motivation

This section first describes the *interference problem*: how the network traffic of one container can interfere with CPU allocated to another container. Afterwards, we place Iron in the context of past solutions, and finally empirically examine the impact of interference.

2.1.1 Network traffic breaks isolation

Sending or receiving traffic can allow a container to obtain more CPU utilization than allowed by its cgroup. In short, this occurs because the Linux scheduler does not properly account for time spent servicing interrupts for network traffic. A brief background on Linux container scheduling, Linux interrupt handling, and kernel packet processing follows.

Linux container scheduling Cgroups can limit the amount of CPU allocated to a container by defining how long a container can run (quota) over a certain time period. At a high-level, the scheduler keeps a runtime variable that accrues how long the container has run within the current period. When the total runtime of a container reaches its allocated quota, the container is throttled. At the end of a period, the container's runtime can be recharged to its quota. The scheduler is discussed in [111].

Interrupt handling in Linux Linux limits interrupt overhead by servicing interrupts in two parts: a top half (*i.e.*, , hardware interrupts) and bottom half (*i.e.*, , software interrupts). A hardware interrupt can occur at any time, regardless of which container or process is running. The top half is designed to be light-weight, and it only performs the critical actions necessary to service an interrupt. For example, the top half will acknowledge the hardware's interrupt and may directly interface with the device. The top half then schedules the bottom half to execute (i.e., raises a software interrupt). The bottom half is responsible for actions that can be delayed without affecting the performance of the kernel or I/O device. Networking

in Linux typically employs *softirqs* (a type of software interrupt) to implement the bottom half. Softirqs are used to transmit deferred transmissions, manage packet data structures, and navigate received packets through the network stack.

The way in which Linux handles softirqs directly leads to the interference problem. Software interrupts are checked at the end of hardware interrupt processing or whenever the kernel reenables softirq processing. Software interrupts run in *process context*. That is, whichever unlucky process is running will have to use its scheduled time to service the softirq. Here, isolation is broken when a container has to use its own CPU time to process another container's network traffic.

The kernel tries to minimize softirq handling in process context, so as a result, it allows the softirq handler to run for a certain time or budgeted amount of packets. When the budget is exceeded, the softirq code stops executing and schedules ksoftirqd to run. Ksoftirqd is a kernel thread (it does not run in process context) that services the remaining softirqs. There is one ksoftirqd thread per processor. Because ksoftirqd is a kernel thread, the time it spends processing packets is not charged to any container. This breaks isolation by limiting the amount of available time to schedule other containers or allowing a container that has already exhausted its cgroup quota to obtain more processing resources.

Kernel packet processing Consider a "normal" packet transmission in Linux: a packet traverses the kernel from its socket to the NIC. Although this traversal is done in the kernel, it is performed in process context, and hence the time spent sending a packet is charged to the correct container. There are, however, two cases in which isolation can break on the sender. First, when the NIC finishes a packet transmission, it schedules an interrupt to free packet resources. This work is done in softirq context, and hence may be charged to a container that did not send the traffic. The second case arises when there is buffering along the stack, which can

commonly occur with TCP (for congestion control and reliability) or with traffic shaping (in qdisc [8]). The packet is taken from the socket to the buffer in process context. Upon buffering the packet, however, the kernel system call exits. Software interrupts are then responsible for dequeueing the packet from its buffer and moving it further down the network stack. As before, this breaks isolation when softirqs are handled by ksoftirqd or charged to a container that didn't send the traffic to begin with.

Receiving packets incurs higher softirq overhead than sending packets. Under reception, packets are moved from the ring buffer all the way to the application socket in softirq context. This traversal may require interfacing with multiple protocol handlers (IP, TCP), NFVs, NIC offloads (GRO [36]), or even sending new packets (TCP ACKs or ICMP messages). In summary, the whole of the receive chain is performed in softirq context and therefore a substantial amount of time may not be charged to the correct container.

2.1.2 Putting Iron in context

Previous works can mitigate the interference problem by designing new abstractions to account for container resource consumption or redesigning the OS. Below, Iron's contributions are put in context.

System container abstraction In a seminal paper Banga proposed resource containers [27], an abstraction to capture and charge system resources in use by a particular activity. The work extends Lazy Receiver Processing (LRP) [47]. When a process is scheduled, a receive system call lazily invokes protocol processing in the kernel, and thus time spent processing packets is correctly charged to a process. This approach is inefficient for TCP because at most one window of data can be consumed between successive system calls [47], and therefore LRP employs a per-socket thread that is associated with the receiving process to perform asynchronous protocol processing such that CPU consumption is charged

appropriately.

Although LRP solves the accounting problem, the following issues must be considered. First, as the name implies, LRP only handles receiving traffic and cannot fully capture the overheads of sending traffic. Second, LRP requires a per-socket thread to perform asynchronous TCP processing[1]. Maintaining extra threads leads to additional context switching, which can incur significant overhead for processing large amounts of flows [66]. Third, the scheduler must be made aware of, and potentially prioritize, threads with outstanding protocol processing otherwise TCP can suffer from increased latencies and even drops while it waits for its socket's thread to be scheduled. A similar notion of per-thread softirq processing was proposed in the Linux Real-Time kernel, but ultimately dropped because it increases configuration complexity and reduces performance [55].

Iron explicitly addresses the above concerns. First, Iron correctly accounts for transmission overheads. Second, Iron seamlessly integrates with interrupt processing in Linux to maintain efficiency and responsiveness. In Linux, all of a core's traffic is processed by that core's softirq handler. Processing interrupts in this shared manner, rather than in a per-thread manner, maintains efficiency by minimizing context switching. Additionally, by servicing hardware interrupts in process context, network protocol processing is performed in a responsive fashion. Linux's design, however, directly leads to the interference problem. Therefore, a major contribution of our work is showing accurate accounting for network processing is possible even when interrupt handling is performed in a shared manner.

Redesigning the operating system Library OSes [51, 78, 89, 92] redesign the OS by moving network protocol processing from the kernel to application libraries. In these schemes, packet demultiplexing is performed at the

[1]Banga's paper uses a per-process asynchronous thread

lowest level of the network stack: typically the NIC directly copies packets to buffers shared with applications. Since applications process packets from their buffers directly, network-based processing is correctly charged.

Library OSes have numerous practical concerns, however. First, these works face similar challenges as LRP with threaded protocol processing. Second, explicitly removing network processing from the kernel can make management more difficult. In multi-tenant datacenters, servers can host services such as rate limiting, virtual networking, billing, traffic engineering, health monitoring, and security. In the library OS model, admin-defined network processing must be performed in the NIC or in user-level software. Neither approach is ideal. Application libraries linked by developers make it difficult for admins to insert policies and functionalities at the host. For example, this may limit an admin's ability to perform traffic shaping or simply configure a TCP stack. Furthermore, porting network services to user-level applications on the host reintroduces network processing overheads that must be charged to tenants in order to reduce interference. NIC-based techniques can cost more (upgrade all hosts), scale more poorly in the number of flows and services, and be less flexible and harder to configure than software. And while NICs are becoming more flexible [123], it is likely network management will be dictated by a combination of admin-controlled software and hardware in the future. As such, schemes like Iron can help track and enforce software-based network processing. Finally, adapting Library OSes to support multi-tenancy and replacing currently deployed ecosystems can have a high barrier to entry for providers and customers.

Contributions summary Iron differs from prior art in that it easily integrates with Linux interrupt handling to account for network-based processing in the kernel. In addition, Iron implements novel enforcement schemes to provide hardened isolation in a low-overhead manner. Iron is designed to be practical with a low barrier to entry. Finally, a measurement study

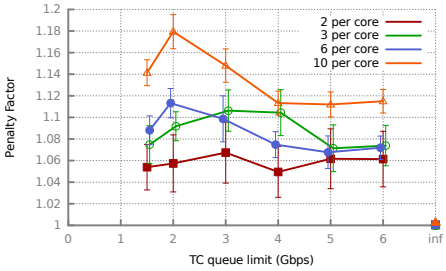


Figure 2.1: Penalty factor of UDP senders.

details the interference problem in modern containerized environments.

2.1.3 Impact of network traffic

In this section, a set of controlled experiments quantifies the impact of both UDP and TCP network processing on isolation in containerized environments.

Methodology In each experiment, n containers are allocated per core, where n varies from 2, 3, 6, or 10. Each container is configured to obtain an equal share of the core (*i.e.*, , $\mathtt{quota} = \mathtt{period}/n$). This allocation is replicated over all cores. NICs are 25 Gbps, and Section 2.3 further details methodology. One container per core, denoted the *victim*, runs a CPU-intensive sysbench workload [76]. The time to complete each victim's workload is measured under two scenarios. In the first scenario all non-victim containers, henceforth denoted *interferers*, also run sysbench. This serves as a baseline case. In the second scenario, the interferers run a simple network flooding application that sends as many back-to-back packets as possible. The victim's completion time is measured under both scenarios, and a *penalty factor* indicates the fraction of time the victim's workload takes when competing with traffic versus competing with sysbench. Penalty factors greater than one indicate isolation is broken because traffic is impacting the victim in an adverse way.

For the reception tests, containers are allocated on a single core and all NIC interrupts are serviced on the same core as the containers. This ensures cores without containers will not process any traffic. As before, the victim container runs sysbench, but the interferers now run a simple receiver. A multi-threaded sender varies its rate to the core, using 1400 byte packets and dividing flows evenly amongst the receivers. All results are averaged over 10 runs.

UDP senders These results show the impact when the interfering containers flood 1400 byte UDP traffic. Studies have shown rate limiters can increase computational overhead [95], so the penalty factor is measured when no rate limiters are configured and also when hierarchical token bucket (HTB) [8] is deployed for traffic shaping.

Figure 2.1 presents the results. In this figure, lines denote how many containers are allocated on a core. The x-axis denotes the rate limit imposed on a core and the y-axis indicates the penalty factor. With n containers per core, each container receives $\frac{1}{n}^{th}$ of the bandwidth allocated to the core, regardless of the workload. The right-most point labeled “inf" is when no rate limiter is configured. We note the following trends. First, there is no penalty factor with no rate limiting because the application demands are lower than the link bandwidth, so there is no queuing at the NIC. Second, rate limiting causes penalty factors as high as 1.18. The summed application demands can be higher than the imposed rate limit on each core, which means packets are queued in the rate limiter. Softirq handling interferes with the processing time of the victims, leading to high penalty factors. Third, HTB experiences a relatively higher penalty for 1-3 Gbps. When rate limits are 4 Gbps and above, the rate limiter does not shape traffic because senders are CPU-bound and they cannot generate more than 4 Gbps of traffic demand. Isolation still breaks because the rate limiters must maintain state and perform locking (this overhead was also witnessed in [68]). For rates below 4 Gbps, senders generate more traffic

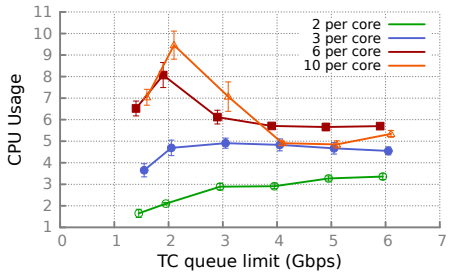
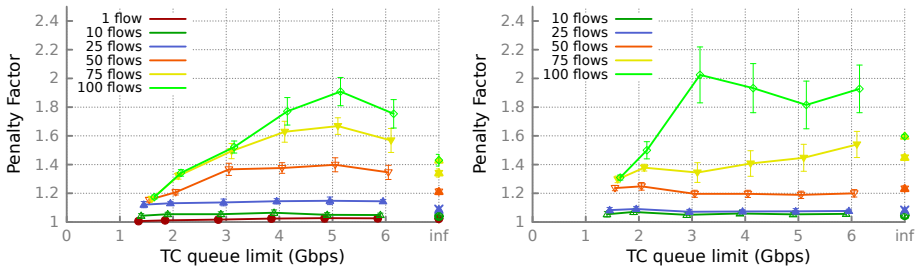


Figure 2.2: ksoftirqd overhead with UDP senders.



(a) 2 containers per core

(b) 10 containers per core

Figure 2.3: Penalty factor of victims with TCP senders.

than the enforced rate and higher overheads occur.

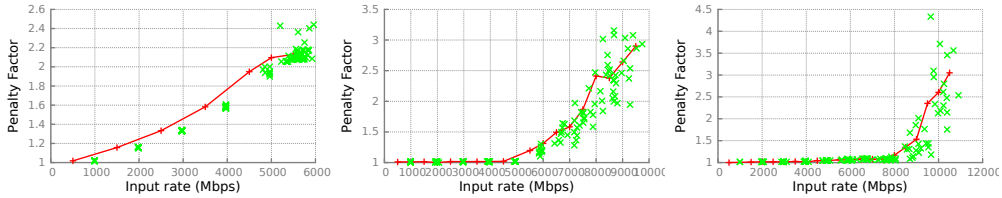
Figure 2.2 shows the CPU usage of ksoftirqd (on core 0) for the HTB experiment in Figure 2.1. The trends roughly correspond to the penalty factor overhead. The time spent in ksoftrirqd is not attributed to any process, which means ksoftirqd takes time on the core that cannot be issued to other containers. This increases the time it takes for the victim workload to complete. To understand the remaining penalty, we instrumented a run with perf [17]. With 10 containers per core and 2 Gbps rate limit, the victim spent 6.99% of its scheduled time servicing softirqs, even though it had no traffic to send.

TCP senders Figure 2.3 shows TCP sender performance for 2 and 10 containers per core. Different from the UDP results, the number of flows

per core is varied, and flows are divided equally amongst all sending containers on a core. We note the following trends. First, TCP overheads are higher than UDP overheads– in the worst case, the overhead can be as high as 1.95X. TCP overheads are higher because TCP senders receive packets, *i.e.*, , ACKs, and also buffer packets at the TCP layer. Both ACK processing and pushing buffered packets to the NIC are completed via softirqs. Therefore, the no rate limit cases have higher overhead in TCP than UDP. The second interesting trend is overheads increase as the number of flows increase. This occurs for two reasons. First, the number of TCP ACKs increase with flows, and in general, there exists more protocol processing for more flows. Second, a single TCP flow can adapt to the rate limit enforced upon it, but multiple flows create burstier traffic patterns that increase queuing at the rate limiter.

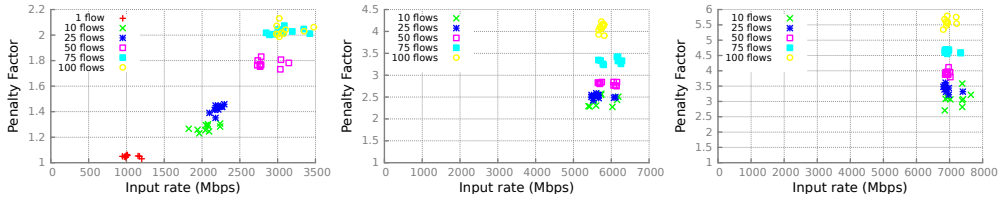
UDP receivers Figure 2.4 shows the UDP receiver results. Ten containers are allocated on the core, and the number of receivers varies. If i containers receive UDP traffic, then $10 - i$ containers run sysbench. The sender increases its sending rate from 1 Gbps to 12 Gbps at 1 Gbps increments. For each sending rate, 10 trials are run. Each green dot represents the result of a trial. The x-axis indicates the input rate to the core, which may be different from the sending rate due to drops. The red line averages the penalty factor in 500 Mbps buckets and is provided for reference only. We varied the number of receivers from 1 to 9, but only show 1, 5, and 9 receivers in the interest of space.

We note the following trends. First, the penalty factor for receiving UDP is higher than sending UDP. Packets traverse the whole network stack in softirq context and therefore overheads are larger. Next, as more of the core is allocated to receive (as i increases), the rate at which the server can process traffic increases. As the rate of incoming traffic increases, so does the penalty factor. Under high levels of traffic, the overheads from softirqs cause the victim to take almost 4.5X longer.



(a) 1 receiver (b) 5 receivers (c) 9 receivers

Figure 2.4: Penalty factor when there are 10 containers on 1 core. $i = 1, 5, 9$ of the containers are UDP receiver.



(a) 1 receiver (b) 5 receivers (c) 9 receivers

Figure 2.5: Penalty factor when there are 10 containers on a 1 core. $i = 1, 5, 9$ of the containers are TCP receivers.

TCP receivers Figure 2.5 shows the results when the interfering containers receive TCP traffic. Different from the UDP experiment, TCP senders are configured to send as much as they can. TCP will naturally adapt its rate when drops occur (from congestion control) or when receive buffers fill (from flow control). As before, the penalty factor increases as the input rate increases and also when the number of flows increase. In the worst case, interference from TCP traffic can cause the victim to take almost six times longer. To further understand this overhead, we instrument sysbench with perf for nine TCP receivers and 100 flows. Here, ksoftirqd used 54% of the core and sysbench spent 37% of its time servicing softirqs. This indicates that isolation techniques must capture softirq overhead in both ksoftirqd and process context.

2.2 Design

This section details Iron's design. At a high-level, Iron first *accounts* for time spent sending and receiving packets in softirq context. After obtaining packet costs, Iron integrates with the Linux scheduler to charge or credit containers for their softirq processing. When a container's runtime is exhausted, Iron *enforces* hardened isolation by throttling containers and dropping incoming packets via a novel hardware-based method.

2.2.1 Accounting

This section outlines how to obtain per-packet costs in order to ensure accounting is accurate. First, receiver-based accounting is detailed, followed by sender-based accounting. Afterwards, we describe how to assign packets to containers and the state used for accounting.

Receiver-based accounting In Linux, packets traverse the network stack through a series of nested function calls (see Figure 2.6a). For example, the IP handler of a given packet will directly call the transport handler. Therefore, a function low in the call stack can obtain the time spent processing a packet by subtracting the time the function starts from the time the function ends. Iron instruments netif_receive_skb to obtain per-packet costs because it is the first function that handles individual packets outside the driver, regardless of their transport protocol[2].

Obtaining the time difference is nontrivial because the kernel is preemptable and functions in the call tree can be interrupted at any time. To ensure only the time spent processing packets is captured, Iron relies on scheduler data. The scheduler keeps the cumulative execution time a thread has been running (*cumtime*), as well as the time a thread was last swapped in (*swaptime*). Coupled with the local clock (*now*), the start and end times can be calculated as: $time = cumtime + (now - swaptime)$.

[2]TCP first traverses GRO, but we instrument here for uniformity

Besides the per-packet cost, there is also a fixed cost associated with processing traffic. That is, there are overheads for entering the function that processes hardware interrupts (do_IRQ), processing softirqs, and performing skb garbage collection. In Iron, these overheads are lumped together and assigned to packet costs in a weighted fashion. In Linux, six types of softirqs are processed by the softirq handler (do_softirq): HI, TX, RX, TIMER, SCSI, and TASKLET. For each interrupt, we obtain the total do_IRQ cost, denoted H, and the cost for processing each specific softirq (denoted S_{HI}, S_{TX}, etc). Note software interrupts are processed at the end of a hardware interrupt, so $H > \sum_i S_i$. Then the overhead associated with processing an interrupt is defined as: $O = H - \sum_i S_i$ and the fair share of the receive overhead within that interrupt is: $O_{RX} = O\frac{S_{RX}}{\sum_i S_i}$. Last, O_{RX} is evenly split amongst packets processed in a given do_softirq iteration to obtain a fixed charge that is added to each packet.

Finally, we note this scheme is effective in capturing TCP overhead. That is, Iron gracefully handles TCP ACK overhead and TCP buffering. A TCP flow is handled within a single thread, so when a data packet is received, the thread directly calls a function to send an ACK. When the ACK function returns, the received data packet continues to be processed as normal. Therefore, ACK overhead is captured by our start and end timestamps. Buffering is also handled correctly. Say packet $i-1$ is lost and therefore received packet i is buffered. The system call exits at that point, and netif_receive_skb moves to the next packet in its list. When retransmitted packet $i-1$ is received the gap in sequence numbers is filled, and TCP will push up all of the packets to the socket. This means correct charging occurs because the cost of moving packet i from the buffer to the socket is captured in the cost of the retransmitted packet $i-1$.

Sender-based accounting The transmit chain is different from the receive chain. When sending packets, the kernel has to obtain a lock on the appropriate NIC queue. Obtaining a lock on a per-packet basis has high

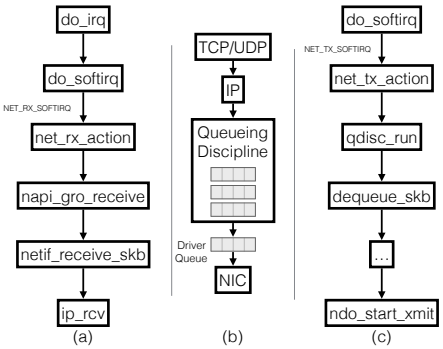


Figure 2.6: Overview of network stack in Linux.

overhead, so as a result, packets are often batched for transmission [38]. Therefore, Iron measures the cost of sending a batch and then charges each packet within the batch for an equal share of the batch cost. The do_softirq function calls net_tx_action to process transmit softirqs (refer to Figure 2.6b,c). Then net_tx_action calls into the qdisc layer to retrieve packets. Multiple qdisc queues can be dequeued, and each queue may return multiple packets. As a result, a linked list of skbs is eventually created and sent to the NIC. Similar to the receiver instrumentation, net_tx_action obtains a start and end time for sending the batch. As with the sender, O_{TX} is obtained to split the transmission's fixed cost overheads. This information is processed per core because HTB is work conserving and may dequeue a packet on a different core than it was enqueued on.

Mapping packets to containers Iron must identify the container a given packet belongs to. On the sender, an skb is associated with its cgroup when enqueued in the qdisc layer. This allows us to determine how many packets each container sends within each batch. On the receiver, Iron maintains a hash table on IP addresses that is filled when copying packets to a socket.

Accounting data structures As described in Section 2.2.2, the time spent processing packets needs to be associated to the appropriate cgroup.

In Iron, each process maintains a local (per-core) list of the packets it processed in softirq context and their individual costs. The per-process structures are eventually merged into a global per-cgroup structure. Iron does this in a way that does not increase locking by only merging state when the scheduler obtains a global lock. The per-cgroup structure maintains a variable (gained) that indicates if a cgroup should be credited for network processing.

2.2.2 Enforcement

This subsection shows how isolation is enforced. Isolation is achieved by integrating accounting data with bandwidth control in Linux's CFS scheduler [111] and dropping packets when a container becomes throttled.

Scheduler integration The CFS scheduler implements bandwidth control for cgroups via a hybrid scheme that keeps both local (*i.e.*, , per core) and global state. Containers are allowed to run for a given quota within a period. The scheduler minimizes locking overhead by updating local state on a fine-grained level and interfacing with global state on a coarse-grained level. At the global level a runtime variable is set to quota at the beginning of a period. The scheduler subtracts a slice from runtime and allocates it to a local core. The runtime continues to be decremented until either it reaches zero or the period ends. Regardless, at the end of a period runtime is refilled to the quota.

On the local level, a rt_remain variable is assigned the slice intervals pulled from the global runtime. The scheduler decrements rt_remain as the task within a cgroup consumes CPU. When rt_remain hits zero, the scheduler tries to obtain a slice from the global pool. If successful, rt_remain is recharged with a slice and the task within the cgroup can continue to run. If the global pool is exhausted, then the local cgroup gets *throttled* and its tasks are no longer scheduled until the period ends.

Iron's modifications to the global scheduler are presented in Algorithm 2.7. A global variable gained is added to track the time a container should get back because it processed another container's softirqs. Line 2 adds gained to runtime. Next, runtime is reset to 0 if the container didn't use its previous allocation because it was limited by its demand (lines 5-7). This preserves a CFS policy that disallows unused cycles to be accumulated for use in subsequent periods. Last, line 8 refills runtime. The old runtime is positive when interference prevents a container from using its allocation and negative when a container exceeds its allocated time by sending or receiving too much traffic.

Iron's local algorithm is listed in Algorithm 2.8. The scheduler invokes this function when rt_remain $\leqslant 0$ and after obtaining appropriate locks. The cpuusage variable is added to maintain local accounting: positive values indicate the container needs to be charged for unaccounted networking cycles and negative values indicate the container needs a credit for work it did on another container's behalf. Lines 3-9 cover when a container is to be charged, trying to take from gained first if possible. Lines 10-12 cover the case when a container is to be credited, so gained is increased. Lines 14-18 cover a corner case where the runtime may be exhausted, but some credit was accrued and can be used. Lines 19-22 are unchanged: they ensure the container has global runtime left to use. If not, then the amount variable will remain 0. Line 23 updates the new rt_remain by amount.

Dropping excess packets While scheduler-based enforcement improves isolation, packets may still need to be dropped so a throttled container cannot accrue more network-based processing. The current implementation of Iron does not explicitly drop packets at the sender because throttled containers already cannot generate more outgoing traffic. There exists a corner case when a container has some runtime left and sends a large burst of packets. Currently, the scheduler will charge this overage on the next quota refill. We did implement a *proactive* charging scheme that estimates

```
1: if gained > 0 then
2:   runtime ← runtime + gained
3:   gained ← 0
4: end if
5: if cgroup_idled() and runtime > 0 then
6:   runtime ← 0
7: end if
8: runtime ← quota + runtime
9: set_timer(now + period)
```

Figure 2.7: Global runtime refill at period's end

```
1: amount ← 0
2: min_amount ← slice - rt_remain
3: if cpuusage > 0 then
4:   if cpuusage > gained then
5:     runtime ← runtime - (cpuusage - gained)
6:     gained ← 0
7:   else
8:     gained ← gained - cpususage
9:   end if
10: else
11:   gained ← gained + abs(cpuusage)
12: end if
13: cpuusage ← 0
14: if runtime = 0 and gained > 0 then
15:   refill ← min(min_amount, gained)
16:   runtime ← refill
17:   gained ← gained - refill
18: end if
19: if runtime > 0 then
20:   amount ← min(runtime, min_amount)
21:   runtime ← runtime - amount
22: end if
23: rt_remain ← rt_remain + amount
```

Figure 2.8: Local runtime refill

the cost of packet transmission, charges it up-front, and drops packets if necessary. This scheme didn't substantially affect performance, however.

Dropping the receiver's excess packets is more important because a throttled receiver may continue to receive network traffic, hence breaking isolation. Iron implements a novel hardware-based dropping mechanism that integrates with current architectures. Today, NICs can insert incoming packets into multiple queues. Each queue has its own interrupt that can be assigned to specified cores. To improve isolation, packets are steered to the core in which their container runs via advanced receive flow steering [62] (FlexNIC [71] also works). Upon reception, the NIC DMAs a packet to a ring buffer in shared memory. Then, the NIC generates an IRQ for the queue, which triggers the interrupt handler in the driver. Modern systems manage network interrupts with NAPI [13]. Upon receiving a new packet, NAPI disables hardware interrupts and notifies the OS to schedule a polling method to retrieve packets. Meanwhile, additionally received packets are simply put in the ring buffer by the NIC. When the kernel polls the NIC, it removes as many packets from the ring buffer as possible, bounded by a budget. NAPI polling exits and interrupt-driven reception is resumed when the number of packets removed is less than the budget.

Our hardware-based dropping mechanism works as follows. First, assume the NIC has one queue per container. Iron augments the NAPI queue data structure with a map from a queue to its container (*i.e.*, , task_group). When the scheduler throttles a container, it modifies a boolean in task_group. Different from default NAPI behavior, Iron does not poll packets from queues whose containers are throttled. From the kernel's point of view, the queue is stripped from the polling list so that it isn't constantly repolled. From the NIC's point of view, the kernel is not polling packets from the queue, so it stays in polling mode and keeps hardware interrupts disabled. If new packets are received, they are simply

inserted into the ring buffer. This technique effectively mitigates receiving overhead because the kernel is not being interrupted or required to do any work on behalf of the throttled container. When the scheduler unthrottles a container, it resets its boolean and schedules a softirq to process any packets that may be enqueued.

As a slight optimization, Iron can also drop packets before a container is throttled. That is, if a container is receiving high amounts of traffic and the container is within T% of its quota, packets can be dropped. This allows the container to use some of its remaining runtime to stop a flood of incoming packets.

The hardware-based dropping technique is effective when there are a large number of queues per NIC. Even though NICs are increasingly outfitted with extra queues (*e.g.*, , Solarflare SFN8500-series NICs have 2048 receive queues), in practice the number of queues may not equal the number of containers. Iron can allocate a fixed number of queues per core and then dynamically map problematic containers onto their own queue. Containers without heavy traffic can incur a software-based drop by augmenting the __netif_receive_skb function early in the softirq processing stack. This dynamic allocation scheme draws inspiration from SENIC [95], which uses a similar approach to scale NIC-based rate limiters. Alternatively, containers can be mapped to queues based on prepurchased bandwidth allocations.

2.3 Evaluation

This section evaluates the effectiveness of Iron. First, a set of macrobenchmarks show Iron isolates controlled and realistic workloads. Then, a set of microbenchmarks investigates Iron's overhead and design choices.

Methodology The tests are run on Super Micro 5039MS-H8TRF servers with Intel Xeon E3-1271 CPUs. The machines have four cores, with hyper-

threading disabled and CPU frequency fixed to 3.2 Ghz. The servers are equipped with Broadcom BCM57304 NetXtreme-C 25 Gbps NICs (driver 1.2.3 and firmware 20.2.25/1.2.2). The servers run Ubuntu 16.04 LTS with Linux kernel 4.4.17. The NICs are set to 25 Gbps for UDP results and 10 Gbps for TCP results (we noticed some instability with TCP at 25 Gbps).

We use lxc to create containers and Open Virtual Switch as the virtual switch. Simple UDP and TCP sender and receiver programs create network traffic. The sysbench's CPU benchmark is used to measure the computational overhead from competing network traffic. Rate limiters are configured with default burst settings.

2.3.1 Macrobenchmarks

Sender-side experiments We run the same experiments in Section 2.1 to evaluate how well Iron isolates sender interference. Figure 2.9 shows the impact of UDP senders on sysbench. Note this experimental setup is the same as Figure 2.1. Iron obtains average penalty factors less than 1.01 for 2, 3, and 6 containers, as compared penalty factors as high as 1.11 without Iron. With 10 containers, Iron's penalty factor remains below 1.04, a significant decrease from the maximum of 1.18 without Iron.

Figure 2.10 shows the performance of Iron with TCP senders, and can be compared to Figure 2.3a. The maximum penalty factor experienced by Iron is 1.04, whereas the maximum penalty factor without Iron is 1.85. These results show Iron can effectively curtail interference from network-based processing associated with sending traffic.

Receiver-side experiments We rerun the experiments in Section 2.1 to evaluate how well Iron isolates receiver interference. Even though our NICs support more than eight receive queues, we were unable to modify the driver to expose more queues than cores. Therefore, different from Section 2.1, a single core is allocated with 8 containers, instead of 10. In

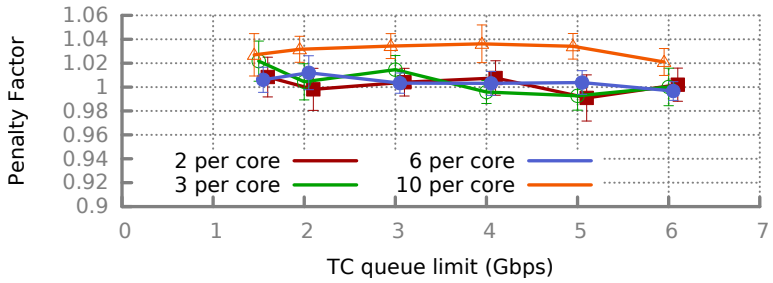


Figure 2.9: Performance penalty of victim with UDP senders. Compare to Figure 2.1.

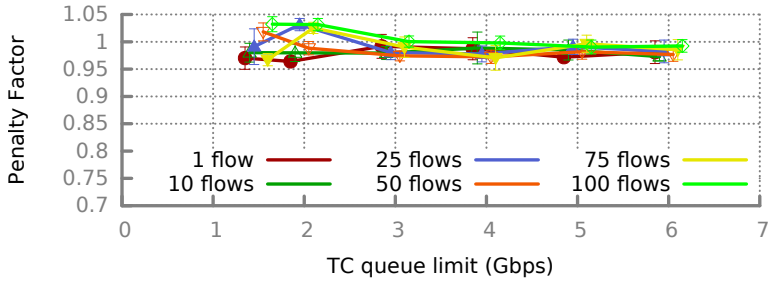


Figure 2.10: Performance penalty of victim with TCP senders. Compare to Figure 2.3a.

these experiments, the number of receiver containers varies from 1, 4, or 7. Containers that are not receivers run an interfering sysbench workload. For the UDP experiments, the hardware-based enforcing mechanism was employed, while the TCP experiments utilize our software-based enforcing mechanism.

Figure 2.11 shows the impact of UDP receivers. The x-axis shows aggregated traffic rate at the sender. This is different from the graphs in Section 2.1 because Iron drops packets when container quotas are exceeded, causing received rates to converge. Each number of receivers is indicated by a different bar color. The error bars represent the 5% and 95%. The height of the bars indicate the 25% to 75% and the red horizontal line within each bar is the median. In the previous results without Iron, penalty

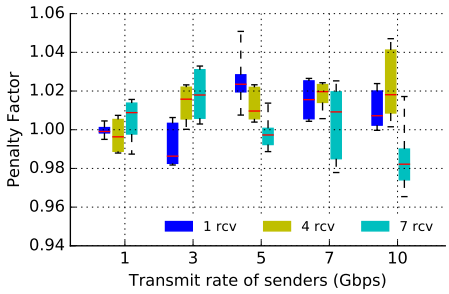


Figure 2.11: Performance penalty of victim when there are 8 containers on a core. `i` of the containers are UDP receivers.

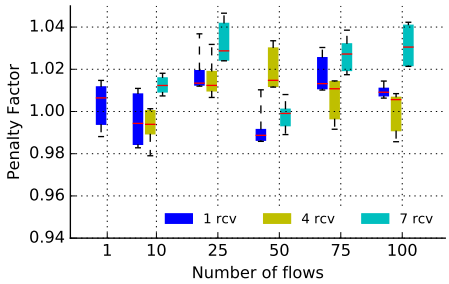
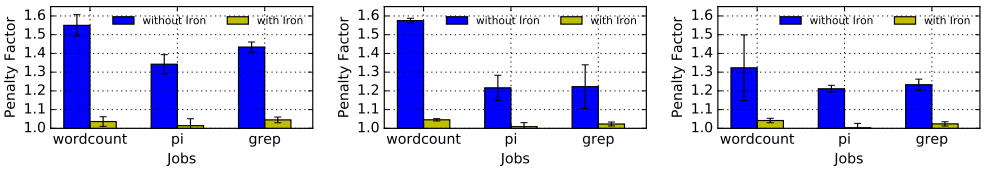


Figure 2.12: Performance penalty of victim when there are 8 containers on a core. `i` of the containers are TCP receivers.

factors ranged from maximums of 2.45 to 4.45. With Iron, the median penalty factor ranges between 0.98 and 1.02 and never exceeds 1.05. The penalty factor can be lower than 1 when Iron overestimates hard interrupt overheads.

Figure 2.12 shows when interfering containers receive TCP traffic. Unlike UDP, TCP adapts its rate when packet drops occur. Therefore, the software-based rate limiter is effective in reducing interference. In Section 2.1, the maximum penalty factor ranged from 2.2 to 6. However, with Iron, penalty factors do not exceed 1.05.

Realistic applications Here we evaluate the impact of interference on real applications. We run the experiment on a cluster of 48 containers spread

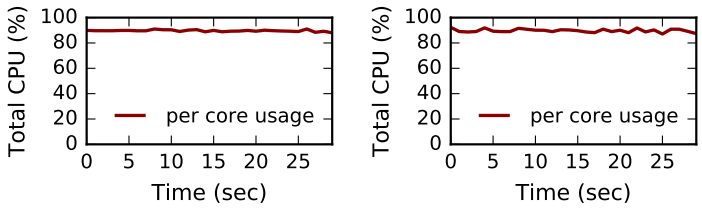


(a) with TeraSort (b) with data-mining workload (c) with web-search workload

Figure 2.13: Penalty factor when MapReduce jobs share resources with other workloads.

over 6 machines. Each machine has 8 containers (2 per core). The cluster is divided into two equal subclusters such that a container in a subcluster does not share the core with a container from the same subcluster. HTB evenly divides bandwidth between all containers on a machine.

Three MapReduce applications serve as the victims: *pi* computes the value of pi, *wordcount* counts word frequencies in a large file, and *grep* searches for a given word in a file. Three different trace-based interferers run on the other subcluster: the shuffle phase of a *TeraSort* job with a 115GB input file, a *web-search* workload [22] and a *data-mining* workload [21]. For the latter two workloads, applications maintain long-lived TCP connections to every other container in the subcluster, sequentially sending messages to a random destination with sizes distributed from each trace. Figure 2.13 shows the impact of interference on real applications. Iron obtains an average penalty factor less than 1.04 over all workloads, whereas the average penalty factor ranges from 1.21-1.57 without Iron. These results show Iron can effectively eliminate interference that arises in realistic conditions.



(a) CPU usage with senders (b) CPU usage with receivers

Figure 2.14: CPU overhead benchmarks.

2.3.2 Microbenchmarks

This subsection evaluates Iron's overhead, the usefulness of runtime packet cost calculation, and the benefits of hardware-based packet dropping.

Performance overhead To measure how accurately Iron limits CPU usage, we allocated 3 containers on a core with each container having 30% of the core. One of the containers ran sysbench, while the other two were UDP senders. Figure 2.14a shows the total CPU used by all containers over a 30 second window. On average, the consumed CPU was around 90.02%. In an ideal case, no more than 90% of the CPU should be utilized. This indicates Iron does not have high overhead in limiting cgroup bandwidth to its respective limits. We also ran the experiment with a UDP receiver, as shown in Figure 2.14b. On average Iron ensures an idle CPU of 10.07%, which again shows the effectiveness of our scheme.

Next we analyzed if Iron hurts a network-intensive workload. We instrumented a container to receive traffic and allowed it 100% of the core. Then, at the sender, we generated UDP traffic at 2 Gbps. Using mpstat, we measured the CPU consumed by the receiver. The receiver consumed 35% of the core and received traffic at 1.93 Gbps. Next, we ran the same experiment with the receiver, but this time limited the container to 35% of the core. With Iron limiting the CPU usage, the receiver received traffic at 1.90 Gbps (and used no more than 35% of the CPU). This indicates

Packet type	Average packet cost (usec)
UDP	0.706
TCP	1.670
GRE Tunnel	1.184

Table 2.15: Average packet processing cost at the receiver.

the overhead of Iron on network traffic is minimal. The slight decrease in throughput is attributed to Iron's preemptive dropping mechanism described in Section 2.2.2. We ran a similar experiment with the UDP sender and observed no degradation in traffic rate. Unlike the receiver, if a sender is out of CPU cycles it will be throttled, thus generating no extra traffic.

Packet cost variation A simple accounting mechanism may charge a fixed packet cost. This scheme is likely to be ineffective because packet processing costs vary significantly at the receiver. Table 2.15 shows the average packet cost for three different classes of traffic. TCP requires bookkeeping for flow control and reliability and results in higher costs than UDP. UDP packets encapsulated in GRE experience extra cost because those packets traverse the stack twice.

Dropping mechanism We compared the impact of software-based dropping versus hardware-based dropping. UDP traffic is generated with 8 containers on a core and 7 of those containers are configured as receivers. The sending rate is varied. As shown in Figure 2.16, both approaches mitigate interference when rates of traffic are low. However, when rates of traffic are high, the median penalty factor of the software-based rate limiter increases to 1.19, with the 95% approaching a penalty factor of 1.6. The hardware rate-limiter maintains a near-constant penalty factor, regardless of rate.

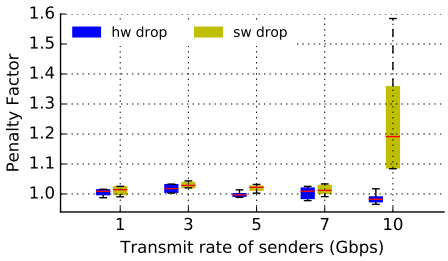


Figure 2.16: Impact of software and hardware-based packet dropping mechanisms on penalty factor for 7 receivers.

2.4 Related Work

Here we augment Section 2.1.2 to further detail prior art.

Isolation of kernel-level resources in servers Many studies have examined how colocated computation suffers from interference [79, 82, 105, 120]. As such, providing resource isolation to underlying server hardware has been a rich area of research. For example, researchers have investigated how to isolate CPU time [28, 33, 111], processor caches [50, 70], memory bandwidth [65, 118], and storage [83, 109, 115, 119]. These schemes address problems orthogonal to our work, and none can be generalized to solve the problem Iron solves.

A large class of research allocates network bandwidth over shared datacenter environments [26, 56, 67, 85, 90, 91, 100, 107]. In short, these schemes partition and isolate network bandwidth, but do nothing for network-based processing time. Therefore, these schemes are complimentary to Iron. Last, some schemes isolate dataplane and application processing on core granularity [29, 66], but do not generalize to support many containers per core nor explicitly study the interference problem.

Resource management and isolation in cloud Determining how to place computation within the cloud has received significant attention. For example, Paragon [42] schedules jobs in an interference-aware fashion to ensure

service-level objectives can be achieved. Several other schemes, such as Borg [113], Quasar [43], Heracles [79], and Retro [80] can provision, monitor, and control resource consumption in colocated environments to ensure performance is not degraded due to interference. Iron is largely complementary to these schemes. By providing hardened isolation, Iron allows resource managers to make more informed decisions, so network-heavy jobs cannot impact colocated processor-heavy jobs.

VM network-based accounting Gupta's scheme accounts for processing performed in device drivers for an individual VM [57]. The scheme measures VM-based resource consumption in the hypervisor, integrates with the scheduler to charge for usage, and limits traffic when necessary. Iron differs in many regards. Iron provides performance isolation in container-based environments, instead of VM-based environments. The difference is significant because packets consume more processing time with containers because the network stack lies within the server's kernel, and not the VM's. Gupta relies on a fixed cost to charge for packet reception and transmission, but our results show packet costs vary significantly. Furthermore, because container-based environments incur more processing overhead for traffic, it is important that received traffic is discarded efficiently when necessary. Hence, Iron contains a novel hardware-based enforcement scheme, whereas Gupta's work relies on software.

Shared NFV infrastructure Many works study how to allocate multiple NFVs and their resources on a server [45, 54, 77, 81, 112]. Similar to library OSes, NFV servers require kernel bypass for latency and control. As discussed, kernel bypass approaches cannot easily generalize to solve the interference problem in multi-tenant containerized clouds.

Real-time kernel Real-time (RT) kernels typically aren't used for multi-tenancy, but some RT OSes redesign interrupt processing in a way that could mitigate the interference problem. For example, the RT Linux kernel

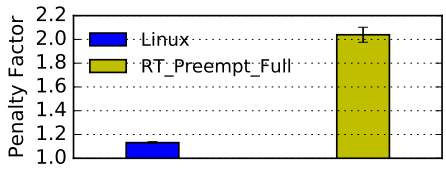


Figure 2.17: Performance penalty with RT Linux.

patches the OS so the only type of softirq served in a process's context are those which originated within that process [37]. While this patch doesn't help with receptions, it prevents a container with *no* outgoing traffic from processing interrupts from another container's outgoing traffic. To understand this solution, we ran an experiment with 2 Gbps rate limit and 6 equally-prioritized containers per core: one sysbench victim and 5 interferers that flood outgoing UDP traffic. Figure 2.17 shows the penalty factor for normal Linux and RT Linux (RT_Preempt_Full) (Iron not shown). It is surprising the penalty factor of RT Linux is significantly higher than Linux because in RT Linux the victim doesn't process interrupts in its context. Instead, interrupt processing is moved to kernel threads. The processing time used by the kernel threads reduces the time available to the victim. Additionally, RT Linux tries to minimize softirq processing latency and perf shows the victim experiences 270x more involuntary context switches as compared to normal Linux.

Finally, Zhang [121] proposes a RT OS that increases predictability by scheduling interrupts based on process priority and accounting for time spent in interrupts. Zhang's accounting scheme has up to 20% error [121], likely because it is coarse-grained in time and does not use actual, per-packet costs. Overheads in Iron are less than 5% because its accounting mechanism is immediately responsive to actual, per-packet costs. In addition, Iron comprehensively studies the interference problem and introduces novel enforcement schemes.

2.5 Conclusion

We present Iron, a system that provides hardened isolation for network-based processing in containerized environments. Network-based processing can have significant overhead in modern systems, and our case study shows that a container running a CPU-intensive task may suffer up to a 6X slowdown when colocated with a container running a network-intensive task. Iron enforces isolation by first instrumenting a novel accounting mechanism that accurately measures the time spent processing network traffic in softirq context within the kernel. Then, Iron relies on an enforcement algorithm that integrates with the Linux scheduler to throttle containers when necessary. Throttling alone is insufficient to provide isolation, because a throttled container may still receive network traffic. Therefore, Iron contains a novel hardware-based dropping mechanism to drop packets with minimal overhead. Our scheme seamlessly integrates with modern Linux and server architectures. Finally, the evaluation shows Iron reduces overheads from network-based processing to less than 5%.

3

CHC

In chapter 2, we developed a scheme to provide strong CPU isolation in multi-tenant environments. This enables running NFs without causing any interference to other applications. In this chapter, we propose a new ground up framework, which enables network operators to use multiple NFs to implement network policies, while hiding the complexity of providing elastic scaling and ensuring fault tolerance from the NF developers and operators. Our framework supports *chain output equivalence* (COE) and high performance for NFV chains.

NFV vastly improves network management. It allows operators to implement rich security and access control policies using NF chains [1, 11, 14, 30, 48]. Operators can overcome NF failure and performance issues by spinning up additional instances, and dynamically redistributing traffic [52, 106]. To enforce policies correctly, NFV is required to provide COE. Ensuring COE is made challenging by NFs' *statefulness*. Most NFs maintain detailed internal state that could be updated as often as per packet. Some of the states may be shared across instances. For example, the IDS instances in Figure 3.1a may share *cross-flow state*, e.g., per port counters. They may also maintain *per-flow* state, e.g., bytes per flow, which is confined to within an instance.

Ensuring COE under statefulness requires that, as traffic is being processed by many instances, or being reassigned across instances, updates to state at various NFs must happen in a "correct" fashion. For example, shared state updates due to packets arriving at IDS1 must be reflected

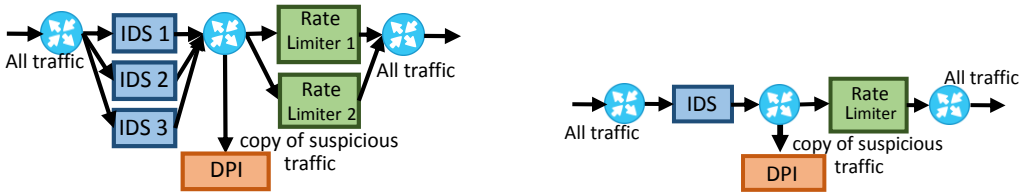


Figure 3.1: (a) Example NFV chain with many instances per NF (b) logical view with infinite capacity NFs/links for COE.

at IDS2; likewise, when reallocating a flow, say f_1, from IDS1 to 2, f_1's state should be updated due to in-flight f_1 packets arriving at both IDSes 1 and 2. Finally, *how* the state is updated can determine an NF's action. For example, the off-path Trojan detector [40] in Figure 3.2 relies on knowing the exact order in which connection attempts were made. When there is a discrepancy in the order observed w.r.t. the true order – e.g., due to intervening NFs running slow or failing – the Trojan detector can arrive at incorrect decisions, violating COE.

Many NFV frameworks exist today [31, 69, 86, 96, 98, 106, 117]. Several of them focus on managing NF state migration or updates upon traffic reallocation during scaling or failover [53, 96, 98, 106, 117]. However, they either violate COE, or suffer from poor performance (or both).

First, most systems **ignore shared state** [86, 96, 98, 106]. They assume that NFs do not use cross-flow state, or that traffic can be split across NF instances such that sharing is completely avoided. Unfortunately, neither assumption is valid; many NFs [16, 20, 23, 88] have cross-flow state, and the need for fine-grained traffic partitioning for load balancing can easily force cross-flow state sharing across instances. Because shared state is critical to NF processing, ignoring how it is updated can lead to inconsistent NF actions under dynamics, violating COE (§3.1.2).

Second, existing approaches **cannot support chain-level consistency**.

They cannot ensure that the order of updates made to an NF's state (e.g., at the Trojan detector [40] in Figure 3.2) are consistent with the input packet stream. This inability can lead to NFs arriving at incorrect decisions, e.g., missing out on detecting attacks (as is the case in Figure 3.2), violating COE. Similar issues arise in the inability to correctly suppress spurious duplicate updates observed at an NF due to recovery actions at upstream NFs (§3.1.1).

Finally, existing frameworks impose **high overhead** on state maintenance, e.g., 100s of milliseconds to move per-flow state across instances when traffic is reallocated (§3.1.2).

We present a new NFV framework, CHC ("correct, high-performance chains"), which overcomes these drawbacks. For COE, CHC uses three building blocks. CHC stores NF state in an in-memory **external state store**. This ensures that state continues to be available after NF instances' recover from failure, which is necessary for COE. Second, it maintains simple **metadata**. It adds a "root" at the entry of a chain that: (1) applies a unique logical clock to every packet, and (2) logs packets whose processing is still ongoing in the chain. At the store and NFs, CHC tracks packet clocks along with update operations each NF issues. Clocks help NFs to reason about relative packet ordering irrespective of intervening NFs' actions, and, together with datastore logs, help suppress duplicates. We develop failure recovery protocols which leverage clocks and logs to ensure correct recovery from the failure. In Appendix A.3, we prove their correctness by showing that the recovered state is same as if no failure has occurred, thereby ensuring COE.

State externalization can potentially slow down performance of state reads/writes. Thus, for performance, CHC introduces **NF-aware algorithms** for shared state management. It uses scope-awareness of state objects to partition traffic so as to minimize cross-instance shared state coordination. It leverages awareness of the state access patterns of NFs to

implement strategies for shared state caching. Because most NFs today perform a simple set of state update operations, CHC offloads operations to the state store, which commits them in the background. This speeds up shared state updates – all coordination is handled by the store which serializes the operations issued by multiple NF instances.

We built a multi-threaded C++ prototype of CHC along with four NFs. We evaluate this prototype using two campus-to-EC2 packet traces. We find that CHC's state management optimizations reduce latency overhead to **0.02μs - 0.54μs** per packet compared to traditional NFs (no state externalization). CHC failover offers **6X** better 75%-ile per packet latency than [106]. CHC is 99% faster in updating strongly consistent shared state, compared to [53]. CHC obtains per-instance throughput of 9.42Gbps – **same as maximum achievable** with standalone NFs. CHC's support for chain-wide guarantees adds little overhead, but **eliminates false positives/negatives** seen when using certain security NFs in existing NFV frameworks. Thus, CHC is the only framework to support COE, and it does so at state-of-the-art performance.

3.1 Motivation

NFV allows operators to connect NFs together in chains, where each type of NF can use multiple instances to process input traffic demand. Use of software NFs and SDN [94] means that when incoming traffic load spikes, or processing is unbalanced across instances, operators can scale up by adding NF instances and/or reallocate flow processing across instances. Furthermore, hot-standby NFs can be used to continue packet processing when an instance crashes. Due to these benefits, cloud providers and ISPs are increasingly considering deploying NFV in their networks [12].

3.1.1 Key Requirements for COE

NFV chains are central to security and compliance policies, they must always operate correctly, i.e., ensure COE (§3). Ensuring COE is challenging: (1) NFs are stateful; they maintain state objects for individual and group of flows. These state objects may be updated on every packet and the value of these state objects may be used to determine the action on the packet. This requires support for fine gained NF state management. (2) In addition to this, COE also require that the per-NF and chain-wide state updates are consistent with the input packet stream. (3) Since chaining may create a dependency between the action taken in upstream instances and its downstream instances, it is important that the impact of a local action taken for failure recovery should be isolated from the rest of the chain. These challenges naturally map to three classes of requirements for supporting COE:

State Access: The processing of each packet requires access to up-to-date state; thus, the following requirement are necessary to ensure COE under dynamics:

• (R1) *State availability:* When an NF instance fails, all state it has built up internally disappears. For a failover instance to take over packet processing it needs access to the state that the failed instance maintained just prior to crashing.

• (R2) *Safe cross-instance state transfers:* When traffic is reallocated across NF instances to rebalance load, the state corresponding to the reallocated traffic (which exists at the old instance where traffic was being processed) must be made available at the reallocated traffic's new location.

Consistency: Action taken by a given NF instance may depend on shared-state updates made by other instances of the same NF, or state actions at upstream NFs in the chain. Ensuring that said NF instances' actions adhere to COE boils down to following requirements:

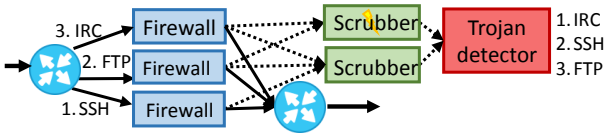


Figure 3.2: Illustrating violation of chain-wide ordering.

• (R3) *Consistent shared state:* Depending on the nature of an NF's state, it may not be possible to completely avoid sharing a subset of it across instances, no matter how traffic is partitioned (e.g., port counts at the IDSes in Figure 3.1a). Such state needs to be kept consistent across the instances that are sharing; that is, writes/updates made locally to shared state by different instances should be executed at all other instances sharing the state in the same global order. Otherwise, instances may end up with different views of shared state leading to inconsistent and hence incorrect actions.

• (R4) *Chain-wide ordering:* Some NFs rely on knowing the order in which traffic entered the network. Consider Figure 3.2. The off-path Trojan detector [40] works on a copy of traffic and identifies a Trojan by looking for this sequence of steps: (1) open an SSH connection; (2) download HTML, ZIP, and EXE files over an FTP connection; (3) generate IRC activity. When a Trojan is detected, the network blocks the relevant external host. A different order does not necessarily indicate a Trojan. It is crucial that the Trojan detector be able to reason about the true arrival order as seen at traffic input.

In Figure 3.2, either due to one of the scrubbers being slowed down due to resource contention or recovering from failure [106], the order of connections seen at the Trojan detector may differ from that in the traffic arriving at the input switch. Thus, the Trojan detector can either incorrectly mark Trojan traffic as benign, or vice versa. When multiple instances of the Trojan detector are used, the problem is compounded because it might not be possible to partition traffic such that all three flows are processed

at one instance.

• (R5) *Duplicate suppression:* In order to manage straggler NFs, NFV frameworks can (a) deploy clones initialized with the state of a slow NF instance; (b) use packet replay to bring the clone up to speed with the straggler's state since state initialization; and (c) replicate packets to the straggler and clone (§3.4.3). Depending on when the clone's state was initialized, replay can lead to duplicate state updates at the straggler. Also, the original and clone instances will then both generate duplicate output traffic. Unless such duplicate updates and traffic are suppressed, the actions of the straggler and of downstream NFs can be impacted (spurious duplicates may trigger an anomaly). The need for duplicate suppression also arises during fault recovery (§3.4.4).

Isolation: NFs in a chain should not be impacted by failure recovery of NFs upstream from them. Specifically:

• (R6) *Safe chain-wide recovery:* When NF failures occur and recovery takes place, it is important that the state at each NF in the chain subsequent to recovery have the same value as in the no-failure case. In other words, actions taken during recovery should not impact the processing, state, or decisions of NFs upstream or downstream from the recovering NF — we will exemplify this shortly when we describe failings of existing systems in meeting this requirement.

The network today already reorders or drops packets. Our goal is to ensure that NF replication, chaining, and traffic reallocation together do not induce artificial ordering or loss on top of network-induced issues. This is particularly crucial for many important off-path NFs (e.g., DPI engines and exfiltration checkers) which can be thwarted by artificially induced reordering or loss.

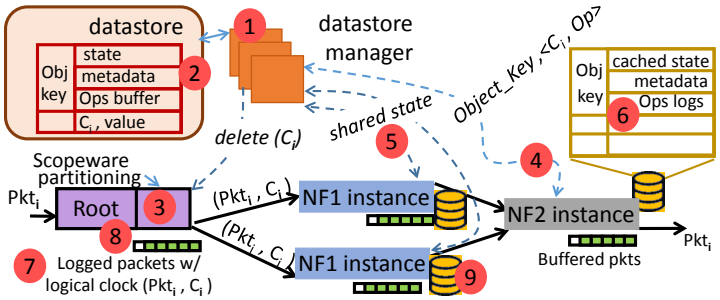


Figure 3.3: CHC architecture

3.1.2 Related work, and Our Contributions

A variety of NFV frameworks exist today [25, 31, 48, 53, 69, 86, 93, 96, 98, 104, 106, 117]. We review their drawbacks below.

Incomplete support for correctness requirements: Most existing frameworks focus on handling requirements R1 and/or R2. Split/Merge [98], OpenNF [53] and S6 [117] support cross-instance state transfers (R2). FTMB [106] and Pico Replication [96] focus on state availability (R1).

More fundamentally, Split/Merge, Pico Replication and FTMB focus on availability of the state contained entirely within an NF instance. They either ignore state shared across instances, or focus on the small class of NFs where such state is not used. Thus, these frameworks cannot handle R3.

Among existing frameworks, only OpenNF and S6 can support consistency for shared state (R3), but this comes at high performance cost. For example, OpenNF imposes a 166µs per packet overhead to ensure strong consistency! (§4.6). Similarly, S6 cannot support frequent updates to strongly consistent shared state.

Equally crucially, all of the above frameworks focus on a single NF; they cannot handle chains. Thus, none of them support chain-wide ordering (R4).

Support for R5 is also missing. StatelessNF [69] and S6 [117] update shared state in an external store or remote NF, respectively, but they do not support atomic updates to all state objects an instance can access. Thus, when a clone is created to mitigate a straggler off-path NF (as outlined above), the straggler may have updated other state objects that are not reflected in the clone's initialized state. Upon replay, the straggler can make duplicate state updates (likewise, duplicate packets can also arise). For the same reason, R6 is also violated: when an NF fails over, replaying packets to bring the recovery NF up to speed can result in duplicate processing in downstream NFs.

State management performance is poor: FTMB's periodic checkpointing significantly inflates NF packet processing latency (§4.6). As mentioned above, OpenNF imposes performance overhead for shared state. The overhead is high even for cross-instance transfers of per-flow state: this is because such transfers require extracting state from an instance and installing it in another while ensuring that incoming packets are directed to the state's new location.

Our contributions: How do we support requirements R1-R6 while ensuring good state management performance? Some NFs or operating scenarios may just need a subset of R1-R6. However, we seek a *single framework* that meets all requirements/scenarios because, with NFV becoming mainstream, we believe we can no longer trade-off general correctness requirements for performance or functionality (specific NFs). Thus, we identify *basic building blocks* and study how to synthesize them into one framework. Building such a framework is especially challenging because we must carefully deal with shared state and NF chaining.

Our system, CHC, has three building blocks (Figure 3.3): We maintain NF state in a **state store external to NFs** (1; §3.3). NFs access the store to read/write relevant state objects. This ensures state availability (R1). The store's state object metadata simplifies reasoning about state ownership

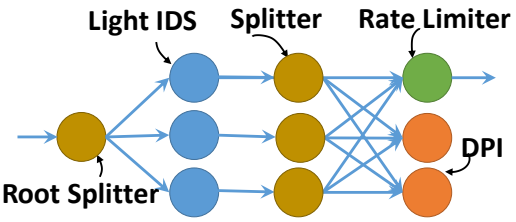


Figure 3.4: Physical chain that CHC runs.

and concurrency control across instances (2; §3.3.3). This makes state transfer safety (R2) and shared state consistency (R3) simple and efficient (§3.4.1).

We propose **NF state-aware algorithms** for good state read/write performance which is a key concern with state externalization. These include (§3.3.3): automatic state scope-aware traffic partitioning to minimize shared-state coordination (3); asynchronous state updates for state that is updated often but read infrequently; this allows packet processing to progress unimpeded (4); NFs sending update operations, as opposed to updated state, to the store, which simplifies synchronization and serialization of shared-state updates (5); scope- and access pattern-aware state caching strategies, which balances caching benefits against making cache updates immediately visible to other instances (6).

Finally, we maintain a small amount of **metadata – clocks and logs**. We insert per packet logical clocks (7; §3.4) which directly supports cross-instance ordering (R4). We couple clocks with logs to support duplicate suppression (R5; §3.4.3) and COE under failover of NFs and framework components (R6; §3.4.4). We log every packet that is currently being processed at some NF in the chain (8). Logged packets are replayed across the entire chain during failover. At the state store, we store logical clocks of packets along with the state updates they resulted in, which aids duplicate suppression. At each NF, we store packet clocks along with the update operations issued and the most recently read state value (9).

Together with state store snapshots, these NF-side logs support COE under datastore recovery.

Though StatelessNF [69] first advocated for externalizing state, but it has serious issues. Aside from a lack of support for R4–R6, it lacks atomic state updates: when a single NF fails after updating some but not all state objects, a failover NF can boot up with incorrect state! It requires locks for shared state updates, which degrades performance. Also, it assumes Infiniband networks for performance.

3.2 Framework: Operator View

In CHC, operators define "logical" NF chains (such as Figure 3.1b) using a DAG API. We elide low level details of the API, such as how policies are specified, and focus on aspects related to correctness and performance. Each "vertex" of the DAG is an NF and consists of operator supplied NF code, input/output, configuration, and state objects. Edges represent the flow of data (packets and/or contextual output).

The CHC framework compiles the logical DAG into a physical DAG with logical vertex mapped to one or more *instances* (Figure 3.4). For example, the IDS in Figure 3.1b is mapped to three instances in Figure 3.4. The operator can provide default parallelism per vertex, or this can be determined at run time using operator-supplied logic (see below). CHC deploys the instances across a cluster. Each instance processes a partition of the traffic input to the logical vertex; CHC automatically determines the traffic split to ensure even load distribution (§3.3).

The CHC framework supports chain elastic scaling and straggler mitigation. Note that the logic, e.g., when to scale is not our focus; we are interested in high performance state management and COE during such actions. Nevertheless, we outline the operator-side view for complete-

ness: Operators must supply relevant logic for each vertex (i.e., scaling[1]; identifying stragglers[2]). CHC executes the logic with input from a "vertex manager", a logical entity is responsible for collecting statistics from each vertex's instances, aggregating them, and providing them periodically to the logic.

Based on user-supplied logic, CHC redirects traffic to (from) scaled up (down) NF instances or clones of straggler NFs. CHC manages state under such dynamic actions to ensure COE. CHC also ensures system-wide fault tolerance. It automatically recovers from failures of NFs or of CHC framework components while always preserving COE.

3.3 Traffic and State Management

We discuss how CHC processes traffic and manages state. The framework automatically partitions traffic among NF instances (§3.3.1) and manages delivery of packets to downstream NFs (§3.3.2). As packets flow, different NFs process them and update state in an external store; CHC leverages several algorithms for fast state I/O; the main challenge here is dealing with shared state (§3.3.3).

3.3.1 Traffic partitioning

CHC performs *scope-aware partitioning*: traffic from an upstream instance is partitioned across downstream instances such that: (1) each flow is processed at a single instance, (2) groups of flows are allocated to instances such that most state an instance updates for the allocated flows is not updated by other instances, and (3) load is balanced. #1 and #2 reduce the need for cross-instance coordination for shared state.

[1] e.g., "when input traffic volume increased by a certain θ"
[2] when an instance processing θ% slower than other instances

In CHC, state scope is a *first-class entity*. A function *.scope()* associated with a vertex program returns a list of scopes i.e., the set of packet header fields which are used to key into the objects that store the states for an NF; i.e., these are the different granularities at which states can be queried/updated. CHC orders the list from the most to least fine grained scope. Suppose the DPI vertex in Figure 3.1b has two state objects: one corresponding to records of whether a connection is successful or not; and another corresponding to the number of connections per host. The scope for the former is the 5-tuple (*src IP, dst IP, src port, dst port, protocol*); the scope for the latter is *src IP*.

CHC first attempts to partition traffic at instances immediately upstream (which, for the DPI in Figure 3.1b would be the IDSes) based on the most coarse-grained state scope (for the DPI this is *src IP*); such splitting results in no state sharing at the downstream (DPI) instances. However, being coarse grained, it may result in uneven load across instances. The framework gathers this information via the (DPI) vertex manager. It then considers progressively finer grained scopes and repeats the above process until load is even.

The final scope to partition on is provided in common to the *splitters* upstream. The framework inserts a splitter after every NF instance (Figure 3.4). The splitter partitions the output traffic of the NF instance to instances downstream.

The *root* of a physical DAG is a special splitter that receives and splits input traffic. Roots can use multiple instances to handle traffic; in CHC, we fix root parallelism to some constant R. Network operators are required to statically partition traffic among the R roots such that the traffic processed by a root instance has no overlap in any of the 5-tuple dimensions with that processed by another instance.

Scope	Any	Per-flow	Cross-flow	Cross-flow
Access pattern	Write mostly, read rarely	Any	Write rarely (read heavy)	Write/read often
	Non-blocking ops. No caching	Caching \w periodic non-blocking flush	Caching \w callbacks	Depends upon traffic split. Cache, if split allows; flush periodically

Table 3.5: Strategies for state management performance

3.3.2 Communication

Inter-NF communication is asynchronous and non-blocking. Each NF's outputs are received by the CHC framework which is responsible for routing the output to downstream instances via the splitter. The framework stores all the outputs received from upstream instances in a queue per downstream instance; downstream instances poll the queue for input. This approach offers three benefits: (a) upstream instances can produce output independent of the consumption rate of downstream instances, (b) the framework can operate on queue contents (e.g., delete messages before they are processed downstream), which is useful for certain correctness properties, e.g., duplicate suppression (§3.4), (c) user logic can use persistent queues to identify stragglers/uneven load.

3.3.3 State Maintenance

CHC *externalizes* NF state and stores it in an external distributed key-value datastore. Thus, state survives NF crashes, improving availability and satisfying requirement R1 (§3.1). All state operations are managed by the datastore (Figure 3.3). As described below, CHC incorporates novel algorithms and metadata to improve performance (Table 3.5).

State metadata: The datastore's client-side library appends metadata to the key of the state that an NF instance stores. This contains vertex_ID and instance_ID, which are immutable and are assigned by the framework. In CHC, the key for a per-flow (5 tuple) state object is: *vertex_ID* +

Operation	Description
Increment/ decrement a value	Increment or decrement the value stored at key by the given value.
Push/pop a value to/from list	Push or pop the value in/from the list stored at the given key.
Compare and update	Update the value, if the condition is true.

Table 3.6: Basic operations offloaded to datastore manager

instance_ID + obj_key, where obj_key is a unique ID for the state object. The instance_ID ensures that only the instance to which the flow is assigned can update the corresponding state object. Thus, this metadata simplifies reasoning about ownership and concurrency control. Likewise, the key for shared objects, e.g., *pkt_count*, is: *vertex_ID + obj_key*. All the instances of a logical vertex can update such objects. When two logical vertices use the same key to store their state, vertex_ID prevents any conflicts.

Offloading operations: Most NFs today perform simple operations on state. Table 3.6 shows common examples. In CHC, an instance can *offload operations* and instruct the datastore to perform them on state on its behalf (developed contemporarily with [117]). Developers can also load custom operations. The benefit of this approach is that NF instances do not have to contend for shared state. The datastore serializes operations issued by different instances for the same shared state object and applies them in the background (§A.3.1 proves that updates will always result in consistent state.). This offers vastly better performance than the natural approach of acquiring a lock on state, reading it, updating, writing it back, and releasing the lock (§4.6).

Non-blocking updates: In many cases, upon receiving a packet, an NF updates state, but does not use (read) the updated value; e.g., typical packet counters (e.g., [16, 20, 88]) are updated every input packet, but the updated value is only read infrequently. For such state that is written mostly and read rarely, we offer *non-blocking updates* (Table 3.5): the datas-

tore immediately sends the requesting instance an ACK for the operation, and applies the update in the background. As a further optimization, NFs do not even wait for the ACK of a non-blocking operation; the framework handles operation retransmission if an ACK is not received before a timeout. If an instance wishes to read a value, the datastore applies all previous outstanding updates to the value, in the order NFs issued them, before serving the read.

Caching: For all the objects which are not amenable to non-blocking updates, we improve state access performance using novel caching strategies that leverage state objects' scope and access patterns (ready-heavy vs. not).

Per-flow state: CHC's scope-aware partitioning ensures that flows that update per-flow state objects are processed by a single instance; thus, these objects do not have cross-instance consistency requirements. The datastore's client-side library caches them at the relevant instance, which improves state update latency and throughput. However, for fault tolerance, we require local updates made to cached objects to be flushed to the store; to improve performance, these flush operations have non-blocking semantics (Table 3.5).

Cross-flow state: Cross-flow state objects can be updated by multiple instances simultaneously. Unlike prior works that largely ignore such state, CHC supports high performance shared state management. Some shared objects are rarely updated; developers can identify such objects as read-heavy. CHC (1) caches such an object at the instances needing them; and (2) the client-side library at each of these instances registers a *callback* with the store, which is invoked whenever the store updates the object on behalf of another instance.

The cached objects only *serve read requests*. Whenever an (rare) update is issued by an instance - operation is immediately sent to the store, The store applies the operation and sends back the updated object to the update

initiator. At the same time, the client-side library of other instances receives callback from the store and updates the locally cached value (Table 3.5). We prove this approach results in consistent updates to shared state in §A.3.2.

For other cross-flow objects (not rarely-updated), the datastore allows them to be cached at an instance only as long as no other instance is accessing them (Table 3.5); otherwise, the objects are flushed. CHC notifies the client-side library when to cache or flush the state based on (changes to) the traffic partitioning at the immediate upstream splitter.

For **scale and fault tolerance** we use multiple datastore instances, each handling state for a subset of NF instances. Each datastore instance is multi-threaded. A thread can handle multiple state objects; however, each state object is only handled by a single thread to avoid locking overhead.

3.4 Correctness

So far, we focused on state management and its performance. We also showed how CHC supports requirement R1 (state availability) by design. We now show how it supports the requirements R2–R6. This is made challenging both by shared state and by chaining. To support R2-R6, CHC maintains/adds metadata at the datastore, NFs and to packets. We first describe how the most basic of the metadata – logical packet clocks and packet logs – are maintained. We describe other metadata along with the requirements they most pertain to.

Logical clocks, logging: The root (§3.3.1) attaches with every input packet a unique *logical clock* that is incremented per packet. The root also *logs* in the datastore each packet, the packet clock, and to which immediate downstream instance the packet was forwarded. When the last NF in a chain is done processing a packet, updating state and generating relevant output, it informs the CHC framework. CHC sends a "delete" request

with the packet's clock to the root which then removes the packet from the log. Thus, at any time, the root logs all packets that are being processed by one or more chain instances. When any NF in the chain cannot handle the traffic rate, the root log builds in size; CHC drops packets at the root when this size crosses a threshold to avoid buffer bloat. When multiple root instances are in use (§3.3.1), we encode the identifier of the root instance into the higher order bits of the logical clock inserted by it to help the framework deliver "delete" requests to the appropriate root instance.

3.4.1 R2, R3: Elastic scaling

In some situations, we may need to reallocate ongoing processing of traffic across instances. This arises, e.g., in elastic scaling, where a flow may be processed at an "old" instance and reallocated to a "new" scaled up instance. We must ensure here that the old and new instances operate on the correct values of per- and cross-flow state even as traffic is reassigned (requirements R2 and R3).

Specifically, for cross-flow shared state, we require that: *Updates made to the shared state by every incoming packet are reflected in a globally consistent order irrespective of which NF instance processed the corresponding packet.*

Existing systems achieve this at high overhead: OpenNF [53] copies shared internal state from/to the instances sharing it, each time it is updated by an incoming packet! In contrast, ensuring this property in CHC is straightforward due to externalization and operation offloading (§3.3.3): when multiple instances issue update operations for shared state, the datastore serializes the operations and applies in the background. All subsequent accesses to the shared state then read a consistent state value.

Per-flow state's handling must be correctly reallocated across instances, too (R2). One approach is to disassociate the old instance from the state object (by having the instance remove its instance_ID from the object's metadata) and associate the new instance (by adding its instance_ID). But,

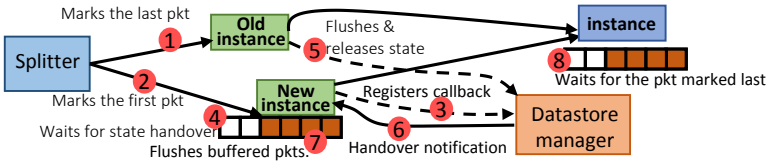


Figure 3.7: State handover.

this does not ensure correct handover when there are in-transit packets that update the state: even if the upstream splitter immediately updates the partitioning rules and the traffic starts reaching the new instance, there might be packets in-transit to, or buffered within, the old instance. If the new instance starts processing incoming packets right away then state updates due to in-flight/buffered packets may be disallowed by the datastore (as a new instance is now associated with the state object) and hence the updates will be lost.

Thus, to satisfy R2, we require: *Loss-freeness, i.e., the state update due to every incoming packet must be reflected in the state object.* Furthermore, some NFs may also need *order-preservation: updates must happen in the order of packet arrivals into the network.*

These properties are crucial for off-path NFs, e.g., IDS. Such NFs cannot rely on end-to-end retransmissions to recover from lost updates induced by traffic reallocation [53]. Similarly, they may have to process packets in the order in which they are exchanged across two directions of a flow, and may be thwarted by a reordering induced by reallocation (resulting in false positives/negatives).

Figure 3.7 shows the sequence of steps CHC takes for R2: (1) The splitter marks the "last" packet sent to the old instance to inform the old instance that the flow has been moved. This mark indicates to the old instance that it should flush any cached state associated with the particular flow(s) to the datastore and disassociate its ID from the per flow state, once it has processed the "last" packet. (2) The splitter also marks the "first"

packet from the traffic being moved to the new instance. 3 When the new instance receives the "first" packet, it tries to access the per flow state from the datastore. If the state is still associated with the old instance_ID, it registers a callback with the datastore to be notified of metadata updates. 4 The new instance starts buffering all the packets associated with the flow which is being moved. 5 After processing the packet marked as "last", the old instance flushes the cached state and updates the metadata to disassociate itself from the state. 6 The datastore notifies the new instance about the state handover. 7 The new instance associates its ID with the state, and flushes its buffered packets.

The above ensure that updates are not lost *and* that they happen in the order in which packets arrived at the upstream splitter. In contrast, OpenNF provides *separate algorithms* for loss-freeness and order-preservation; an NF author has the arduous task of choosing from them!

Note also that packets may arrive out of order at a *downstream* instance, causing it to make out-of-order state updates. To prevent this: 8 The framework ensures that packets of the moved flow emitted by the new instance are not enqueued at the downstream instance, but instead are buffered internally within the framework until the packet marked as "last" from the old instance is enqueued at the new instance.

3.4.2 R4: Chain-wide ordering

To support R4, we require that: *Any NF in a chain should be able to process packets, potentially spread across flows, in the order in which they entered the NF chain.* CHC's logical clocks naturally allow NFs to reason about cross-flow chain-wide ordering and satisfy R4. E.g., the Trojan detector from §3.1.1 can use packets' logical clocks to determine the arrival order of SSH, FTP and IRC connections.

3.4.3 R5: Straggler mitigation

R5 calls for the following: *All duplicate outputs, duplicate state updates, and duplicate processing are suppressed.*

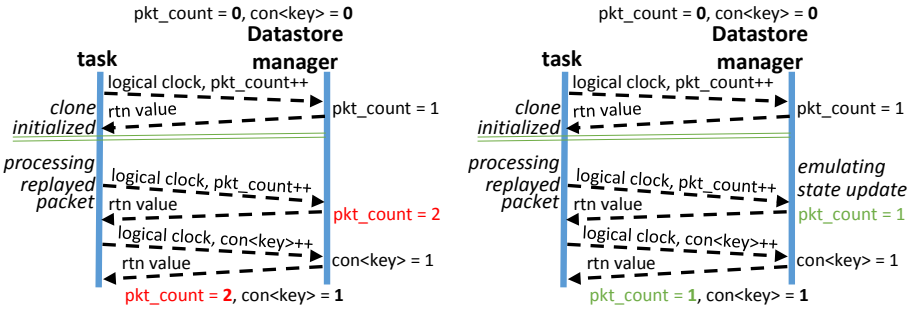
A key scenario in which duplicate suppression is needed is straggler mitigation. A straggler is a slow NF that causes the entire NF chain's performance to suffer. We first describe CHC's mechanism for straggler mitigation (which kicks in once user-provided logic identifies stragglers; §3.2), followed by duplicate suppression.

Clone and replay: To mitigate stragglers CHC deploys *clones*. A clone instance processes the same input as the original in parallel. CHC retains the faster instance, killing the other. CHC initializes the clone with the straggler's latest state from the datastore. It then replicates incoming traffic from the upstream splitter to the straggler and the clone.

This in itself is not enough, because we need to satisfy R2, i.e., ensure that the state updates due to packets that were in-transit to the straggler at the time the clone's state was initialized are reflected in the state that the clone accesses. To address this, we *replay* all logged packets from the root. The root continues to forward new incoming packets alongside replayed ones. The clone processes replayed traffic first, and the framework buffers replicated traffic. To indicate end of replay traffic, the root marks the "last" replayed packet (this is the most recent logged packet at the time the root started replaying). When replay ends (i.e., the packet marked "last" was processed by the clone), the framework hands buffered packets to the clone for processing.

Given the above approach for straggler mitigation, there are three forms of duplicates that can arise. CHC suppresses them by maintaining suitable metadata.

1. Duplicate outputs: Replicating input to the clone results in duplicate outputs. Here, the framework suppresses duplicate outputs associated with the same logical clock at message queue(s) of immediate



(a) Naive reprocessing (b) Reprocessing with emulation

Figure 3.8: Duplicate update suppression

downstream instance(s).

2. Duplicate state updates: Some of the replayed packets may have already updated some of the stragglers' state objects. For example, an IDS updates both the total packet count and the number of active connections per host. A clone IDS may have been initialized after the straggler updated the former but not the latter. In such cases, processing a replayed packet can *incorrectly* update the same state (total packet count) multiple times at the straggler (Figure 3.8a). To address this, the datastore logs the state value corresponding to each state update request issued by any instance, as well as the logical clock of the corresponding packet. This is only done for packets that are currently being processed by some NF in the chain. During replay, when the straggler or clone sends an update for a state object, the datastore checks if an update corresponding to the logical clock of the replayed packet has already been applied; if so, the datastore *emulates* the execution of the update by returning the value corresponding to the update (Figure 3.8b). In Appendix A.2, we describe how CHC handles non-deterministic state update operations.

3. Duplicate upstream processing: NFs upstream from the clone/straggler would have already processed some of the in-transit packets. In such cases, reprocessing replayed packets leads to incorrect actions at

upstream NFs (e.g., an IDS may raise false alarms). To address this, each replayed packet is marked and it carries the ID of the clone where it will be processed. Such packets need special handling: the intervening instances recognize that they are not suspicious duplicates; if necessary, the instances read the store for state corresponding to the replayed packet, make any needed modifications to the packet's headers, and produce relevant output; the instances can issue updates to state, too, but in such cases the datastore *emulates* updates as before. The clone's ID is cleared once it processed the packet.

3.4.4 R6: Safe Fault Recovery

Our description of R6 in §3.1 focused on NF failures; however, since CHC introduces framework components, we generalize R6 to cover other failures as well. Specifically, we require the following general guarantee:

Safe recovery Guarantee: *When an NF instance or a framework component fails and a recovery occurs, we must ensure that the state at each NF in the chain has the same value as under no failure.*

We assume the standard fail-stop model, that a machine/node can crash at any time and that the other machines/nodes in the system can immediately detect the failure.

First, we show how CHC leverages metadata to handle the failure of individual components. Then, we discuss scenarios involving simultaneous failure of multiple components.

NF Failover: When an NF fails, a failover instance takes over the failed instance's processing. The datastore manager associates the failover instance's ID with relevant state. Packet replay brings state up-to-speed (from updates due to in-transit packets). Similar to cloning (§3.4.3), we suppress duplicate state updates and upstream processing.

Since "delete" requests are generated after the last NF is done processing a packet, failure of such an NF needs special handling: consider such

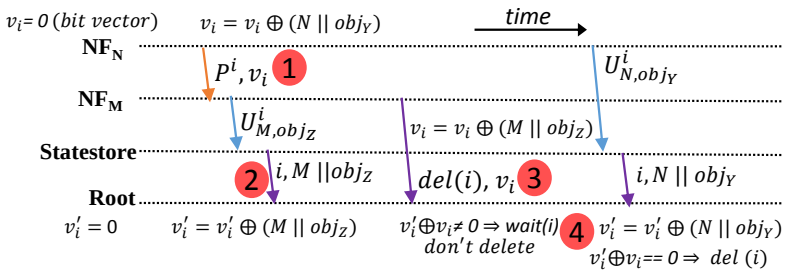


Figure 3.9: Recovery under non-blocking operations. Consider a packet P^i which is processed by NF_N, followed by NF_M, the last NF in the chain. NF_N and NF_M update objects obj_Y and obj_Z, resply.

an instance T failing after generating an output packet for some input packet P, but before the framework sends a "delete" request for P. When P is replayed, T's failover instance produces output again, resulting in duplicate packets at the receiving end host. To prevent this, for the last NF in the chain, our framework sends the "delete" request for P *before* the NF generates the output packet. If the NF fails before the "delete" request, then P will be replayed, but this does not result in duplicate downstream processing since the NF did not generate output. If the NF fails after the "delete" request but before generating output, then P is not replayed, and hence the end host will not receive any output packet corresponding to P. This will appear as a packet loss at the host, causing P to be retransmitted from the source and resulting in correct overall behavior. In §A.3.4, we show that using this protocol an NF instance recovers with state similar to that under no failure.

Non-blocking operations: Non-blocking updates, where NF instances don't wait for ACKs, instead relying on the framework to handle reliable delivery, can introduce the following failure mode: a instance may fail after issuing state update but before the update is committed and an ACK was received. In such cases, to ensure R6, we need that *the framework must re-execute the incomplete update operation.*

Suppose an instance N fails after processing packet P_i (i is the logical clock) but before the corresponding state update operation $U^i_{N,obj}$ (obj is the state object ID) completes. P_i may have induced such operations at a subset of NF instances $\{N\}$ along the chain. A natural idea to ensure the above property is to replay packets from the root to reproduce $U^i_{N,obj}$ at various N's. For this, however, P_i must be logged and should not have been deleted. If P_i is deleted it can't be replayed.

We need to ensure P_i continues to be logged as long as there is some N for which $U^i_{N,obj}$ is not committed. Our approach for this is shown in Figure 3.9: (1) Each packet carries a 32-bit vector v_i (object ID and instance ID; 16b each) that is initialized to zero. Each NF instance where processing the packet resulted in a state update XORs the concatenation of its ID and the corresponding state objects' IDs into the bit vector. (2) When committing a given NF's state update, the state store signals to the root the clock value of the packet that induced the update as well as the concatenated IDs. (3) The last instance sends the final vector along with its "delete" request to the root. (4) When a delete request and the final vector are received, the root XORs the concatenated IDs with the concatenated IDs reported by each signal from the state store in step 2. If the result is zero, this implies that updates induced by the packet at all NF instances $\{N\}$ were committed to the store; the root then proceeds to delete the packet from the log. Otherwise, the packet updated state at some NF, but the NF has not yet reported that the state was committed; here, the root does not delete the packet.

Root: To ensure R6 under root failover, we need that a new root must start with the logical clock value and current flow allocation at the time of root failure. This is so that the new root processes subsequent packets correctly. To ensure this, the failover root reads the last updated value of the logical clock from the datastore, and retrieves how to partition traffic by querying downstream instances' flow allocation. The framework buffers

incoming packets during root recovery. We prove this approach ensures recovery with a state similar to that under no failure in §A.3.3.

Datastore instance: Recall that different NFs can store their states in different storage instances (§3.3.3). This ensures that store failures impact availability of only a portion of the overall state being tracked. Now, to ensure R6 under the failure of a datastore instance, we need that the recovered state in the new store instance must represent the value which would have resulted if there was no failure. The recovered state must also be consistent with the NF instances' view of packet processing thus far (i.e., until failure).

To support this property we distinguish between per-flow and shared state. For the former, we leverage the insight that all the NFs already maintain an updated cached copy of per-flow state. If a datastore instance fails, we can simply query the last updated value of the cached per-flow state from all NF instances that were using the store.

Recovering shared state is nuanced. For this, we use checkpointing with write-ahead logging [84]. The datastore periodically checkpoints shared state along with the metadata, "TS", which is the set of logical clocks of the packets corresponding to the last state operation executed by the store on behalf of each NF instance. Each instance locally writes shared-state update operations in a write-ahead log. Say the latest checkpoint was at time t and failure happens at $t + \delta$. A failover datastore instance boots with state from the checkpoint at t. This state now needs to be "rolled forward" to $t + \delta$ and made consistent with the NF instances' view of packet processing at $t + \delta$. Two cases arise:

(Case 1) If NF instances that were using the store instance don't read shared state in the δ time interval, then to recover shared state, the framework re-executes state update operations from the local write-ahead log on behalf of each NF, starting from the logical clocks included in the metadata TS in the checkpoint. Recall that in our design the store applies updates in

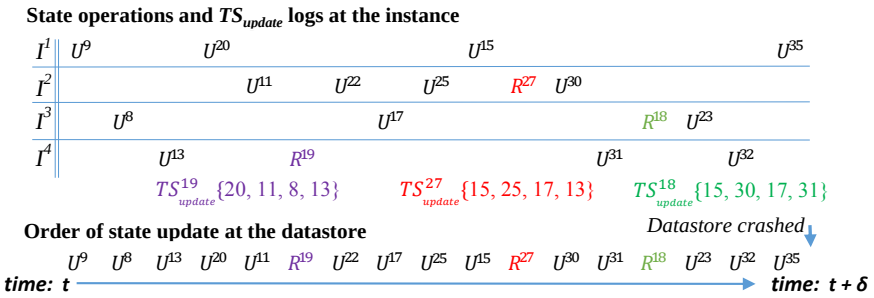


Figure 3.10: Recovering shared state at the datastore. I^k are instances. $U^{logical\_clock}$ and $R^{logical\_clock}$ represent "update" and "read".

the background, and this update order is unknown to NF instances. Thus, our approach ensures that the state updates upon re-execution match that produced by a *plausible* sequence of updates that the store may have enforced prior to failure. This consistency property suffices because, in Case 1, NFs are not reading shared state in the δ interval.

(Case 2) Say an NF instance issues a read between t and $t + \delta$; e.g., I^3 in Figure 3.10 issues R^{18}. Following the above approach may lead to an order of re-execution such that the actual state I^3 read in R^{18} is different from the state in the store after recovery. To ensure that the store's state is consistent with all I^k's current view, the framework must re-execute operations in such an order that the datastore would have produced the same value for each read in $[t, t + \delta]$.

To ensure this, on every read operation, the datastore returns TS along with the latest value of the shared state (e.g., TS^{19} is returned with I^4's R^{19}). The instance then logs the value of the shared state along with the corresponding TS. Re-execution upon failure then needs to select, among all TS's at different instances, the one corresponding to the most recent read from the store prior to the crash (i.e., TS^{18}, since R^{18} in the most recent read; most recent clock does not correspond to most recent read). How selection is done is explained shortly; but note that when the

framework re-executes updates starting from the clock values indicated by this selected TS that would bring the store in sync with all NFs. In our example, TS^{18} is the selected TS; we initialize the store state with the value in the corresponding read (R^{18}). From the write-ahead log of each NF, the framework re-executes update operations that come after their corresponding logical clocks in TS^{18}. At instance I^1, this is the update after U^{15}, i.e., U^{35}. At I^3 and I^4 these are U^{23} and U^{32}, respectively. Shared state is now in sync with all NFs.

TS selection works as follows: first we form a set of all the TS's at each instance, i.e., $Set = \{TS^{18}, TS^{19}, TS^{27}\}$. Since the log of operations at an instance follows a strict clock order we traverse it in the reverse order to find the latest update operation whose corresponding logical clock value is in Set. For example, if we traverse the log of I^1, we find that the logical clock of U^{15} exists in Set. After identifying such a logical clock value, we remove all the entries from Set which do not contain the particular logical clock value (such TSs cannot have the most recent read); e.g., we remove TS^{19} as it does not contain logical clock 15. Similarly, we remove TS^{27}, after traversing I^2's log. Upon doing this for all instances we end up selecting TS^{18} for recovery. In §A.3.5, we prove that using this protocol the store recovers with state similar to that under no failure.

Correlated failures: Using the above approaches, CHC can also handle correlated failures (Table 3.11) of multiple NF instances, root, and storage instances. However, CHC cannot withstand correlated failure of a store instance with any other component that has stored its state in that particular instance. Replication of store instances can help recover from such correlated failures, but that comes at the cost of increasing the per packet processing latency.

	NF instance	Root
Store instance	✓*	✓*
NF instance	✓	✓

Table 3.11: Handling of correlated failures (*Cannot recover if component and the store instance storing its state fail together).

3.5 Implementation

Our prototype consists of an execution framework and a datastore, implemented in C++. NFs runs in LXC containers [10] as multithreaded processes. NFs are implemented using our CHC library that provides support for input message queues, client side datastore handling, retransmissions of un-ACK'd state updates (§3.3.3), statistics monitoring and state handling. Packet reception, transmission, processing and datastore connection are handled by different threads.

For low latency, we leverage Mellanox messaging accelerator (VMA) [114] which enables user-space networking with kernel bypass similar to DPDK [46]. In addition to this, VMA also supports TCP/UDP/IP networking protocols and does not require any application modification. Even though we use VMA, we expect similar performance with other standard kernel bypass techniques. Protobuf-c [18] is used to encode and decode messages between a NF instance and the datastore. Each NF instance is configured to connect to a "framework manager" to receive information about it's downstream instances (to which it connects via tunnels), datastore instances and other control information.

The framework manager can dynamically change the NF chain by instantiating new types of NFs or NF instances and updating partitioning information in upstream splitters[3]. Our datastore implements an in-memory key-value store and supports the operations in Table 3.6. We

[3]based on statistics from vertex managers

NF	Description of state object	Scope; access pattern
NAT	Available ports	Cross-flow; write/read often
	Total TCP packets	Cross-flow; write mostly, read rarely
	Total packets	Cross-flow; write mostly, read rarely
	Per conn. port mapping	Per-flow; write rarely, read mostly
Trojan detector	Arrival time of IRC, FTP and SSH flows for each host	Cross-flow; write/read often
Portscan detector	Likelihood of being malicious (per host)	Cross-flow; write/read often
	Pending conn. initiation req. along with its timestamp	Per-flow; write/read often
Load balancer	Per server active # of conn.	Cross-flow; write/read often
	Per server byte counter	Cross-flow; write mostly, read rarely
	Conn. to server mapping	Per-flow; write rarely, read mostly

Table 3.12: NFs and description of their state objects

reimplemented four NFs atop CHC. Table 3.12 shows their state objects, along with the state's scope and access patterns.

NAT: maintains the dynamic list of available ports in the datastore. When a new connection arrives, it obtains an available port from the datastore (The datastore pops an entry from the list of available ports on behalf of the NF). It then updates: 1) per-connection port mapping (only once) and, 2) (every packet) L3/L4 packet counters.

Portscan detector [102]: detects infected port scanner hosts. It tracks new connection initiation for each host and whether it was successful or not. On each connection attempt, it updates the likelihood of a host being malicious, and blocks a host when the likelihood crosses a threshold.

Trojan detector: implementation here follows [40].

Load balancer: maintains the load on each backend server. Upon a new connection, it obtains the IP of the least loaded server from the datastore and increments load. It then updates: 1) connection-to-server mapping 2) per server #connections and, 3) (every packet) per server byte

counter .

3.6 Evaluation

We use two packet traces (Trace{1,2}) collected on the link between our institution and AWS EC2 for trace-driven evaluation of our prototype. Trace1 has 3.8M packets with 1.7K connections and Trace2 has 6.4M packets with 199K connections. The median packet sizes are 368B and 1434B. We conducted all experiments with both traces and found the results to be similar; for brevity, we only show results from Trace2. We use six CloudLab [2] servers each with 8-core Intel Xeon-D1548 CPUs and a dual-port 10G NIC. One port is used to forward traffic, and the other for datastore communication and control messages. To process at 10Gbps, each NF instance runs multiple processing threads. CHC performs scope-aware partitioning of input traffic between these threads. Our datastore runs on a dedicated server.

3.6.1 State Management Performance

Externalization: We study three models which reflect the state access optimizations discussed in (§3.3.3): #1) All state is externalized and non-blocking operations are used. #2) Further, NFs cache relevant state objects. #3) Further, NFs do not wait for ACKs of non-blocking operations to state objects; the framework is responsible for retransmission (§3.3.3). The state objects per NF that benefit from #2 and #3 can be inferred from Table 3.5 and Table 3.12; e.g., for NAT, available ports and per-connection port mapping are cached in #2, and the two packet counters benefit from non-blocking updates in #3. We compare these models with a "traditional" NF where all state is NF-local. We study each NF type in isolation first.

Figure 3.13 shows the per packet processing times. The median times for traditional NAT and load balancer are 2.07µs and 2.25µs, resply. In

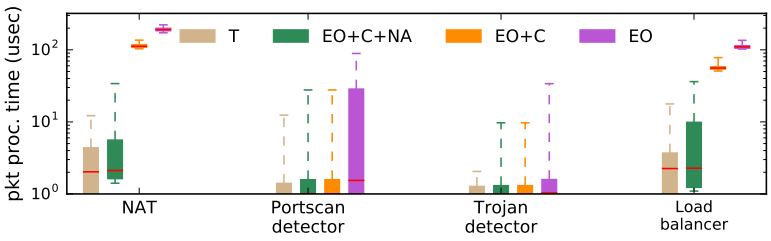


Figure 3.13: 5%ile, 25%ile, median, 75%ile and 95%ile pkt processing times. (T = Traditional NF, EO = Externalized state operations, C = with caching, NA = without waiting for the ACK)

model #1, this increases by 190.67µs and 109.87µs, resply., with network RTT contributing to most of this (e.g., NAT needs three RTTs on average per packet: one for reading the port mapping and other two for updating the two counters). We don't see a noticeable impact for scan and Trojan detectors (they don't update state on every packet).

Relative to #1, caching (#2) further lowers median processing times by 111.98µs and 55.94µs for NAT[4] and load balancer. For portscan and Trojan detector, reduces it by 0.54µs and 0.1µs (overhead becomes **+0.1µs** as compared to traditional NFs) as CHC caches the cross-flow state. Later, we evaluate the benefits of cross-flow caching in detail. Finally, #3 results in median packet processing times of **2.61µs** for NAT (which now needs 0 RTTs on average) and **2.27µs** for load balancer. These represent small overheads compared to traditional NFs: **+0.54µs** for NAT, and **+0.02µs** for the load balancer (at the median). Note that for portscan and Trojan detector the performance of #3 is comparable to #2 as they don't have any blocking operations.

We constructed a simple chain consisting of one instance each of NAT, portscan detector and load balancer in sequence, and the Trojan detector operating off-path attached to the NAT. With model #3, the median end-

[4]NAT needs 2 RTTs to update counters as port mapping is cached.

to-end overhead was **11.3µsec** compared to using traditional NFs.

Operation offloading: We compare CHC's operation offloading against a naive approach where an NF first reads state from the datastore, updates it, and then writes it back. We turn off caching optimizations. We now use two NAT instances updating shared state (available ports and counters). We find that the median packet processing latency of the naive approach is **2.17X** worse (64.6µs vs 29.7µs), because it not only requires 2 RTTs to update state (one for reading the state and the other for writing it back), but it may also have NFs wait to acquire locks. CHC's aggregate throughput across the two instances is >2X better.

Cross-flow state caching: To show the performance of our cross-flow state caching schemes (Table 3.5; Col 5), we run the following experiment: we start with a single portscan detector. After it has processed around 212K packets, we add a second instance and split traffic such that for particular hosts, captured by the set $\mathcal{H}$, processing happens at both instances. At around 213K packets, we revert to using a single instance for all processing. Figure 3.14 shows the benefits of caching the shared state. At 212K packets, when the second instance is added, the upstream splitter signals the original instance to flush shared state corresponding to hosts $\in \mathcal{H}$ (Table 3.12). From this point on, both instances make blocking state update operations to update the likelihood of hosts $\in \mathcal{H}$ being malicious on every successful/unsuccessful connection initiation. Thus, we see an increase in per packet processing latency for every SYN-ACK/RST packet. At packet number 213K, all processing for $\mathcal{H}$ happens at a single instance which can start caching state again. Thus, the processing latency for SYN-ACK/RST packets drops again, because now state update operations are applied locally and updates are flushed in a non-blocking fashion to the store.

Throughput: We measure degradation in per NF throughput for models #1 and #3 above compared to traditional NFs. Figure 3.15 shows that

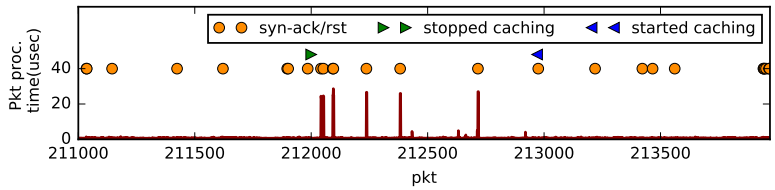


Figure 3.14: Per packet processing latency with cross-flow state caching

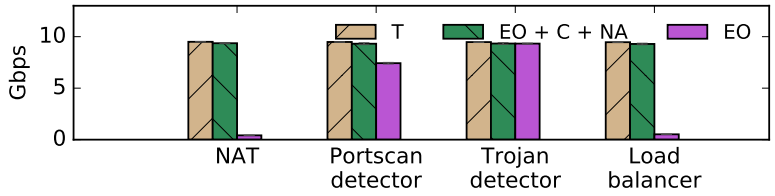


Figure 3.15: Per instance throughput. (T = Traditional NF, EO = Externalized state operations, C = with caching, NA = without waiting for the ACK)

the max. per NF throughput for traditional NFs is around 9.5Gbps. Under model #1, load balancer and NAT throughput drops to 0.5Gbps. The former needs to update a byte counter (which takes 1 RTT) on every packet; likewise, the NAT needs three RTTs per packet. The port scan and Trojan detectors do not experience throughput degradation because they don't update state on every packet. Model #3 increases throughput to 9.43Gbps, matching traditional load balancer and NAT. We repeated our experiment with the aforementioned single-instance NF chain and observed similar maximal performance (9.25Gbps with both CHC and traditional NFs) in Model #3.

Datastore performance We benchmarked the datastore using the workload imposed by our state operations. We found that a single instance of our datastore supports ~**5.1M** ops/s (increment at 5.1M ops/s, get at 5.2M ops/s, set at 5.1M ops/s; Table 3.6). The datastore can be easily scaled to support a greater rate of operations by simply adding multiple

instances; each state object is stored at exactly one store node and hence no cross-store node coordination is needed.

3.6.2 Metadata Overhead

Clocks: The root writes packet clocks to the datastore for fault tolerance. This adds a 29µs latency per packet (dominated by RTT). We optimize further by writing the clock to the store after every n^{th} packet.[5] The average overhead per packet reduces to 3.5µs and 0.4µs for $n = 10, 100$.

Packet logging: We evaluated two models of logging: 1) locally at the root, 2) in the datastore. The former offers better performance, adding 1µs latency per packet, whereas the latter adds 34.2µs but is more fault tolerant (for simultaneous root and NF failures). We also studied the overhead imposed by the framework logging clocks and operations at NFs, the datastore logging clocks and state, and the XOR-ing of identifiers (§3.4.4); the performance impact for our chain (latency and throughput overhead) was negligible ($< 1\%$).

XOR check and delete request: (§3.4.4) XOR checks of bit vectors are performed asynchronously in the background and do not introduce any latency overhead. However, ensuring the successful delivery of "delete" request to root before forwarding the packet introduces a median latency overhead of 7.9µsec. Asynchronous "delete" request operation eliminates this overhead but failure of the last NF in a chain may result in duplicate packets at the receiver end host.

3.6.3 Correctness Requirements: R1–R6

R1: State availability: Using our NAT, we compare FTMB's [106] checkpointing approach with CHC writing all state to a store. We could not

[5] After a crash, this may lead the root to assign to a packet an already assigned clock value. To overcome this issue, the root starts with *n + last update* so that clock values assigned to packets represent their arrival order.

obtain access to FTMB's code; thus, we emulate its checkpointing overhead using a queuing delay of 5000µs after every 200ms (from Figure 6 in [106]). Figure 3.17 (with 50% load level) shows that checkpointing in FTMB has a significant impact: the 75th%-ile latency is 25.5µsec – which is **6X** worse than that under CHC (median is 2.7X worse). FTMB's checkpointing causes incoming packets to be buffered. Because of externalization in CHC, there is no need for such checkpointing. Also, FTMB does not support recovery of the packet logger [106]. CHC intrinsically supports this (§3.4.4), and we evaluate it in §3.6.3.

R2: Cross-instance state transfers: We elastically scale up NAT as follows: We replay our trace for 30s through a single instance; midway through replay, we reallocate 4000 flows to a new instance, forcing a move of the state corresponding to these flows. We compare CHC with OpenNF's loss-free move; recall that CHC provides both loss-freeness and order preservation. CHC's move operation takes 97% **or 35X** less time (0.071ms vs 2.5ms), because, unlike OpenNF, CHC does not need to transfer state. It notifies the datastore manager to update the relevant instance_IDs. However, when instances are caching state, they are required to flush cached state operations before updating instance_IDs. Even then, CHC is 89% better because it *flushes only operations*.

R3: Cross-instance state sharing: We compare CHC against OpenNF w.r.t. the performance of strongly consistent shared state updates across NAT instances, i.e., updates are serialized according to some global order. Figure 3.16 (with 50% load level) shows that CHC's median per-packet latency is 99% lower than OpenNF's (1.8µs vs 0.166ms). The OpenNF controller receives all packets from NFs; each is forwarded to every instance; the next packet is released only after all instances ACK. CHC's store simply serializes all instances' offloaded operations.

R4: Chain-wide ordering: We revisit the chain in Figure 3.2. Each scrubber instance processes either FTP, SSH, or IRC flows. To measure

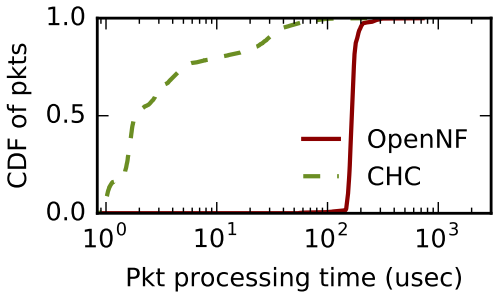


Figure 3.16: State sharing.

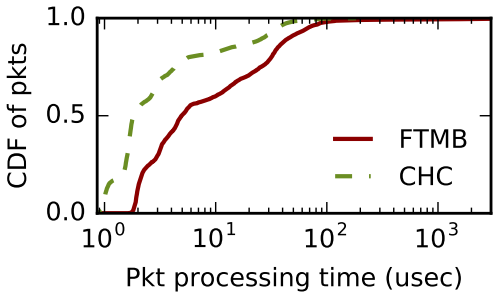


Figure 3.17: Fault recovery.

	30%load	50%load
Duplicate packets	13768	34351
Duplicate state updates	233	545

Table 3.18: Duplicate packet and state update at the downstream portscan detector without duplicate suppression.

the accuracy of the Trojan detector, we added the signature of a Trojan at 11 different points in our trace. We use three different workloads with varying upstream NF processing speed: W1) One of the upstream NFs adds a random delay between 50-100µs to each packet. W2) Two of the upstreams add the random delay. W3) All three add random delays. We observed that CHC's use of chain-wide logical clocks helps the Trojan detector identify **all 11 signatures**. We compare against OpenNF which does not offer any chain-wide guarantees; we find that **OpenNF misses 7, 10, and 11 signatures** across W1–W3.

R5: Duplicate suppression: Here, we emulated a straggler NAT by adding a random per packet delay between between 3-10µs. A portscan detector is immediately downstream from the NAT. CHC launches a clone NAT instance according to §3.4.3. We vary the input traffic load. Table 3.18 shows the number of duplicate packets generated by the NAT instances under different loads, as well as the number of duplicate state updates at the portscan detector – which happen whenever a duplicate packet

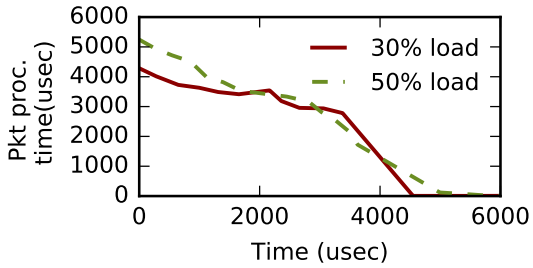


Figure 3.19: Packet proc time.

triggers the scan detector to *spuriously* log a connection setup/teardown attempt. Duplicate updates create both false positives/negatives and their incidence worsens with load. No existing framework can detect such duplicate updates; CHC simply suppresses them.

R6: Fault Tolerance: We study CHC failure recovery.

NF Failure: We fail a single NAT instance and measure the recovery and per packet processing times. Our NAT performs non-blocking updates without waiting for the framework ACK; here, we use the 32bit vector (§3.4.4) to enable recovery of packets whose non-blocked operations are not yet committed to the store. To focus on CHC's state recovery, we assume the failover container is launched immediately. Figure 3.19 shows the average processing time of packets that arrive at the new instance at two different loads. The average is calculated over 500µs windows. Latency during recovery spikes to over 4ms, but it only takes **4.5ms** and **5.6ms** at 30% and 50% loads, resply., for it to return to normal.

Root failure: Recovering a root requires just reading the last updated logical clock from the datastore and flow mapping from downstream NFs. This takes < 41.2µs.

Datastore instance failure: Recovering a datastore instance failure requires reading per-flow state from NFs using it, and replaying update operations to rebuild shared state. Reading the latest values of per-flow

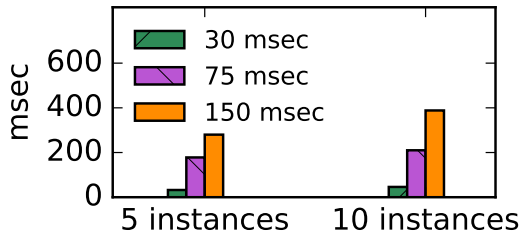


Figure 3.20: Store recovery.

state is fast. Recovering shared state however is more time-consuming. Figure 3.20 shows the time to rebuild shared state with 5 and 10 NAT instances updating the same state objects at a single store instance. We replayed the state update operation logs generated by these instances. The instances were processing 9.4Gbps of traffic; periodic checkpoints occurred at intervals of 30ms, 75ms, and 150ms. The recovery time is $\leqslant$ **388.2ms** for 10 NATs with checkpoints at 150ms intervals. In other words, a storage instance can be quickly recovered.

3.7 Conclusion

We presented a ground-up NFV framework called CHC to support COE and high performance for NFV chains. CHC relies on managing state external to NFs, but couples that with several caching and state update algorithms to ensure low latency and high throughput. In addition, it leverage simple metadata to ensure various correctness properties are maintained even under traffic reallocation, NF failures, as well as failures of key CHC framework components.

4

StateAlyzr

In chapter 2, we designed a scheme to ensure strong CPU isolation in containerized environments, and in chapter 3, we designed a ground up framework to support chain output equivalence (COE) and high performance for NFV chains. In this chapter, we present Statealyzr, a system we designed to simplify the process of modifying middleboxes or network functions to work with new frameworks such as [25, 31, 48, 53, 69, 86, 93, 96, 98, 104, 106, 117]. Middleboxes are complex softwares with multiple state objects which require explicit handling to ensure fault tolerance and provide elastic scaling.

To reduce manual effort and ease adoption, we develop StateAlyzr, a system that relies on *data and control-flow analysis* to automate identification of state objects that need explicit handling. Using StateAlyzr's output, developers can easily make framework-compliant changes to arbitrary middleboxes, e.g., identify which state to allocate using custom libraries for [69, 96, 98], determine where to track updates to state [53, 98, 106], (de)serialize relevant state objects for transfer/cloning [53], and merge externally provided state with internal structures [53, 96]. In practice we find StateAlyzr to be highly effective. For example, leveraging StateAlyzr to make PRADS OpenNF-compliant took under 6 man-hours of work.

Importantly, transferring/cloning state objects identified with StateAlyzr is provably *sound* and *precise*. The former means that the aggregate output of a collection of instances following redistribution is equivalent to the output that would have been produced had redistribution not occurred.

The latter means that StateAlyzr identifies minimal state to transfer so as to ensure that redistribution offers good performance and incurs low overhead.

However, achieving high precision without compromising soundness is challenging. Key attributes of middlebox code contribute to this: e.g., numerous data structures and procedures, large callgraphs, heavy use of (multi-level) pointers, and indirect calls to packet processing routines that modify state (See Table 4.3).

To overcome these challenges, StateAlyzr cleverly adapts program analysis techniques, such as slicing [63, 116] and pointer analysis [24, 108], to typical middlebox code structure and design patterns, contributing new algorithms for detailed classification of middlebox state. These algorithms can automatically identify: (i) variables corresponding to state objects that pertain to individual or groups of flows, (ii) the subset of these that correspond to state objects that can be updated by an arbitrary incoming packet at runtime, (iii) the flow space corresponding to a state object, (iv) middlebox I/O actions that are impacted by each state object, and (v) objects updated at runtime by an incoming packet.

To evaluate StateAlyzr, we both prove that our algorithms are sound (Appendix A.1) and use experiments to demonstrate precision and the resultant impact on the efficiency of state transfer/cloning. We run StateAlyzr on four open source middleboxes—Passive Real-time Asset Detection System (PRADS) [16], HAProxy load balancer [5], Snort Intrusion Detection System [19], and OpenVPN gateway [15]—and find:

- StateAlyzr's algorithms improve precision significantly: whereas the middleboxes have 1500-18k variables, only 29-131 correspond to state that needs explicit handling, and 10-148 are updateable at run time. By automatically identifying updateable state, StateAlyzr allows developers to focus on the necessary subset of variables among the many present. StateAlyzr can be imprecise: 18% of the updateable

variables are mis-labeled (they are in fact read-only), but the information StateAlyzr provides allows developers to ignore processing these variables.

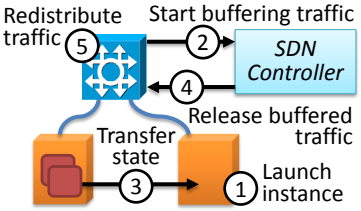
- Using StateAlyzr output, we modified PRADS and Snort to support fault tolerance using OpenNF [53]. We find that StateAlyzr reduces the manual effort needed. We could modify Snort (our most complex middlebox) and PRADS in 90 and 6 man-hours, respectively. Further, by helping track which flowspace an incoming packet belongs to, and which state objects it had updated, StateAlyzr reduces unneeded runtime state transfers between the primary and backup instances of PRADS and Snort by 600× and 8000× respectively compared to naive approaches.
- StateAlyzr can process middlebox code in a reasonable amount of time. Finally, it helped us identify important variables that we missed in our earlier modifications to PRADS, underscoring its usefulness.

4.1 Motivation

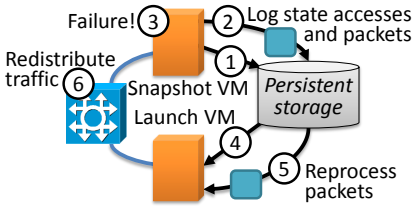
A central goal of NFV is to create more scalable and fault tolerant middlebox deployments, where middleboxes *automatically scale* themselves in accordance with network load and *automatically heal* themselves when software, hardware, or link failures occur [11]. Scaling, and possibly fault tolerance, requires launching middlebox instances on demand. Both require redistributing network traffic among instances, as shown in Figure 4.1.

4.1.1 Need for Handling State

Middlebox scaling and failure recovery should be transparent to end-users and applications. Key to ensuring this is maintaining *output equivalence*: for any input traffic stream, the aggregate output of a dynamic set of



(a) Scaling with Split/Merge [98]



(b) Failure recovery with FTMB [106]

Figure 4.1: Scaling and failure recovery process with recently state management frameworks

middlebox instances should be equivalent to the output produced by a single, monolithic, always-available instance that processes the entire input [98]. The output may include network traffic and middlebox logs.

As shown in prior works [53, 98, 106], achieving output equivalence is hard because middleboxes are *stateful*. Every packet the middlebox receives may trigger updates to multiple pieces of internal state, and middlebox output is highly dependent on the current state. Thus, malfunctions can occur when traffic is rerouted to a middlebox instance without *the relevant internal state being made available at the instance*. Approaches like naively rerouting newly arriving flows or forcibly rerouting flows with pertinent state can violate output equivalence. The reader is referred to [53, 96] for a more formal treatment of the need to handle internal state.

4.1.2 Approaches for Handling State

Traditional approaches for replicating and sharing application state are resource intensive and slow [53, 96, 98]. Thus, researchers have introduced fast and efficient frameworks that transfer, clone, or share live internal middlebox state across instances. Examples include: *Split/Merge* [98] and *StatelessNF* [69] that focus on elasticity; *Pico replication* [96] and *FTMB* [106] that focus on fault tolerance; and *OpenNF* [53] that applies to both. Unfor-

Framework	Provides	Required Modifications: State Alloc.	State Access	Serial-ization	Merge State
Split/Merge [98]	Elasticity	✓	✓		✓
Pico Rep. [96]	Fault tol.	✓	✓		
OpenNF [53]	Both			✓	✓
FTMB [106]	Fault tol.		✓		
StatelessNF [69]	Both	✓	✓		

Table 4.2: Middlebox modifications in different frameworks

tunately, these frameworks require detailed modifications to middlebox code to handle state (see Table 4.2):

- Split/Merge [98] and Pico Replication [96] require middleboxes to allocate and access all per- and cross-flow state—i.e., state that supports the processing of multiple packets within and across flows, respectively—through a specialized shared library, instead of using system-provided functions (e.g., malloc). This allows the frameworks to transfer and replicate middlebox state without serializing or updating middlebox-internal structures.
- OpenNF [53] requires middleboxes to identify and serialize per- and cross-flow state objects pertaining to a particular flowspace, as well as deserialize and integrate objects received from other middlebox instances. This allows OpenNF to transfer and copy flow-related state between middlebox instances.
- FTMB [106] requires middleboxes to log: (*i*) accesses to cross-flow state, and (*ii*) invocations of non-deterministic functions (e.g., gettimeofday). The logs allow FTMB to deterministically reprocess packets on a different middlebox instance in case the current instance fails before an up-to-date snapshot of its state can be captured.
- StatelessNF [69] requires middleboxes to create, read, and update all state values from a central, RDMA (remote direct memory access)

Middlebox	LOC (C/C++)	Classes/ Structs	Event based?	Level of pointers	Number of procedures	Size of callgraph
Snort IDS [19]	275K	898	No	10	4617	3391
HAProxy load balancer [5]	63K	191*	No	8*	2560	1018
OpenVPN [15]	62K	194*	No	2*	2023	1184
PRADS asset detector [16]	10K	40	No	4	297	115
Bro IDS [88]	97K	1798	No	-	3034	-
Squid caching proxy [20]	166K	875	Yes	-	2133	-

*Shows the lower bound. It does not include the number of structs used by the libraries and kernel.

Table 4.3: Code complexity for popular middleboxes. Those above the line are analyzed in greater detail later.

based key/value store. This enables any middlebox instance to have access to any state, and hence any instance can safely process any packet.

Making the above modifications to middleboxes is difficult because middlebox code is complex. As shown in Table 4.3, several popular middleboxes have between 60K and 275K lines of code (LOC), dozens of different structures and classes, and, in some cases, complex event-based control flow. If a developer misses a change to some structure, class, or function, then output equivalence may be violated under certain input patterns, and a middlebox may fail in unexpected ways at run time. FTMB is the only system that aims to avoid such problems. It automatically modifies middleboxes using LLVM [9]. However, there are two problems: (i) developers must still manually specify which variables may contain/point-to cross-flow state; (ii) the tool is limited to Click-based middleboxes [75].

4.1.3 Simplifying Modification and its Requirements

Making the aforementioned changes to even simple middleboxes can take numerous man-hours as our own experience with OpenNF suggests. This is a serious barrier to adopting any of the previously mentioned systems.

A system that can automatically identify what state a middlebox creates, where the state is created, and how the state is used could be immensely helpful in reducing the man-hours. It can provide developers guidance on writing custom state allocation routines, and on adding appropriate state filtering, serialization, and merging functions. Thus, it would greatly lower the barriers to adopting the above frameworks.

Building such a system is challenging because of *soundness* and *precision* requirements. Soundness means that the system must not miss any types, storage locations, allocations, or uses of state required for output equivalence. A precise system identifies the minimal set of state that requires special handling to ensure state handling at runtime is fast and low-overhead.

4.1.4 Options

Well-known *program analysis* approaches can be applied to identify middlebox state and its characteristics.

Dynamic analysis. We could use dynamic taint analysis [103] to monitor which pieces of state are used and modified while a middlebox processes some sample input. Unfortunately, the sample inputs may not exercise all code paths, causing the analysis to miss some state. We also find that such monitoring can significantly slow middleboxes down (e.g., PRADS [16] and Snort IDS [19] are slowed down $> 10\times$).

Static analysis. Alternatively, we could use symbolic execution [34] or data-/control-flow analysis [49, 63].[1]

Symbolic execution can be employed to explore all possible code paths by representing input and runtime state as a series of symbols rather than concrete values. We can then track the state used in each path. While

[1] Abstract interpretation [39] is another candidate, but it suffers from the well known problem of incompleteness, i.e., it over-approximates the middlebox's processing and may not identify all relevant state.

this is sound, the complexity of most middleboxes (Table 4.3) makes it impossible to explore all execution paths in a tractable amount of time. For example, we symbolically executed PRADS—which has just 10K LOC—for 8 hours using S2E [34], and only 13% of PRAD's code was covered. The complexity worsens exponentially for middleboxes with larger codebases. Recent advances in symbolic execution of middleboxes [44] do not help as they over come state space explosion by abstracting away middlebox state, which is precisely what we aim to analyze.

We make clever use of *data-/control-flow analysis* to automatically evaluate how to handle middlebox state. Naively applying standard data-/control-flow analysis identifies all variables as pertaining to 'state that needs handling' (e.g., variables pertaining to per-packet state, read-only state, and state that falls outside the scope of a flowspace of interest); if developers modify a middlebox to specially handle all these variables, it can result in arbitrarily poor runtime performance during redistribution. We show how *middlebox code structure and design patterns* can be used to design *novel algorithms* that employ *static program analysis* techniques in a way that significantly improves *precision* without compromising *soundness*. Our approach is general and does not assume use of any particular state management framework.

4.2 Overview of StateAlyzr

Most middleboxes' code can be logically divided into three basic parts (Figure 4.4): initialization, packet receive loop, and packet processing. The initialization code runs when the middlebox starts. It reads and parses configuration input, loads supplementary modules or files, and opens log files. All of this can be done in the main() procedure, or in separate procedures called by main. The packet receive loop is responsible for reading a packet (or byte stream) from the kernel (via a socket) and

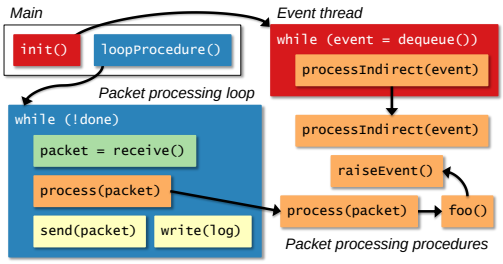


Figure 4.4: Logical structure of middlebox code

passing it to the packet processing procedure(s). The latter analyzes, and potentially modifies, the packet. This procedure(s) reads/writes internal middlebox state to inform the processing of the current (and future) packet.

Our approach consists of three primary stages that leverage this structure. In each stage we further refine our characterization of a middlebox's state. The stages and their main challenges are described next:

1) Identify Per-/Cross-Flow State. In the first stage, we identify the storage location for all per- and cross-flow state created by the middlebox. The final output of this stage is a list of what we call *top-level variables* that contain or indirectly refer to such state.

Unlike state that is only used for processing the current packet, per-/cross-flow state influences other packets' processing. Consequently, the *lifetime* of this state extends beyond the processing a single packet. We leverage this property, along with knowledge of the relation between variable and value lifetimes, to first identify variables that may contain or refer to per-/cross-flow state.

We improve precision by considering which variables are actually *used* in packet processing code, thereby eliminating variables that contain or refer to state that is only used for middlebox initialization. We call the remaining variables "top-level". The main challenge here is dealing with indirect calls to packet processing in event-based middleboxes (Figure 4.4), which complicate the task of identifying all packet processing code. We

develop an algorithm that adapts *forward program slicing* [63] to address this challenge (§4.3.1).

2) Identify Updateable State. The second stage further categorizes state based on whether it may be updated while a packet is processed. If state is read-only, we can avoid repeated cloning (in Pico Replication and OpenNF), avoid unnecessary logging of accesses (FTMB), and allow simultaneous access from multiple instances (StatelessNF); all of these will reduce the frameworks' overhead. We can trivially identify updateable state by looking for assignment statements in packet processing procedures. However, this strawman is complicated by heavy use of pointers in middlebox code which can be used to indirect state update. To address this challenge we show how to employ flow-, context-, and field-insensitive *pointer analysis* [24, 108] (§4.3.2).

3) Identify States' Flowspace Dimensions. Finally, the third stage determines a state's *flowspace*: a set of packet header fields (e.g. src_ip, dest_ip, src_port, dest_port & proto) that delineate the subset of traffic that relates to the state. Flowspace must be considered when modifying a middlebox to use custom allocation functions [96, 98] or filter state in preparation for export [53]. It is important to avoid the inclusion of irrelevant header fields and the exclusion of relevant fields in a state's flowspace, because it impacts runtime correctness and performance, respectively. To solve this problem we developed an algorithm that leverages common state access patterns in middleboxes to identify program points where we can apply *program chopping* [99] to determine relevant header fields (§4.3.3).

Soundness. In order for StateAlyzr to be sound it is necessary for these three stages to be sound. In Appendix A.1, we prove the soundness of our algorithms.

Assumptions about middlebox code. Our proofs are based on the assumption that middleboxes use standard API or system calls to read/write packets and hashtables or link-lists to store state. These assumption are

not limitations of our analysis algorithms. Instead, they are made to ease the implementation of StateAlyzr. Our implementation can be extended to add additional packet read/write methods or other data structures to store the state.

4.3 StateAlyzr Foundations

We now describe our novel algorithms for detailed state classification. To describe the algorithms, we use the example of a simple middlebox that blocks external hosts creating too many new connections (Figure 4.3).

```
1  struct host {
2    uint ip;
3    int count;
4    struct host *next;
5  }
6
7  pcap_t *intPcap, *extPcap;
8  int threshold;
9  char * queue[100];
10 int head = 0, tail = 0;
11 struct host *hosts = NULL;
12
13 int main(int argc, char **argv) {
14   pthread_t thread;
15   intPcap = pcap_create(argv[0]);
16   extPcap = pcap_create(argv[1]);
17   threshold = atoi(argv[2]);
18   pthread_create (&thread,(void*)&processPacket);
19 }
20
21 int loopProcedure() {
22   while(1) {
23     struct pcap_pkthdr pcapHdr;
24     char *pkt = pcap_next(extPcap, &pcapHdr);
25     ifFull_Wait();
26     enqueue(pkt);
27     if (entry->count < threshold)
28       pcap_inject(intPcap, pkt, pcapHdr->caplen);
29 } }
```

```
30
31 void enqueue(char* pkt){
32   head = (head + 1)%100;
33   queue[head] = pkt;
34 }
35
36 char* dequeue(){
37    int *index = &tail;
38    *index = (*index + 1)%100;
39    return queue[*index];
40 }
41
42 void processPacket(){
43   while(1){
44     ifEmpty_Wait();
45     char* pkt = dequeue();
46     struct ethhdr *ethHdr = (struct ethhdr)pkt;
47     struct iphdr *ipHdr = (struct iphdr*)(ethHdr+1);
48     struct tcphdr *tcpHdr = (struct tcphdr*)(ipHdr+1);
49     struct host *entry = lookup(ipHdr->saddr, hosts);
50     if (NULL == host){
51       struct host *new = malloc(sizeof(struct host));
52       new->ip = ipHdr->saddr;
53       new->next = hosts;
54       hosts = new;
55     }
56     if (tcpHdr->syn && !tcpHdr->ack)
57       entry->count++;
58 }  }
59
60 struct host *lookup(uint ip) {
61   struct host *curr = hosts;
62   while (curr != NULL) {
63     if (curr->ip == ip)
64       return curr;
65     curr = curr->next;
66 } }
```

Figure 4.5: Code for our running example.

4.3.1 Per-/Cross-Flow State

Our analysis begins by identifying the storage location for all relevant per- and cross-flow state created by the middlebox. This has two parts: (*i*) exhaustively identifying persistent variables to ensure soundness, and (*ii*) carefully limiting to top-level variables that contain or refer to per-/cross-flow values to ensure precision.

Identifying Persistent Variables

Because per-/cross-flow state necessarily influences two or more packets within/across flows, values corresponding to such state must be created during or prior to the processing of one packet, and be destroyed during or after the processing of a subsequent packet. Hence, the corresponding variables must be *persistent*, i.e., their values persist beyond a single iteration of the packet processing loop. In Figure 4.3, variables declared on lines 7 to 11 are persistent, whereas curr on line 61 is not. Our algorithm first identifies such variables.

```
   Input: prog
   Output: persistVars
1  persistVars = {}
2  persistVars = persistVars ∪ GlobalVarDecls(prog)
3  foreach proc in Procedures(prog) do
4     persistVars = persistVars ∪ StaticVarDecls(proc)
5  persistVars = persistVars ∪ LocalVarDecls(loopProc)
6  persistVars = persistVars ∪ FormalParams(loopProc)
```

Figure 4.6: Identifying persistent variables

Analysis Algorithm. We traverse a middlebox's code, as shown in Figure 4.6. The values of all global and static variables exist for the entire duration of the middlebox's execution, so these variables are always persis-

tent. Variables local to the *loop-procedure*[2]—i.e., the procedure containing the packet processing loop—exist for the duration of this procedure, and hence the duration of the packet processing loop, so they are also persistent.

Local variables of procedures that precede the loop-procedure on the call stack are also persistent, because the procedures' stack frames last longer than the packet processing loop. However, these variables cannot be used within the packet processing loop, or a procedure called therein, because the variables are out of scope. Thus we exclude these from our list of persistent variables, improving precision.

The above analysis implicitly considers heap-allocated values by considering the values of global, static, and local variables, which can point to values on the heap. Values on the heap exist until they are explicitly freed (or the middlebox terminates), but their *usable lifetime* is limited to the time frame in which they are reachable from a variable's value.[3] Therefore, we can conclude that a heap-allocated value's persistence is predicated on the persistence of a variable identified by our algorithm.

Limiting to Top-level Variables

The above algorithm identifies a superset of variables that may be bound, or point, to per-/cross-flow state. It includes variables bound to state used in initialization for loading/processing configuration/signature files: e.g., variables intPcap and extPcap in Figure 4.3. Such variables don't need handling during traffic redistribution; they can simply be copied when an instance is launched. To eliminate such variables and improve precision, the key insight we leverage is that, by definition, per-/cross-flow state is *used* in some way during packet processing. However, identifying all such variables is non-trivial, and missing variables impact analysis soundness.

[2]To automatically detect packet processing loops, we use the fact that middleboxes

```
   Input: prog, persistVars
   Output: pktProcs, percrossflowVars
 1 pktProcs = {}
 2 sdg = SystemDependenceGraph(prog)
 3 foreach stmt in Statements(loopProc) do //Statements() returns all
   statements in a procedure
 4   if stmt calls PKT_RECV_FUNC then
 5     slice = ForwardSlice(sdg, stmt, stmt.LHS)
 6     pktProcs = pktProcs ∪ Procedures(slice) //Procedures() returns all
       procedures in a slice
 7 percrossflowVars = {}
 8 foreach proc in pktProcs do
 9   foreach stmt in Statements(proc) do
10     foreach var in Vars(stmt) do
       //Vars() returns all variables used in a statement
11       if var in persistVars then
12         percrossflowVars = percrossflowVars ∪ {var}
```

Figure 4.7: Identifying per-/cross-flow variables

Identifying Packet Processing Procedures. Figure 4.7 shows our algorithm for identifying top-level variables that contain or refer to per-/cross-flow values. The first half of the algorithm (lines 1–6) focuses on identifying packet processing code. Obviously any code contained in the packet processing loop is used for processing packets, but, crucially, the code of procedures (indirectly) called from within the loop is also packet processing code.

We considered a strawman approach of using call graphs to identify packet processing procedure. A call graph is constructed by starting at each procedure call within the packet processing loop, and classifying each appearing procedure as a packet processing procedure. However, this analysis does not capture packet processing procedures that are called indirectly. The Squid proxy, e.g., does initial processing of the received

read packets using standard library/system functions.

[3] A heap value whose lifetime is longer than its usable lifetime is a memory leak.

packet, then enqueues an event to trigger further processing through later calls to additional procedures. Hence the analysis may incorrectly eliminate some legitimate per-/cross-flow state which is used in such procedures.

Thus, we need an approach that exhaustively considers the dependencies between the receipt of a packet and both direct and indirect invocations of packet processing procedures. Below, we show how *system dependence graphs* [49] and *program slicing* [63] can be used for this.

A *system dependence graph* (SDG) consists of multiple program dependence graphs (PDGs) — one for each procedure. Each PDG contains vertices for each statement along with their data and control dependency edges. A *data dependence* edge is created between statements p and q if there is an execution path between them, and p may update the value of some variable that q reads. A *control dependence* edge is created if p is a conditional statement, and whether or not q executes depends on p. A snippet of the control and data edges for our example in Figure 4.3 is in Figure 4.8.

Whereas control edges capture direct invocations of packet processing, we can rely on data edges to capture indirect procedure calls. For example, the dashed yellow lines in Figure 4.8 fail to capture invocation of the processPacket procedure on bottom right (because there is no control edge from the while loop or any of its subsequent procedures to processPacket). In contrast, we can follow the data edges, the dashed red line, to track such calls.

Given a middlebox's SDG, we compute a *forward program slice* from a packet receive function call for the variable which stores the received packet. A forward slice contains the set of *statements that are affected* by the value of a variable starting from a specific point in the program [63]. Most middleboxes use standard library/system functions to receive packets—e.g., pcap_next, or recv—so we can easily identify these calls and the vari-

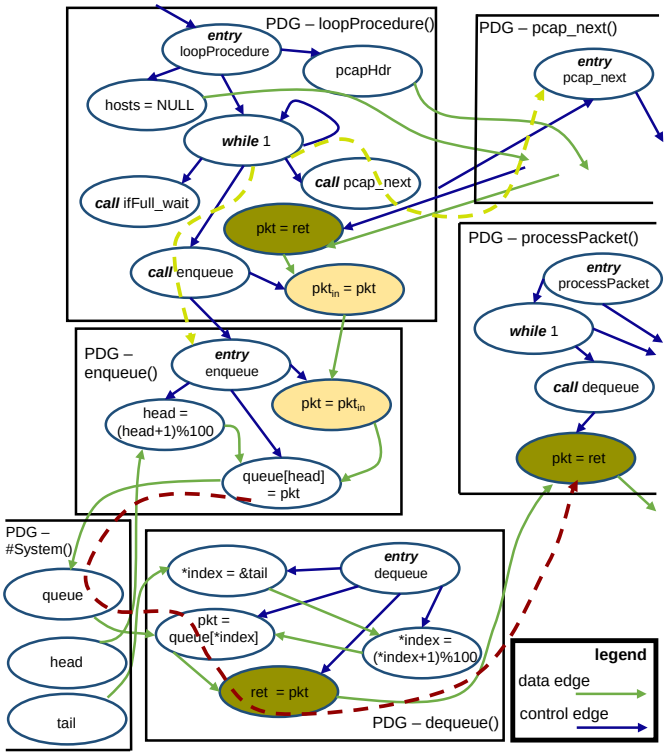


Figure 4.8: Snippet of System dependence graph (SDG) for the code in Figure 4.3; green edges indicate data dependencies and blue edges indicate control dependencies; light yellow nodes represent formal and actual parameters, while dark yellow nodes represent return values.

able pointing to the received packet. We consider any procedure appearing in the computed slice to be a packet processing procedure. For middleboxes which invoke packet receive functions at multiple points, we compute forward slices from every call site and take the union of the procedures appearing in all such slices.

Values Used in Packet Processing Procedures. The second half of our algorithm (Figure 4.7, lines 7–12) focuses on identifying persistent values that are used within some packet processing procedure. We analyze each statement in the packet processing procedures. If the statement contains a persistent variable, then we mark that persistent variable as a top-level

variable.

4.3.2 Updateable State

Next, we delineate *updateable* top-level variables from *read only* variables to further improve precision. In Figure 4.3, variable head, tail, hosts and queue are updateable, whereas threshold is not. Because state is updated through assignment statements, one strawman choice here is to statically identify top-level variables on the left-hand-side (LHS) of assignment statements. In Figure 4.3, this identifies head, hosts and queue.

However, this falls short due to *aliasing*, where multiple variables are bound to the same storage location due to the use of pointers [35]. Aliasing allows a value reachable from a top-level variable to be updated through the use of a different variable. Thus our strawman can mis-label top-level variables as read-only, compromising soundness. For example, tail is mis-labeled in Figure 4.3, because it never appears on the LHS of assignment statements. But on line 38 index is updated which points to tail.

```
   Input: pktProcs, percrossflowVars
   Output: updateableVars
1  percrossflowVars = {}
2  foreach proc in pktProcs do
3    foreach stmt in AssignmentStmts(proc) do //AssignmentStmts() returns all
     assignment statements in a procedure
4      foreach var in percrossflowVars do
5        if stmt.LHS == var
           or var in PointsTo(stmt.LHS)
           or PointsTo(var) ∩ PointsTo(stmt.LHS) ≠ ∅ then
6          updateableVars = updateableVars ∪ {var }
```

Figure 4.9: Identifying updateable variables

Analysis Algorithm. We develop an algorithm to identify *updateable top-level variable* (Figure 4.9). Since we are concerned with variables whose

(referenced) values are updated during packet processing, we analyze each assignment statement contained in the packet processing procedures identified in the first stage of our analysis (§12). If the assignment statement's LHS contains a top-level variable, then we mark the variable as updateable (similar to our strawman). Otherwise, we compute the *points-to* set for the variable on the LHS and compare this with the set of updateable top-level variables and their points-to sets. A variable's points-to set contains all variables whose associated storage locations are reachable from the variable. To compute this set, we employ flow-, context-, and field-insensitive pointer analysis [24]. If the points-to set of the variable on the LHS contains a top-level variable, or has a non-null intersection with the points-to set of a top-level variable, then we mark the top-level variable as updateable.

Due to limitations of pointer analysis, our algorithm may still mark read-only top-level variables as updateable. E.g., field insensitive pointer analysis can mark a top-level struct variable as updateable even if just one of its sub-fields is updateable.

4.3.3 State Flowspaces

Finally, we identify the packet header fields that define the flowspace associated with the values of each top-level variable. Identifying too fine-grained of a flowspace for a value—i.e., more header fields than those that actually define the flowspace—is unsound; such an error will cause a middlebox to incorrectly filter out the value when it is requested by a middlebox state management framework [53, 69, 96, 98]. Contrarily, assuming an overly permissive flowspace (e.g., the entire flowspace) for a value hurts precision.

To identify flowspaces, we leverage common middlebox design patterns in updating or accessing state. Middleboxes typically use simple data structures (e.g., a hash table or linked list) to organize state of the

same type for different network entities (connections, applications, subnets, URLs, etc.). When processing a packet, a middlebox uses header fields[4] to lookup the entry in the data structure that contains a reference to the values that should be read/updated for this packet. In the case of a hash table, the middlebox computes an *index* from the packet header fields to identify the entry pointing to the relevant values. For a linked list, the middlebox *iterates* over entries in the data structure and compares packet header fields against the values pointed to by the entry.

```
   Input: pktProcs, percrossflowVars
   Output: chop, flowspace
 1 keyedVars = {}
 2 foreach var in percrossflowVars do
 3   if Type(var) == pointer
       or Type(var) == struct then
 4     keyedVars = keyedVars ∪ {keyedVars}
 5 foreach proc in pktProcs do
 6   foreach loopStmt in LoopStmts(proc) do
 7     condVars = {}
 8     foreach var in Vars(loopStmt.condition) do
 9       if var in keyedVars
           or PointsTo(var) ∩ keyedVars ≠ ∅ then
10         for condStmt in ConditionalStmts(loopStmt.body) do
11           for condVar in Vars(condStmt) do
12             if condVar ≠ var then
13               condVars = condVars ∪ {condVar}
14     chop = Chop(sdg,pktVar,condVars)
15     flowspace = ExtractFlowspace(chop)
```

Figure 4.10: Identifying packet header fields that define a per-/cross-flow variable's associated flowspace

Algorithm. We leverage the above design patterns in our algorithm shown in Figure 4.10. In the first step (lines 2-4), if the top-level variable is a struct

[4]In cases where keys are not based on the packet header fields e.g. URL, a middlebox usually keeps another data structure to maintain the mapping between such keys and packet header fields

or a pointer, we mark it as a possible candidate for having a flowspace associated with it. This filters out all the top-level variables which cannot represent more than one entry; e.g., variables head and tail in Figure 4.3.

We assume that middleboxes use hash tables or linked lists to organize their values,[5] and that these data structures are accessed using:
square brackets, e.g.

```
entry = table[index];
```

pointer arithmetic, e.g.

```
entry = head + offset;
```

or iteration[6], e.g.

```
while(entry->next!=null){entry=entry->next;}
for(i=0; i<list.length; i++) {...}
```

The second step is thus to identify all statements like these where a top-level variable marked above is on the right-hand-side (RHS) of the statement (square brackets or pointer arithmetic scenario) or in the conditional expression (iteration scenario).

When square brackets or pointer arithmetic are used, we compute a *chop* between the variables in the access statement and the variable containing the packet returned by the packet receive procedure. A chop between a set of variables U at program point p and a set of variables V at program point q is the set of statements that (i) may be affected by the value of variables in U at point p, and (ii) may affect the values of variables in V at point q. Thus, the chop we compute above is a snippet of executable code which takes a packet as input and outputs the index or offset required to extract the value from the hashtable.

In a similar fashion, when iteration is used, we identify all conditional statements in the body of the loop. We compute a chop between the packet returned by the packet receive procedure and the set of all the

[5]Our approach can easily be extended to other data structures.

[6]Middleboxes may also use recursion, but we have not found this access pattern in the middleboxes we study, so we do not consider it in our algorithm.

variables in the conditional expression which do not point to any of the top-level variables; in our example (Figure 4.3), the chop starts at line 24 and terminates at line 63. We output the resulting chops, which collectively contain all conditional statements that are required to lookup a value in a linked list data structure based on a flow space definition. Assuming that the middlebox accesses packet fields using standard system-provided structs (e.g., struct ip as defined in netinet/ip.h), we conduct simple string matching on the code snippets to produce a list of packet header fields that define a state's flowspace.

4.4 Enhancements

Data and control flow analysis can help improve precision, but they have some limitations in that they cannot guarantee that exactly the relevant state and nothing else has been identified. In particular, static analysis cannot differentiate between multiple memory regions that are allocated through separate invocations of malloc from the same call site. Therefore, we cannot statically determine if only a subset of these memory regions have been updated after processing a set of packets. To overcome potential efficiency loss due to such limitations, we can employ custom algorithms that boost precision in specific settings. We present two candidates below.

4.4.1 Output-Impacting State

In addition to the three main code blocks (Figure 4.4), middleboxes may optionally have packet and log output functions. These pass a packet to the kernel for forwarding and record the middlebox's observations and actions in a log file, respectively. These functions are usually called from within the packet processing procedure(s).

In some cases, operators may desire output equivalence only for specific types of output. For example, an operator may want to ensure client

connections are not broken when a NAT fails—i.e., packet output should be equivalent—but may not care if the log of NAT'd connections is accurate. In such cases, internal state that only impacts non-essential forms of output does not need special handling during redistribution and can be ignored.

To aid such optimizations, we develop an algorithm to identify the type of output that updateable state affects. We use two key insights. First, middleboxes typically use *standard libraries and system calls* to produce packet and log output: either PCAP (e.g. pcap_dump) or socket (e.g. send) functions for the former, and regular I/O functions (e.g. write) for the latter.[7] Second, the output produced by these functions can only be impacted by a *handful of parameters* passed to these functions. Thus, we focus on the call sites of these functions, and their parameters.

```
   Input: sdg, updateableVars
   Output: pktoutputVars, logoutputVars
 1 pktoutputVars = {}
 2 logoutputVars = {}
 3 foreach proc in pktProcs do
 4   foreach stmt in Statements(proc) do
 5     if stmt calls PKT_OUTPUT_FUNC
         or stmt calls LOG_OUTPUT_FUNC then
 6       slice = BackwardSlice(sdg, stmt,
                 Vars(stmt.RHS))
 7       foreach sliceStmt in Statements(slice) do
 8         foreach var in Vars(sliceStmt) do
 9           if var in updateableVars then
10             if stmt calls PKT_OUTPUT_FUNC then
11               pktoutputVars = pktoutputVars ∪ {var}
12             else
13               logoutputVars = logoutputVars ∪ {var}
```

Figure 4.11: Identifying output-impacting variables

Algorithm. We use *program slicing* [63] to identify the dependencies be-

[7] Our approach can be easily extended to consider non-standard output functions.

tween a specific type of output and updateable variables. Our algorithm is shown in Figure 4.11. We first identify the call sites of packet or log output functions by checking each statement in each packet processing procedure (§12). Then we use the SDG produced in the first stage of our analysis (Figure 4.7) to compute a *backward slice* from each call site. Such a slice contains the set of statements that affect (*i*) whether the procedure call is executed, and (*ii*) the value of the variables used in the procedure call, such as the parameters passed to the output function. We examine each statement in a backward slice to determine whether it contains an updateable per-/cross-flow variable. Such variables are marked as impacting packet (or log) output.

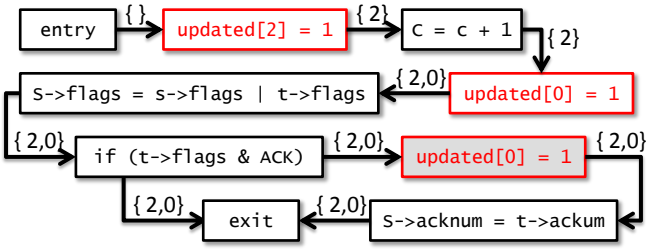
4.4.2 Tracking Runtime Updates

Developers aiming to design fault-tolerant middleboxes can use the algorithms in §4.3 and §4.4.1 to efficiently clone state to backup instances. For example, if traffic will be distributed among multiple instances in the case of failure, then only state whose flowspace overlaps with that assigned to a specific instance needs to be cloned to that instance. However, the potential performance gains from these optimizations may be limited due to constraints imposed by data/control-flow analysis. For example, our analysis can only identify whether a persistent variable's value *may* be updated during the middlebox's execution. If we can determine at runtime exactly which values are updated, and when, then we can further improve the efficiency of state cloning and speed up failover.

To achieve higher precision, we must use (simple) run time monitoring. For example, we can track, at run time, whether part of an object is updated during packet processing. To implement this monitoring, we must modify the middlebox to set an "updated bit" whenever a value reachable from a top-level variable is updated during packet processing. Figure 4.12a shows such modifications, in red, for a simple middlebox. We create a unique

```
1  struct conn tbl[1000]; // Assigned id 0
2  int count; // Assigned id 1
3  int tcpcnt; // Assigned id 2
4  char updated[3];
5  void main() {
6    while(1) {
7      char *pkt = recv();
8      updated[1] = 1;
9      count = count + 1;
10     struct *iphdr i = getIpHdr(pkt);
11     if (i->protocol == TCP) {
12       hdl(&tcpcnt, &tbl[hash(pkt)], getTcpHdr(pkt));
13 } } }
14 void hdl(int *c,struct conn *s,struct tcphdr *t) {
15   updated[2] = 1;
16   c = c + 1;
17   updated[0] = 1;
18   s->flags = s->flags | t->flags;
19   if (t->flags & ACK)
20     updated[0] = 1; // Pruned
21     s->acknum = t->acknum;
22 } }
```

(a) Example middlebox code instrumented for update tracking at run time; statements in red are inserted based on our analysis



(b) Annotated control flow graph used for pruning redundant updated-bit-setting (shaded) statements

Figure 4.12: Implementing update tracking at run time

updated bit for each top-level variable—there are three such variables in the example—and we set the appropriate bit before any statement that updates a value that may be reachable from the corresponding variable.

We use the same analysis discussed in §4.3.2 to determine where to insert statements to set updated bits. For any statement where a top-level variable is updated, we insert a statement—just prior to the assignment statement—that sets the appropriate updated bit.

However, this approach can add a lot more code than needed: if one assignment statement *always* executes before another, and they *always* update the same value, then we only need to set the updated bit before the first assignment statement. For example, line 21 in Figure 4.12a updates the same compound value as line 18, so the code on line 20 is redundant.

We use a straightforward control flow analysis to prune unneeded updated-bit-setting statements. First, we construct a control flow graph (CFG) for each modified packet processing procedure. Next, we perform a depth-first traversal of each CFG, tracking the set of updated bits that have been set along the path; as we traverse each edge, we label it with the current set of updated bits. Figure 4.12b shows this annotated CFG for the handleTcp procedure shown in lines 14-22 of Figure 4.12a. Lastly, for each updated-bit-setting statement in a procedure's CFG, we check whether the bit being set is included in the label for every incoming edge. If this is true, then we prune the statement; e.g., we prune line 20 in Figure 4.12a.

4.5 Implementation

We implement StateAlyzr using CodeSurfer [3] which has built-in support for constructing CFGs, performing flow- and context-insensitive pointer analysis, constructing PDGs/SDGs, and computing forward/backward slices and chops for C/C++ code. CodeSufer uses proven sound algorithms to implement these static analysis techniques. We use CodeSurfer's

Scheme API to access output from these analyses in our algorithms. We applied StateAlyzr to four middleboxes: PRADS asset monitoring [16] and Snort Intrusion Detection System [19], HAproxy load balancer [5], and OpenVPN gateway [15].

Fault Tolerance. We use the output from StateAlyzr to add fault tolerance to PRADS and Snort, both off-path middleboxes. We added code to both to export/import internal state (to a standby). We used the output of our first two analysis phases (§4.3.1 and §4.3.2) to know which top level variables' values we need to export, and where in a hot-standby we should store them. We used the output of our third analysis phase (§4.3.3) as the basis for code that looks up per-/cross-flow state values. This code takes a flowspace as input and returns an array of serialized values. We use OpenNF [53] to transfer serialized values to a hot-standby. Similarly, import code deserializes the state and stores it in the appropriate location. We also implemented both enhancements discussed in §4.4.

4.6 Evaluation

We report on the outcomes of applying StateAlyzr to four middleboxes. We address the following questions:

- *Effectiveness*: Does StateAlyzr help with making modifications to today's middleboxes? How many top-level variables do these middleboxes maintain, relative to all variables? What relative fractions of these pertain to state that may need to be handled during redistribution? How precise is StateAlyzr?

- *Runtime efficiency and manual effort:* To what extent do StateAlyzr's mechanisms help improve the runtime efficiency of state redistribution? How much manual effort does it save?

Mbox	All	Persistent	Top -level	Update -able	pkt/log output impacting	require serialization
PRADS	1529	61	29	10	N.A. / 6	14
Snort	18393	507	333	148	N.A. /148	176
HAproxy	7876	272	176	115	101 / 109	59
OpenVPN	8704	156	131	106	97 / 102	8

Table 4.13: Variables and their properties

- *Practical considerations*: Does StateAlyzr take prohibitively long to run (like symbolic execution; §4.1.4)? Is it sound in practice?

4.6.1 Effectiveness

In Table 4.13, we present a variety of key statistics derived for the four middleboxes using StateAlyzr. We use this to highlight StateAlyzr's ability to improve precision, thereby underscoring its usefulness for developers.

The complexity of middlebox code is underscored by the overall number of variables in Table 4.13, which can vary between 1500 and 18k, and other relevant code complexity metrics shown in Table 4.3. Thus, manually identifying state that needs handling, and optimizing its transfer, is extremely difficult.

We also note from Table 4.13 that StateAlyzr identifies 61-507 variables as persistent across the four middleboxes. A subset of these, 29-333, are top-level variables. Finally, 6-148 top-level variables are updateable; operators only have to deal with handling the values pertaining to these variables at run time. Snort is the most complex middlebox we analyze ($\approx$275K lines of code) and has the largest number of top-level variables (333); the opposite is true for PRADS (10K LOC and 29 top-level variables).

The drastic reduction to the final number of updateable variables shows that naive approaches that attempt to transfer/clone values corresponding to *all* variables can be very inefficient at runtime. (We show this empirically in §4.6.2.) Even so, the number of updateable variables can be as high as 148, and attempting to manually identify them and argument code suit-

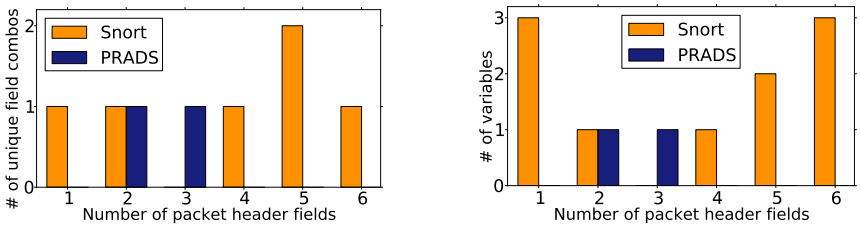


Figure 4.14: Flowspace dims. of keyed per-/cross-flow vars

ably can be very difficult. By automatically identifying them, StateAlyzr simplifies modifications; we provide further details in §4.6.2.

Finally, the reductions we observe in going from persistent variables to top-level variables (16-53% reduction) and further to updateable ones (19-65% reduction) show that our techniques in §4.3.1 and §4.3.2 offer useful improvements in precision.

In Figure 4.14, we characterize the flowspaces for the variables found in Snort and PRADS. From the left figure, we see that Snort maintains state objects that could be keyed by as many as 5 or 6 header fields; the maximum number of such fields for PRADS is 3. The figure on the right shows the number of variables that use a particular number of header fields as flowspace keys; for instance, in the case of Snort, 3 variables each are keyed on 1 and 6 fields. The total number of variables keyed on at least one key is 2 and 10 for Snort and PRADS, respectively (sum of the heights of the respective bars).

These numbers are significantly lower than the updateable variables we discovered for these middleboxes (6 and 148, respectively). Digging deeper into Snort (for example) we find that:

- 111 updateable variables pertain to all flows (i.e., a flowspace key of "*"). Of these, 59 variables are related to configurations and signatures, while 30 are function pointers (that point to different detection and processing plugins). These 89 variables can be updated from the

command line at middlebox run time (when an operator provides new configurations and signatures, or new analysis plugins).

- 27 updateable variables—or 18%—are only used for processing a *single packet*; hence they don't correspond to per-/cross-flow state. This points to StateAlyzr's *imperfect precision*. These variables are global in scope and are used by different functions for processing a single incoming packet, which is why our analysis labels them as updateable. A developer can easily identify these variables and can either remove them from the list of updateable variables or modify code to make them local in scope.

4.6.2 Runtime efficiency and manual effort

Fault Tolerant Middleboxes

Using fault tolerant PRADS/Snort versions (§4.4), we show that StateAlyzr helps significantly cut unneeded state transfers, improving state operation time/overhead.

Man-hours needed. Modifying PRADS based on StateAlyzr analysis took roughly 6 man-hours, down from over 120 man-hours when we originally modified PRADS for OpenNF (Two different persons made these modifications.). Modifying Snort, a much more complex middlebox, took 90 man-hours. In both cases, most of the time ($> 90\%$) was spent in writing serialization code for the data structures identified by StateAlyzr (14 for PRADS and 176 for Snort; Table 4.13). Providing support for exporting/importing state objects according to OpenNF APIs took just 1 and 2 hours, respectively.

Runtime benefits. We consider a primary/hot standby setup, where the primary sends a copy of the state to the hot standby after processing each packet. We use a university-to-cloud packet trace [59] with around 700k packets for our trace-based evaluation of this setup. The primary

instance processes the first half of the trace file until a random point, and the hot standby takes over after that. We consider three models for operating the hot standby which reflect progressive application of the different optimizations in §4.3 and §4.4: (i) the primary instance sends a copy of all the updateable states to the hot standby, (ii) the primary instance only sends the state which applies to the flowspace of the last processed packet, and (iii) in addition to considering the flowspace, we also consider which top level variables are marked as updated for the last processed packet.

Figure 4.15a shows the average case results for the amount of per packet data transferred between the primary and secondary instances for all three models for PRADS. Transferring state which only applies to the flowspace of the last processed packet, i.e., the second model, reduces the data transferred by 305× compared to transferring all per-/cross-flow state. Furthermore, we find that the third model, i.e., run time marking of updated state variables, further reduces the amount of data transferred by 2×, on average. This is because not all values are updated for every packet: the values pertaining to a specific connection are updated for every packet of that connection, but the values pertaining to a particular host and its services are only updated when processing certain packets. This behavior is illustrated in Figure 4.15b, which shows the size of the state transfer after processing each of the first 200 packets in a randomly selected flow.

We measured the increase in per packet processing time purely due to the code instrumentation needed to identify state updates for highly available PRADS. We observed an average increase of 0.04µsec, which is around 0.14% of the average per packet processing time for unmodified PRADS.

Figure 4.16 shows the corresponding results for Snort. Transferring just the updateable state results in a 8800× reduction in the amount of state transferred compared to transferring all per-/cross-flow state. This

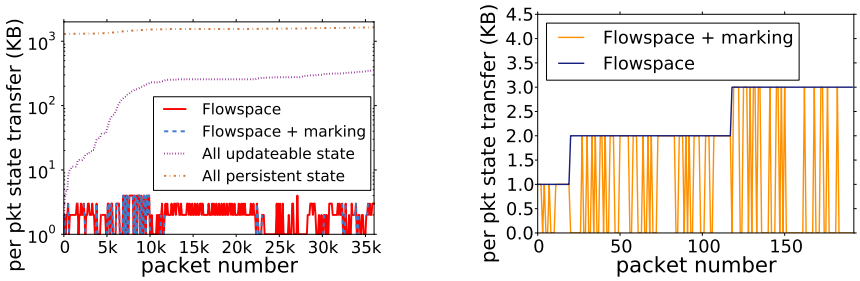


Figure 4.15: (a) Per packet state transfer (b) Per packet state transfer for a single connection

is because, a significant portion of the persistent state in Snort consists of configuration and signatures which are never updated during packet processing. Transferring state which only applies to a particular flowspace further reduces the data transfer by 2.75×. Unlike PRADS, the amount of state transfer in the second model remains constant for a particular flow because most of the state is created on the first few packets of a flow. Finally, runtime marking further reduces the amount of state transferred by 3.6×.

Packet/Log Output

Table 4.13 includes the number of variables that impact packet or log output. For on-path HAproxy (OpenVPN), 87% (91%) of updateable variables affect packet output; a slightly higher fraction impact log output. 95 (93) variables impact both outputs. A much smaller number impacts packet output but not log (6 and 4, respectively). Another handful impact logs but not packets (14 and 9); operators who are interested in just packet output consistency can ignore transferring the state pertaining to these variables, but the benefit will likely not be significant for these middleboxes given the low counts.

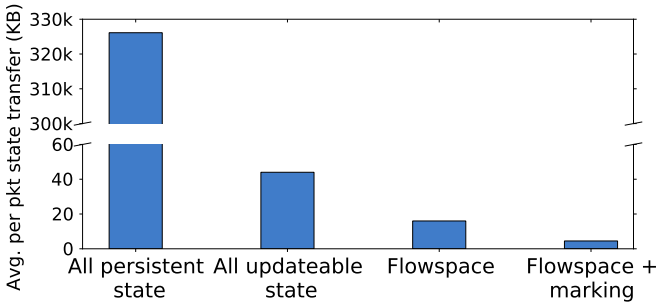


Figure 4.16: Per packet state transfer in Snort

Being off-path, PRADS and Snort have no variables that impact packet output. For PRADS, 6 out of 10 updateable variables impact log output. StateAlyzr did find 4 other updateable variables—tos, tstamp, in_pkt, and mtu—but did not mark them as affecting packet output or log output. Upon manual code inspection we found that these values are updated as packets are processed, but they are never used; thus, these variables can be removed from PRADS without any impact on its output, pointing to another benefit of StateAlyzr—code clean-up.

4.6.3 Practicality

Table 4.17 shows the time and resources required to run our analysis. CodeSurfer computes data and control dependencies and points-to sets at compile time, so the middleboxes take longer than normal to compile. This phase is also memory intensive, as illustrated by peak memory usage. Snort, being complex, takes the longest to compile and analyze (≈20.5h). This is not a concern since StateAlyzr only needs to be run once, and it runs offline.

Mbox	Compile time	Analysis time	Memory
PRADS	0.2	0.25	0.3
Snort	1.5	19	6
HAproxy	0.25	6	6
OpenVPN	0.5	5	7.3

Table 4.17: Time (h) and memory usage (GB)

Empirically Verifying Soundness

Empirically showing soundness in practice is hard. Nevertheless, for the sake of completeness, we use two approaches to verify soundness of the modifications we make on the basis of StateAlyzr's outputs.

First, we use the experimental harness from §4.6.2. We compare logs at PRADS/Snort in the scenario where a single instance processes the complete trace file against concatenated logs of the primary and hot standby, using the trace and the three models as above. In all cases, there was no difference in the two sets of logs.

Next, we compare with manually making all changes. Recall that we had manually modified PRADS to make it OpenNF-compliant. We compared StateAlyzr's output for PRADS against the variables contained in the state transfer code we added during our prior modifications to PRADS. StateAlyzr found all variables we had considered in our prior modifications, and more. Specifically, we found that our prior modifications had *missed* an important compound value that contains a few counters along with configuration settings.

4.7 Other Related Work

Aside from the works discussed in §4.1 and §4.3 [24, 53, 63, 87, 96–98, 101, 108, 116] StateAlyzr is related to a few other efforts. Some prior studies have focused on transforming non-distributed applications into distributed

applications [64, 110]. However, these works aim to run different parts of an application at different locations. We want all analysis steps performed by a middlebox instance to run at one location, but we want different instances to run on a different set of inputs without changing the collective output from all instances.

Dobrescu and Argyarki use symbolic execution to verify middlebox code satisfies crash-freedom, bounded-execution, and other safety properties [44]. They employ small, Click-based middleboxes [75] and abstract away accesses to middlebox state. In contrast, our analysis focuses on identifying state needed for correct middlebox operation and works with regular, popular middleboxes.

Lorenzo et al. [40] use similar static program analysis techniques to identify flowspace, but their identification is limited to just hashtables.

4.8 Summary

Our goal was to aid middlebox developers by identifying state objects that need explicit handling during redistribution operations. In comparison with today's manual and necessarily error-prone techniques, our program analysis based system, StateAlyzr, vastly simplifies this process, and ensures soundness and high precision. Key to StateAlyzr is novel state characterization algorithms that marry standard program analysis tools with middlebox structure and design patterns. StateAlyzr results in nearly 20$\times$ reduction in manual effort, and can automatically eliminate nearly 80% of variables in middlebox code for consideration during framework-specific modifications, resulting in dramatic performance and overhead improvements in state reallocation. Ultimately, we would like to fully automate the process of making middlebox code framework-compliant, thus fulfilling the promise of using NFV effectively for middlebox elasticity and fault tolerance. Our work addresses basic challenges in code analysis,

a difficult problem on its own which is necessary to solve first.

5 Conclusion and Future Work

In this dissertation, we designed and evaluated three systems which can enable faster adoption of NFV. In this closing chapter, we summarize our key contributions and present directions for future research that can help further in paving the way towards the adoption of NFV.

5.1 Iron

In Chapter 2, we showed that computational overhead associated with the network stack of Linux can break isolation in containerized environments which can result in a slowdown as high as 6×. We designed Iron [74] to enforce stronger isolation by accurately measuring the time spent in servicing softirq and integrating it with the OS scheduler. We also proposed a novel hardware based packet dropping mechanism.

Following are some avenues for future work for extending Iron:

Applications with kernel bypass: Kernel bypass based techniques do not have the problem of network based processing not properly charged. However, these approaches can still leverage the accounting aspect of Iron to share recourses appropriately in the userspace. Such approaches make it difficult for admins to insert policies and functionalities at the host (e.g. traffic shaping). Admins are left with either admin-controlled software or leveraging smart-NICs to enforce policies. Though it is not clear which policy aspects can be handled in the software and which aspects can be offloaded to the NIC.

Other softirqs: While Iron just focuses on network-based softirq processing, however, same high level principles can be extended and applied to other softirqs. Future work might deal with developing low-overhead mechanisms for other interrupts

5.2 CHC

In Chapter 3, we presented a ground-up design of an NFV framework to support COE and high performance for NFV chains. CHC [72] relies on externalizing the internal state of NFs along with several novel state caching and update algorithms to ensure low packet processing latency and high throughput. In addition to this, it provides support for elastic scaling and fault tolerance.
FaaS: As a future direction, CHC can be extended to support the model of function as a service(FaaS). In such a model, NF instances are spun up only when there is traffic to process and are charged on pay-for-what-you-use basis. Such frameworks will be responsible for ensuring highly flexible scaling, fault tolerance and near negligible startup time.

5.3 StateAlyzr

In Chapter 4, we introduced StateAlyzr [73] that simplifies the process of modifying NF code by identifying and categorizing NF states, which requires special handling, to ensure consistent NF action, during redistribution of traffic. StateAlyzr embodies novel algorithms adopted from program analysis and NF code structure to provably and automatically identify such states. StateAlyzr has made an important headway towards automating the process of modifying NFs, there are several more challenges which need to be addressed to fully automate this process.

Serialization of state objects: While StateAlyzr significantly reduces the manual effort required to modify an NF code. It does not fully automate that process, in particular requires the developer to write code for serialization and deserialization of state objects. One possible future research direction is to automate this process of generating code for serialization and deserialization.

Simplifying state objects: Our analysis of NF code shows that there is a heavy use of multi-level pointers for structuring and storing NF states. Making such state objects complient with NFV framework which offloads state update operations to remote storage [72, 117] is not only hard but can also degrade the performance. A possible research direction is to split these complex state objects with multi-level pointers into simple objects such that each object can be addressed by a key.

A

Appendix

A.1 Proofs of soundness

We now prove the soundness of our StateAlyzr's algorithms.

Identifying Per-/Cross-Flow State

Slicing [63] and pointer analysis [24] have already been proven sound.

Theorem A.1. *If a middlebox uses standard packet receive functions, then our analysis identifies all packet processing procedures.*

Proof. For a procedure to perform packet processing: (i) there must be a packet to process, and (ii) the procedure must have access to the packet, or access to values derived from the packet. The former is true only after a packet receive function returns. The latter is true only if some variable in a procedure has a data dependency on the received packet. Therefore, a forward slice computed from a packet receive function over the variable containing (a pointer to) the packet will identify all packet processing procedures. □

Theorem A.2. *If a value is per-/cross-flow state, then our analysis outputs a top-level variable containing this value, or containing a reference from which the value can be reached (through arbitrarily many dereferences).*

Proof. Assume no top-level variable is identified for a particular per-/cross-flow value. By the definition, a per-/cross-flow must (i) have a lifetime

longer than the lifetime of any packet processing procedure, and (ii) be used within some packet processing procedure. For a value to be used within a packet processing procedure, it must be the value of, or be a value reachable from the value of, a variable that is in scope in that procedure. Only global variables and the procedure's local variables will be in scope.

Since we identify statements in packet processing procedures that use global variables, and points-to analysis is sound [24], our analysis must identify a global variable used to access/update the value; this contradicts our assumption.

This leaves the case where a local variable is used to access/update the value. When the procedure returns the variable's value will be destroyed. If the variable's value was the per-/cross-flow value, then the value will be destroyed and cannot have a lifetime beyond the packet processing procedure; this is a contradiction. If the variable's value was a reference through which the per-/cross-flow value could be reached, then this reference will be destroyed when the procedure returns. Assuming a value's lifetime ends when there are no longer any references to it, the only way for the per-/cross-flow value to have a lifetime beyond any packet processing procedure is for it be reached through another reference. The only such reference that can exist is through a top-level variable. Since points-to analysis is sound [24] this variable would have been identified, which contradicts our assumption. □

Identifying Updateable State

Theorem A.3. *If a top-level variable's value, or a value reachable through arbitrarily many dereferences starting from this value, may be updated during the lifetime of some packet processing procedure, then our analysis marks this top-level variable as updateable.*

Proof. According to the language semantics, scalar and compound values

can only be updated via assignment statements. According to Theorem A.1, we identify all packet processing procedures. Therefore, identifying all assignment statements in these procedures is sufficient to identify all possible value updates that may occur during the lifetime of some packet processing procedure.

The language semantics also state that the variable on the left-hand-side of an assignment is the variable whose value is updated. Thus, when a top-level variable appears on the left-hand-side of an assignment, we know its value, or a reachable value, is updated. Furthermore, flow-insensitive context-insensitive pointer alias is provably guaranteed to identify all possible points-to relationships [24]. Therefore, any assignment to a variable that may point to a value also pointed to (indirectly) by a top-level variable is identified, and the top-level variable marked updateable. □

Identifying Flowspaces

Theorem A.4. *If a middlebox uses standard patterns for fetching values from data structures, and the flowspace for a top-level variable's value (or a value reachable through arbitrarily many dereferences starting from this value) is not constrained by a particular header field, then our analysis does not include this header field in the flowspace fields for this top-level variable.*

Proof. A header field can only be part of a value's flowspace definition if there is a data or control dependency between that header field in the current packet and the fetching of an entry from a data structure. It follows from the proven soundness and precision of flow-sensitive context-insensitive pointer analysis [35] that the SDG will not include false data or control dependency edges. It also follows from the proven soundness of program slicing [63] that only data and control dependencies between source variables (i.e., the packet variable) and target variables (i.e., the

index variable, increment variable, or variable in a conditional inside a loop) will be included in the chop. □

Identifying Output-Impacting State

Theorem A.5. *If a top-level variable's value, or a value reachable through arbitrarily many dereferences starting from this value, may affect a call to a packet output function or the output produced by the function, then our analysis marks this top-level variable as impacting packet output.*

Proof. Follows from SDG construction soundness [49, 63]. If/when a packet output function is called is determined by a sequence of conditional statements. The path taken at each conditional depends on the values used in the condition. Control and data dependency edges in a system dependence graph capture these features. Since SDG construction is sound [49, 63], we will identify all such dependencies, and thus all values that may affect a call to a packet output function.

Only parameter values, or values reachable through arbitrarily many dereferences starting from these values, can affect the output produced by a packet output function. Thus, knowing what values a parameter value depends on is sufficient to know what values affect the output produced by an output function. Again, since SDG construction is sound, we will identify all such dependencies. □

A.2 Handling non-deterministic values

Non-deterministic values: Non-deterministic values, e.g., "gettimeofday" or "random" require special handling during fault tolerance and straggler mitigation to ensure COE. Specifically, we require NF instances to write every locally computed non-deterministic value into the datastore to pro-

vide determinism during fault tolerance. These values are used during the input replay phase to avoid divergence of internal state.

Non-deterministic values in straggler mitigation: When straggler mitigation is used, to ensure that the state of an NF instance and its clone do not diverge during straggler mitigation, CHC replaces local computation of non-deterministic values with *datastore based computations*. That is, NFs request the datastore, which computes, replies with, and stores each non-deterministic value. If a second request for a non-deterministic value comes with the same packet logical clock (from the straggler), the datastore emulates the computation and returns the same value again.

A.3 Proofs of Correctness and Chain Output Equivalence

The collective action taken by all the NF instances in the chain must be equivalent of the action taken by a chain of ideal NFs. The ideal NF chain consist of NFs with infinite resources where only one instance of each type is required to handle any network load. The ideal NF processes packets in the order of their arrival. Although the ideal NF chain has infinite resources, we still assume a standard network. The network can drop or reorder packets between two *end hosts*.

A.3.1 Consistency Guarantees of Cross-flow State Update

Theorem A.3.1.1. *Suppose we are given a cross-flow state* S *and instances* α *and* β *of the same NF, processing packets* P_α *and* P_β*, respectively.* α *and* β *cannot generate a state* S'' *which is unreachable if both* P_α *and* P_β *were to be processed by a single NF instance.*

Proof: The state update corresponding to the packets P_α and P_β can be applied by the store in any arbitrary order o. For S'' to be unreachable

by the ideal NF, o should not exist in the set of all possible orders $\{O\}$ in which packets can arrive at the ideal NF. As the network does not provide any ordering guarantees, $\{O\}$ contains all possible orders. Hence, o always exists in $\{O\}$. In other words, S is always reachable by an ideal NF under some particular order of updates from the input packets.

A.3.2 Consistency Guarantees of Cached Cross-flow State Update

Theorem A.3.2.1. *Supposed we are given a cross-flow state S copies of which are cached at NF instances α and β of the same NF, processing packets P_α and P_β, respectively. α and β cannot generate a state S'' which is unreachable if both packets are processed by a single NF instance with infinite resources.*

Proof: Cross-flow cached state is only used for read requests. All update operations are performed by the store. This ensures that update operations are applied on the most recent version of the state. According to Theorem A.3.1.1, state update always results in a consistent value, regardless of the order of the cross-flow state updates operations. Hence NF instances α and β cannot generate the state S''.

A.3.3 Safe Recovery of a Root Instance

Theorem A.3.3.1. *If a root with $P_i, ..., P_n$ logged packets ($n > i > 0$) crashes after successfully sending $P_i, ..., P_k$ packets ($i < k < n$) to downstream NF instances, then recovery of the root results in state S' such that S' is reachable by a chain of ideal NFs under some network drop scenario imposed on the input traffic.*

Proof: If the root crashes after transmitting packet P_k, all the remaining logged packets $P_{k+1}, ..., P_n$ are lost. The new root can only start from packet P_{n+1}. In such a case the resultant output/state is equivalent to the case of a chain of ideal NFs with $P_{k+1}, ..., P_n$ dropped by the network.

A.3.4 Safe Recovery of an NF Instance

Theorem A.3.4.1. *If an NF instance crashes after successfully processing packet* P_i*, forwarding it and pushing the corresponding state to the store, then the recovered NF instance reaches the same state as if no failure has occurred in the chain.*

Proof: The NF crashes after successfully forwarding the state corresponding to the packet P_i to the store; thus only in-transit packets are not processed (P_{i+1} and onward). Since all the unprocessed in-transit packets are logged at the root, they are replayed to the new NF instance to result in the same state as if no failure has occurred.

Theorem A.3.4.2. *If an NF instance crashes after successfully processing the packet* P_i *and forwarding it but before the successful pushing the corresponding state update, then the recovered NF instance reaches the same state as if no failure has occurred in the chain.*

Proof: The recovery process starts from packet P_i instead of packet P_{i+1} because the bit vector (§3.4.4) value corresponding to P_i at the root is non-zero which indicates that either the state has not been updated or the packet is lost. This results in replay starting from packet P_i. This replay regenerates the missing state update and results in the same state as if there is no failure. The duplicate suppression mechanism drops the packet P_i at the message queue of the immediate downstream NF instance, preventing any duplicate packets.

Theorem A.3.4.3. *If an NF instance crashes after successfully processing the packet* P_i *and transmitting its respective state but before the successful delivery of the packet to a downstream NF, then the recovered NF instance reaches the same state as if no failure has occurred in the chain.*

Proof: The recovery process starts from packet P_i instead of packet P_{i+1} because (as above) the bit vector value corresponding to P_i at the root

is non-zero which indicates that either the state has not been updated or the packet is lost. As the replay starts from P_i, the downstream NF does not experience any missing packets. The packet P_i may result in a duplicate state update which is suppressed by the store. According to the Theorem A.3.4.1, replay of logged packets (P_{i+1} and onward) recovers the state.

Theorem A.3.4.4. *If the last NF instance in the chain crashes before successfully transmitting the "delete" request for packet P_i to the root, then the recovered NF instance reaches the same state as if no failure has occurred in the chain and the end host (receiver) does not receive a duplicate packet.*

Proof: The packet P_i cannot leave the last NF without the successful transmission of "delete" request; thus, the packet P_i is not forwarded to the receiver. After recovery (according to Theorem A.3.4.2 and Theorem A.3.4.3), the NF instance reaches the same state as if no failure has occurred in the chain.

A.3.5 Safe Recovery of a Store Instance

Theorem A.3.5.1. *If an instance of the store crashes, then each recovered per-flow state S_P of each NF is equivalent to the state that would have resulted if there was no failure.*

Proof: As S_P is updated by a single NF at any given time, so the cached value of the state at the given NF is always the most recent state. Therefore, the new store instance reading all the cached S_Ps correctly recover the per-flow states.

Theorem A.3.5.2. *If an instance of the store crashes before any successful cross-flow state read operation, then the recovered cross-flow state S_C must be reachable by an ideal NF under some input traffic arrival/update order.*

Proof: Write-ahead log is replayed (§3.4.4) to reconstruct the lost cross flow state S_C. As none of the NFs have read the value, the log can be replay in any order (according to Theorem A.3.1.1, state will always be consistent.) to regenerate valid state.

Theorem A.3.5.3. *If an instance of the store crashes after a successful cross-flow state read operation, then the recovered cross-flow state S_C must be reachable by an ideal NF under some particular scenario.*

Proof: As the recovery process (§3.4.4) of cross-flow state starts from the last read value, replay of remaining log operations will always result in a consistent state (according to Theorem A.3.5.2.).

Bibliography

[1] Cisco Network Service Header: draft-quinn-sfc-nsh-03.txt. https://tools.ietf.org/html/draft-quinn-sfc-nsh-03.

[2] Cloud lab. http://cloudlab.us/.

[3] Codesurfer. http://grammatech.com/research/technologies/codesurfer.

[4] Docker swarm. https://github.com/docker/swarm. Accessed: 2017-09-25.

[5] HAProxy: The reliable, high performance TCP/HTTP load balancer. http://haproxy.1wt.eu/.

[6] Introducing mcrouter: A memcached protocol router for scaling memcached deployments. https://code.fb.com/web/introducing-mcrouter-a-memcached-protocol-router-for-scaling-memcached-deployments/. Accessed: 2018-06-21.

[7] Kubernetes. http://kubernetes.io.

[8] Linux advanced routing and traffic control howto. http://lartc.org/lartc.html. Accessed: 2017-09-25.

[9] The LLVM compiler infrastructure. http://llvm.org.

[10] LXC - Linux containers . https://linuxcontainers.org/lxc/introduction/.

[11] Network functions virtualisation – update white paper. https://portal.etsi.org/nfv/nfv_white_paper2.pdf.

[12] Network functions virtualisation – update white paper. https://portal.etsi.org/nfv/nfv_white_paper2.pdf.

[13] Networking napi. https://wiki.linuxfoundation.org/networking/napi. Accessed: 2017-09-25.

[14] NFV Management and Orchestration: An Overview. https://www.ietf.org/proceedings/88/slides/slides-88-opsawg-6.pdf.

[15] OpenVPN. http://openvpn.net.

[16] Passive Real-time Asset Detection System. http://prads.projects.linpro.no.

[17] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2017-09-21.

[18] Protobuf-c. https://github.com/protobuf-c/protobuf-c.

[19] Snort. http://snort.org.

[20] Squid. http://squid-cache.org.

[21] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

[22] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[23] Ashok Anand, Vyas Sekar, and Aditya Akella. Smartre: an architecture for coordinated network-wide redundancy elimination. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 87–98. ACM, 2009.

[24] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[25] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xOMB: Extensible open middleboxes with commodity servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 49–60, 2012.

[26] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 242–253. ACM, 2011.

[27] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, volume 99, pages 45–58, 1999.

[28] Davide B. Bartolini, Filippo Sironi, Donatella Sciuto, and Marco D. Santambrogio. Automated fine-grained cpu provisioning for virtual machines. *ACM Trans. Archit. Code Optim.*, 11(3):27:1–27:25, July 2014.

[29] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, CO, 2014. USENIX Association.

[30] M. Boucadair, C. Jacquenet, R. Parker, D. Lopez, P. Yegani, J. Guichard, and P. Quinn. Differentiated Network-Located Function Chaining Framework. Internet-Draft draft-boucadair-network-function-chaining-02, IETF Secretariat, July 2013.

[31] Anat Bremler-Barr, Yotam Harchol, and David Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 511–524. ACM, 2016.

[32] J. Brutlag. Speed Matters for Google Web Search. Technical report, 2009. https://services.google.com/fh/files/blogs/google_delayexp.pdf.

[33] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, September 2007.

[34] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[35] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, 1993.

[36] Jonathan Corbet. Jls2009: Generic receive offload. *Linux Weekly News (LWN)*, October 2009. https://lwn.net/Articles/358910/.

[37] Jonathan Corbet. Software interrupts and realtime. *Linux Weekly News (LWN)*, October 2012. https://lwn.net/Articles/520076/.

[38] Jonathan Corbet. Bulk network packet transmission. *Linux Weekly News (LWN)*, October 2014. https://lwn.net/Articles/615238/.

[39] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT*, 1977.

[40] Lorenzo De Carli, Robin Sommer, and Somesh Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1378–1390. ACM, 2014.

[41] Jeffrey Dean and Luiz Andre Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[42] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[43] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[44] Mihai Dobrescu and Katerina Argyarki. Software dataplane verification. In *NSDI*, 2014.

[45] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward predictable performance in software packet-processing platforms. In *NSDI*, 2012.

[46] Intel. Data Plane Development Kit. http://dpdk.org/.

[47] Peter Druschel and Gaurav Banga. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *OSDI*, volume 96, pages 261–275, 1996.

[48] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 543–546, 2014.

[49] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[50] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. Ginseng: Market-driven llc allocation. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 295–308, Denver, CO, 2016. USENIX Association.

[51] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceño, Russell Hunt, and Thomas Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.*, 20(1):49–83, February 2002.

[52] Aaron Gember, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. Technical report, Technical Report, 2013.

[53] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling innovation in network function control. In *SIGCOMM*, 2014.

[54] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 1–12, New York, NY, USA, 2012. ACM.

[55] Thomas Gleixner. [announce] 3.6.1-rt1. *Linux Weekly News (LWN)*, October 2012. https://lwn.net/Articles/518993/.

[56] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International Conference*, page 15. ACM, 2010.

[57] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 342–362. Springer, 2006.

[58] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 135–148, Hollywood, CA, 2012. USENIX.

[59] Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 177–190. ACM, 2013.

[60] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 465–478, New York, NY, USA, 2015. ACM.

[61] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *Proceedings of HotCloud*, June 2016.

[62] Tom Herbert and Willem de Bruijn. Scaling in the linux networking stack, 2011. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[63] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.

[64] Galen C. Hunt and Michael L. Scott. The coign automatic distributed partitioning system. In *OSDI*, 1999.

[65] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 25–36, New York, NY, USA, 2007. ACM.

[66] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, 2014. USENIX Association.

[67] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazieres, Balaji Prabhakar, Changhoon Kim, and Windows Azure. Eyeq: Practical network performance isolation for the multi-tenant cloud. In *HotCloud*, 2012.

[68] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[69] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoom, and Eric Keller. Stateless network functions. In *HotMiddlebox*, 2015.

[70] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 729–742, New York, NY, USA, 2014. ACM.

[71] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 67–81, New York, NY, USA, 2016. ACM.

[72] Junaid Khalid and Aditya Akella. Correctness and performance for stateful chained network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, 2019.

[73] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. Paving the way for nfv: simplifying middlebox modifications using statealyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 239–253, 2016.

[74] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based CPU in container environments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 313–328, Renton, WA, 2018. USENIX Association.

[75] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18:263–297, 2000.

[76] Alexey Kopytov. Sysbench manual. *MySQL AB*, 2012.

[77] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 71–84, New York, NY, USA, 2017. ACM.

[78] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J.Sel. A. Commun.*, 14(7):1280–1297, September 2006.

[79] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst.*, 34(2):6:1–6:33, May 2016.

[80] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *NSDI*, pages 589–603, 2015.

[81] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, 2014. USENIX Association.

[82] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[83] John C McCullough, John Dunagan, Alec Wolman, and Alex C Snoeren. Stout: An adaptive interface to scalable cloud storage. In *Proc. of the USENIX Annual Technical Conference–ATC*, pages 47–60, 2010.

[84] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[85] Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. Silverline: Data and network isolation for cloud services. In *HotCloud*, 2011.

[86] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2015.

[87] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *PLDI*, 1992.

[88] Vern Paxson. Bro: a system for detecting network intruders in real-time. In *USENIX Security (SSYM)*, 1998.

[89] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, November 2015.

[90] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[91] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. *ACM SIGCOMM Computer Communication Review*, 43(4):351–362, 2013.

[92] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen Hunt. Rethinking the library os from the top down. Association for Computing Machinery, Inc., March 2011.

[93] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 29–42. ACM, 2015.

[94] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. SIMPLE-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 27–38. ACM, 2013.

[95] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for End-host Rate Limiting. In *NSDI*, 2014.

[96] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico Replication: A high availability framework for middleboxes. In *SoCC*, 2013.

[97] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Escape capsule: Explicit state is robust and scalable. In *HotOS*, 2013.

[98] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System support for elastic execution in virtual middleboxes. In *NSDI*, 2013.

[99] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *ACM SIGSOFT*, 1995.

[100] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival O Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *WIOV*, 2011.

[101] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *POPL*, 1988.

[102] Stuart Schechter, Jaeyeon Jung, and Arthur Berger. Fast detection of scanning worm infections. In *Recent Advances in Intrusion Detection*, pages 59–81. Springer, 2004.

[103] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[104] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, 2012.

[105] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 1:1–1:13, New York, NY, USA, 2016. ACM.

[106] Justine Sherry, Peter Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Macciocco, Maziar Manesh, Joao Martins, Sylvia Ratnasamy, and Luigi Rizzoand Scott Shenker. Rollback recovery for middleboxes. In *SIGCOMM*, 2015.

[107] Alan Shieh, Srikanth Kandula, Albert G Greenberg, and Changhoon Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.

[108] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[109] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM.

[110] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Enhancing java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1):1:1–1:40, August 2009.

[111] Paul Turner, Bharata B Rao, and Nikhil Rao. Cpu bandwidth control for cfs. In *Proceedings of the Linux Symposium*, pages 245–254, 2010.

[112] Laura Vasilescu, Vladimir Olteanu, and Costin Raiciu. Sharing cpus via endpoint congestion control. In *Proceedings of the Workshop on Kernel-Bypass Networks*, KBNets '17, pages 31–36, New York, NY, USA, 2017. ACM.

[113] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[114] Mellanox. Messaging Accelerator (VMA). http://www.mellanox.com/page/software_vma.

[115] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association.

[116] Mark Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, July 1984.

[117] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI'18*, 2018.

[118] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618, New York, NY, USA, 2013. ACM.

[119] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T Kaushik, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 474–489. ACM, 2015.

[120] Wei Zhang, Sundaresan Rajasekaran, Shaohua Duan, Timothy Wood, and Mingfa Zhuy. Minimizing interference and maximizing progress for hadoop virtual machines. *SIGMETRICS Perform. Eval. Rev.*, 42(4):62–71, June 2015.

[121] Yuting Zhang and Richard West. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 191–201, Washington, DC, USA, 2006. IEEE Computer Society.

[122] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, page 5. ACM, 2014.

[123] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41, 2014.