2022

Network-Wide Monitoring And Debugging

Nofel Yaseen
University of Pennsylvania

Network-Wide Monitoring And Debugging

Abstract

Modern networks can encompass over 100,000 servers. Managing such an extensive network with a diverse set of network policies has become more complicated with the introduction of programmable hardwares and distributed network functions. Furthermore, service level agreements (SLAs) require operators to maintain high performance and availability with low latencies. Therefore, it is crucial for operators to resolve any issues in networks quickly. The problems can occur at any layer of stack: network (load imbalance), data-plane (incorrect packet processing), control-plane (bugs in configuration) and the coordination among them. Unfortunately, existing debugging tools are not sufficient to monitor, analyze, or debug modern networks; either they lack visibility in the network, require manual analysis, or cannot check for some properties. These limitations arise from the outdated view of the networks, i.e., that we can look at a single component in isolation. In this thesis, we describe a new approach that looks at measuring, understanding, and debugging the network across devices and time. We also target modern stateful packet processing devices: programmable data-planes and distributed network functions as these becoming increasingly common part of the network. Our key insight is to leverage both in-network packet processing (to collect precise measurements) and out-of-network processing (to coordinate measurements and scale analytics). The resulting systems we design based on this approach can support testing and monitoring at the data center scale, and can handle stateful data in the network. We automate the collection and analysis of measurement data to save operator time and take a step towards self driving networks.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Computer and Information Science

First Advisor

Vincent Liu

Keywords

Computer Networks, Network Debugging, Network Testing, Network Verification

Subject Categories

Computer Sciences

NETWORK-WIDE MONITORING AND DEBUGGING

Nofel Yaseen

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2022

Supervisor of Dissertation

Vincent Liu, Assistant Professor of Computer and Information Science

Graduate Group Chairperson

Rajeev Alur, Zisman Family Professor of Computer and Information Science

Dissertation Committee

Boon Thau Loo, RCA Professor of Computer and Information Science
Rajeev Alur, Zisman Family Professor of Computer and Information Science
Andreas Haeberlen, Professor of Computer and Information Science
Ryan Beckett, Researcher in the Mobility and Networking group of Microsoft

NETWORK-WIDE MONITORING AND DEBUGGING

To Maheen, my parents Zulekha and Yaseen, my siblings Moiz and Laiba, and my closest friend Osama, for all of their love, support and companionship.

ACKNOWLEDGEMENT

I am grateful to everyone that has supported me throughout my doctorate studies. I would first like to thank my advisor, Dr. Vincent Liu. Vincent inspired me to pursue research as a career and encouraged me to seek interesting problems I was passionate about solving. He introduced me to colleagues with shared interests, while always giving me the autonomy that I wanted to develop my intuition and research style.

The support of my committee has been invaluable. Dr. Boon Thau Loo, Dr. Vincent Liu, Dr. Andreas Haeberlen and Dr. Ryan Beckett all provided feedback that improved this dissertation. Working with them has taught me, by example, how to be a better researcher and mentor.

Two mentors helped immensely in my formative years as an undergraduate student: Dr. Ihsan Qazi and Dr. Fareed Zaffar. Dr. Qazi showed me how to be a productive researcher. He also instilled in me the desire to communicate ideas clearly and passionately. Dr. Zaffar helped broaden the scope of my work. They helped to establish the direction of my research.

My doctorate would have been much more difficult without support from the many colleagues and friends that I have met while working in the distributed systems laboratory and Microsoft Research. This includes, but is not limited to: John Sonchack, Caleb Stanford, Liangcheng Yu, Jiaqi Gao, Behnaz Arzani, Ryan Beckett, Selim Ciraci, Daniel S. Berger, Kevin Hsieh, Dan Crankshaw and Srikanth Kandula. Our conversations have led to many great insights (and papers)! It has been inspiring to watch your projects grow and take on lives of their own. Thank you!

This dissertation would not have been possible without my family. They showed me the value of persistence and hard work. Lastly and most importantly, I would like to thank Osama Shiraz Shah and Maheen Iqbal for all their love and support. Osama

has inspired me to stay curious and engage with hobbies outside of research since we were friends in college. Maheen has made life enjoyable and kept me grounded in reality. She is my life partner and helped me through long days research and writing.

ABSTRACT

NETWORK-WIDE MONITORING AND DEBUGGING

Nofel Yaseen

Vincent Liu

Modern networks can encompass over 100,000 servers. Managing such an extensive network with a diverse set of network policies has become more complicated with the introduction of programmable hardwares and distributed network functions. Furthermore, service level agreements (SLAs) require operators to maintain high performance and availability with low latencies. Therefore, it is crucial for operators to resolve any issues in networks quickly. The problems can occur at any layer of stack: network (load imbalance), data-plane (incorrect packet processing), control-plane (bugs in configuration) and the coordination among them. Unfortunately, existing debugging tools are not sufficient to monitor, analyze, or debug modern networks; either they lack visibility in the network, require manual analysis, or cannot check for some properties. These limitations arise from the outdated view of the networks, i.e., that we can look at a single component in isolation.

In this thesis, we describe a new approach that looks at measuring, understanding, and debugging the network across devices and time. We also target modern stateful packet processing devices: programmable data-planes and distributed network functions as these becoming increasingly common part of the network. Our key insight is to leverage both in-network packet processing (to collect precise measurements) and out-of-network processing (to coordinate measurements and scale analytics). The resulting systems we design based on this approach can support testing and monitoring at the data center scale, and can handle stateful data in the network. We automate the collection and analysis of measurement data to save operator time and take a step

towards self driving networks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

Data centers like those of Google, Microsoft, Amazon, etc. underpin much of today's computing. Indeed, it is rare to find any modern application or service — from search engines to driverless cars — that does not require some type of network connection. The applications generate billions of requests every day to these data centers, where they are processed by thousands of servers and network devices.

As the size of networks grows, programmable networks and distributed network functions have emerged as key technologies to make networks more flexible. In programmable networks, operators write data plane code using domain-specific code like P4 [39] and the compiler generates an efficient implementation for the target device. Whereas distributed network functions are implemented entirely in software, distributed across a pool of servers, and replicate state for fault tolerance.

These technologies are opening doors to better support and scale data center applications, but they are also making the networks more complex. In such a large complex network environment, problems are inevitable. Ideally, operators would have self-driving networks that would be fully automated while using minimal resources (CPU, bandwidth, memory, etc.). The network itself could monitor, detect, and debug all problems immediately. These networks could also predict application resource usage and network performance trends to avoid problems that could occur in the future, schedule configuration, and request optimal capacity upgrades. However, such self-driving networks are too far in the future.

Currently, operators need to think of many possibilities to troubleshoot network prob-

lems. For example, there could be many reasons for a packet drop: load imbalance, network congestion, misconfiguration (eg. incorrect forwarding rule) leading to black holes, malfunction of devices (eg interface flapping), or a combination of them. These problems lead to high job completion times, chronic stragglers, etc. Making problems worse, the data centers continue to grow in the number of processing units (a large number of servers), high bandwidth usage (terabytes per data center), diverse network policies (QoS, routing), and flexibility (programmable data planes, network functions).

Operators collect measurements from various parts of the network but focus on a single device or path in isolation as data lacks precision and/or coordination. Even within network devices, operators tend to focus on either the data-plane or the control-plane. New network devices have distributed and/or stateful properties that are not visible in such isolation. Therefore, we need to approach monitoring so we can measure and understand across devices and time; and we can check for properties that span different layers. Hence, this thesis argues that, ***to better diagnose issues in modern networks, operators should leverage both (1) in-network packet processing to collect precise measurements and (2) out-of-network processing to coordinate measurements and scale analytics.***

In this chapter, we shall go through the network infrastructure, enumerate recent trends, limitations and our contributions to network monitoring and debugging.

1.1. Network Infrastructure

Networks play a huge role in enabling smooth running of various applications. These applications use networks to share resources, distribute computation, make backups, etc. They have various requirements for example, there are bandwidth intensive (e.g. Big Data), delay sensitive (e.g. Web Search) or even both (e.g. Video Streaming).

To efficiently scale the applications, operators need to design their networks carefully. Network designing requires many decisions such as topology, switches, control plane functionality, end-hosts role, etc.

Switches: For cost and compatibility reasons, large networks mostly use commodity Ethernet switches that have limited resources [18]. These switches have a large port capacity (32/64/128) and can process up to 40G/100G per port. For such fast packet processing, switches use TCAM (Ternary Content Addressable Memory) to store forwarding rules. When a packet arrives, a switch can lookup the forwarding rules in parallel using TCAM in a single cycle to decide the action. Due to the memory constraints and time constraints, commodity switches only allow few operations per packet, e.g. forwarding rule, ACL, QoS, etc.

Control Plane: In commodity switches, the control plane resides alongside the data plane. The forwarding rules are a function of control plane configuration, and path information collected from protocols like OSPF that computes least-cost paths and BGP that exchange routing and reachability information.

With the advent of OpenFlow, SDNs have gathered a lot of attention that decouples the control plane from the forwarding engine. The control plane is moved to a logically centralized controller. The controller is then responsible to install forwarding rules in the data plane. However, a significant portion of switches still relies on the former configuration-based approach.

End-hosts: End-hosts are the source and sink for most of the data traffic flowing through the network. Traditionally, their influence on the network has been limited by just having using congestion control protocol to manage the amount of data being sent into the network. However, within data centers, operators use end-hosts for monitoring traffic and managing forwarding rules as well.

1.2. Trends in Modern Networks

Programmable Switches: One of the most recent innovations in network hardware is programmable switches. Programmable switches use a language like P4 that has packet processing abstractions, e.g. headers, parsers, control, tables, actions, and deparsers. In the latest version, P4 16, there are 6 programmable blocks: ingress parser, ingress control, ingress deparser, egress parser, egress control, and egress deparser. The control has match-action tables, counters, registers, and stateful ALUs. There are also two platform-dependent blocks: the packet replication engine (PRE) that is placed between the ingress/egress pipeline and the buffer queuing engine (BFE) that is placed at the end of the egress pipeline. These are proprietary hardware implementations and vary by each vendor.

In P4 switches, packet first arrives at the ingress pipeline and the ingress parser transforms the bits into headers according to the programmed parser. After parsing, packet is processed using ingress control before going to ingress deparser. Then packets go to PRE, where vendor specific functions are present like clone, resubmit, etc. After PRE, packet is processed by egress pipeline (similar to ingress) and finally packet is transformed back into bit representation before sending to BFE that sends out the packet to the next hop.

Network Functions: Networks Functions (NFs) are software implementations of middleboxes (e.g., NATs, firewalls, and load balancers) that play a critical role in modern networks. Operators deploy these on several clusters to handle the load at the data center scale. Generally, each network administration deploys its own custom implementations of NFs, but there is also some work in designing a reusable network stack for NFs [92]. NFs can play a variety of roles in the network: scan a connection for malicious behavior, serve content from a cache inside the network to reduce bandwidth costs, or compress data to provide better performance on low-resource mobile devices.

1.3. Problems and Contributions

This section provides discussion on the two problems addressed in this thesis. It covers recent works in network monitoring and debugging, their limitations, outlines the thesis contributions.

Problem 1: *Existing methods of monitoring at device or path level metrics are insufficient to measure and analyze network-wide behaviour.*

There have been many recent solutions to collect and process data from various parts of the network for diagnosis, but they don't provide a holistic picture of the network Path-level and device-level metrics form the foundation of today's measurement tools. Traditional tools like switch counter polling and packet sampling target individual entities in the network. Comparison of measurements of different entities is difficult beyond just averages and long-term behavior. Slightly better are path level metrics like those gathered at the end host [196], through Explicit Congestion Notification (ECN) [20], or In-band Network Telemetry (INT) [104]. These path-level metrics provide similar data as counters and packet sampling, but on the level of entire paths; measurements from different paths are, however, still only comparable at a coarse granularity. Thus, when faced with questions about network-wide behavior, operators are forced to approximate the answer using tangential, but more easily collectible measurements.

Subsequently, identifying prevalent network-level patterns typically requires a significant amount of manual effort and specialized analyses. Existing tools limit to capture flow- and switch-level trends (e.g., heavy hitter analysis [194], network tomography [77] or the vast array of network analytics suites on the market [1, 2, 3, 4, 5, 6, 79]). For instance, to determine the presence of synchronized requests/responses, an operator might need to instrument the start and stop times of all flows in the system,

correct for the time drift of different machines, compute the cluster tendencies of the data (e.g., with a Hopkins statistic or heuristic), and distinguish it from all-to-all traffic by examining the sources and destinations of synchronized flows. To determine whether this pattern is a particularly common one would require additional analyses.

Contributions. This thesis calls for a different approach in collecting and processing measurement data. As the importance and size of networks grow, making the best out of the underlying hardware has far reaching economic and environmental consequence. Programmable switches allow the distribution of responsibilities of monitoring to data plane and control plane. The data plane can perform extremely fine-grained in-band processing of network traffic, but is fundamentally limited in the type of computation and resources available. Augmenting the data plane is a control plane with the opposite tradeoffs.

Towards this directions, this thesis presents Speedlight and `tpprof`, both of them together collect fine-grained network-wide data and analyze the data in real time to find the most common traffic patterns. Speedlight is a fine-grained, accurate, and precise measurement primitive that operates on the scale of an entire network. It captures a set of local measurements, called network snapshot, that together provide a coherent image of the entire network data plane at nearly a single point in time. Through coordination, network snapshots are able to guarantee both causal consistency (i.e., that the measured values are coherent) and approximate synchronicity (i.e., that the measurements were taken near-contemporaneously).

`tpprof` builds on the measurements from Speedlight to identify the most common traffic patterns in the network. We introduce two abstractions, network states and traffic pattern subsequences, that together enable network operators to easily describe and reason about common traffic patterns. `tpprof` then uses domain-specific algorithms to rank both networks states and sequences. Finally, there is a simple grammar

for describing traffic patterns and introduce an algorithm that automatically identifies approximate occurrences of known traffic patterns within network traces.

Problem 2: *Existing tools are insufficient to test or verify the full-stack of deployed latest stateful packet processors adequately.*

As networks grow, operators are increasingly deploying new technologies like network functions and programmable switches. They offer flexibility and computation in the network, but also make the network more complex. They are an emerging bottleneck to correctness and availability.

For network functions, existing work suggest using static verification to prove correctness of distributed systems [81, 103, 125, 144, 178, 185, 188, 192]. While powerful, the need to reason about every possible interleaving of inputs and control flows presents a significant obstacle to the application of these techniques in today's network functions. Attempting to explore the full space of control flow paths often leads to state/path explosion [103, 125, 178]. Runtime verification is alternate approach that only test inputs and control flows that are seen in practice, thus improving scalability and enabling verification of actual deployments running over actual data. However, today's runtime verifiers cannot be applied as-is to deployed network functions. The challenge (for network functions) is the need, at runtime, to: (1) reason about the coordination between events issued at different locations, (2) efficiently aggregate global state after each event, and (3) scale sub-linearly with the size of the original system.

To reduce bugs in programmable switches, recent work has suggested static verification to prove the correctness of P4 programs [59, 119, 65]. For pure, stateless data plane programs, static verification is often effective since there are no complex pointer-based data structures or loops, making analysis both more accurate and tractable. However, real forwarding behavior depends on many other components: the control plane, match-action rules, the compiler translation, the switch state including regis-

ters, and vendor specific hardware and compiler. There is parallel work that fuzz test programmable switches that allows a developer to check the entire device. Prior fuzz testing tools for P4 [140, 164, 163] generate input in software, and therefore severely limit the number of test cases.

Contributions. This thesis takes steps to improve the runtime verification for network functions and fuzz testing of programmable switches. We build on the key insight to collect measurement data at line rate and use off-path CPUs for flexible analysis and computation. We build *Aragog* and *FP4* to target network functions and programmable switches respectively.

Aragog is a scale-out, runtime verification tool for network functions that provides a simple, but expressive language for describing violations of invariants, with a focus on supporting network functions. Examples of network-centric language features that are found in *Aragog* 's Invariant Violation (IV) specifications, but that are uncommon in other runtime verifiers are support for properties that are parametric over the "location" of events, properties that reference stateful variables, the ability to execute partial matches over packet fields, and support for temporal predicates. *Aragog* translates these IV specifications to a set of symbolic automata that can efficiently verify the current global state of the system. In addition, to ensure that the system can scale out to a near-unlimited number of machines.

FP4, a greybox fuzz testing framework for P4-programmable network devices that is both (a) full-stack and (b) line-rate. *FP4* feeds semi-random packets to real programmable switches to attempt to attempt to trigger violations of programmer-specified assertions. The switches are purposely kept as faithful as possible to their production deployments and run instrumented versions of their original P4 programs and control planes

1.4. Organization

In the remainder of the thesis, we will dive into each of the four systems that I built to address the above problems. At the end there is related work and conclusion.

CHAPTER 2

Synchronized Network Snapshots

2.1. Introduction

As networks continue to grow in size and bandwidth, a detailed understanding of their overall behavior is increasingly difficult to come by. Consider the question: does my network's load balancing protocol balance the network's load? A definitive answer to this question (and others like it) is out of the scope of traditional measurement tools.

In order to answer it, we would need visibility into the fine-grained behavior of the entire network. Instead, the target of traditional tools like switch counter polling and packet sampling are individual entities in the network. Comparison of measurements of different entities is difficult beyond just averages and long-term behavior. Slightly better are path-level metrics like those gathered at the end host [196], through Explicit Congestion Notification (ECN) [20], or In-band Network Telemetry (INT) [104]. These path-level metrics provide similar data as counters and packet sampling, but on the level of entire paths; measurements from different paths are, however, still only comparable at a coarse granularity.

Thus, when faced with questions about network-wide behavior, operators are forced to approximate the answer using tangential, but more easily collectible measurements. In the case of load balancing, they might redefine the definition of balance to a purely local metric (e.g., monitoring packet drops or buffer utilization for 'high' values) or look only at average load. Similar workarounds exist for most questions an operator might ask [196, 134, 16], but these approximations can be misleading, especially in networks with bursty load and/or high capacity [193]. The design of network switches,

architectures, and protocols depend on understanding network behavior both in detail and at a network-wide scale.

This dissertation presents the design of a fine-grained, accurate, and precise measurement primitive that operates on the scale of an entire network. The goal of our primitive is the capture of a *Synchronized Network Snapshot*: a set of local measurements that together provide a coherent image of the entire network data plane at nearly a single point in time. Enabling our work is a recent trend toward highly programmable switch data and control planes. We leverage these tools to implement a system, Speedlight, for taking synchronized network snapshots on Wedge100BF-series switches. The implementation uses P4 and the code is open source.[1]

Compared to more traditional measurement primitives, synchronized network snapshots are a fundamentally distributed operation—one that involves tight coordination of the control and data planes of multiple network devices. Through coordination, network snapshots are able to guarantee both causal consistency (i.e., that the measured values are coherent) and approximate synchronicity (i.e., that the measurements were taken near-contemporaneously). The primitive itself is agnostic to the type of local measurement and supports the collection of any variable accessible from the data plane: counters, packet samples, switch state, queue depth, etc. It is also amenable to partial deployment.

At its core, our system is inspired by distributed snapshot protocols [45, 107]. In a classical distributed snapshot, a snapshot initiator sends out a message that propagates among a set of distributed nodes to cause them to (without stopping the system or synchronizing clocks) take snapshots of their local state. The guarantee provided by these protocols is that the snapshot creates a causally consistent partition of the system's events. For any event e that is 'pre-snapshot', any event that can be con-

[1] `https://github.com/eniac/Speedlight`

strued as *causing* e is also pre-snapshot. In the context of networks, this might mean that if a snapshot of queue depth captures a packet p in a queue q, that p will not be counted as part of any other queue, and furthermore that the effects of every send and receive that led to p being in that particular queue at that particular time are also included in the snapshot. For that reason, distributed snapshots are an attractive abstraction; however their application to high-speed networks carries a few challenges.

First, while traditional snapshots provide a set of measurements that *could* have happened simultaneously, one of their primary criticisms is that they do not provide any guarantee of how close in time the measurements occurred. Second, snapshot protocols often make strong assumptions about the system, e.g., that nodes are single-threaded and capable of arbitrary computation, and that they are connected via reliable FIFO channels. Real switches, on the other hand, are highly parallel, extremely limited in their data plane processing capabilities, and exhibit non-FIFO behavior (e.g., prioritization, packet re-circulation, etc.). It can be difficult to adapt certain functionality to programmable data planes [168, 161], and distributed snapshots are no exception.

The key insight of Speedlight is that modern switches are two-level devices. The data plane can perform extremely fine-grained in-band processing of network traffic, but is fundamentally limited in the type of computation and resources available. Augmenting the data plane is a control plane with the opposite tradeoffs.

Speedlight therefore splits the responsibility of taking snapshots such that the data and control planes each mitigate the weaknesses of the other. At a high level, we first break the data plane of each switch into small, simple components that obey single-threaded and FIFO assumptions. The snapshot implementation at each of these data plane components is not fully featured, but provides two key properties: (1) it allows for multiple simultaneous snapshot initiators in the style of [170], and (2) it guarantees

consistency and correctness in all cases, regardless of data plane limitations. The control plane CPU is then responsible for the global, PTP-coordinated initiation of a snapshot at all data plane components, as well as the stitching together of results.

The end result of our system is that all of the individual measurements in a synchronized network snapshot are not only consistent, they are guaranteed to occur almost contemporaneously. Our current implementation guarantees a drift of at most 10s of microseconds (less than a single RTT in most cases); drift can be decreased further using more advanced time synchronization techniques [110]. In addition to presenting a detailed design and implementation, we demonstrate the primitive on real workloads. To summarize, our work makes the following contributions:

- We present a Synchronized Network Snapshot algorithm for the collection of distributed state within the data plane of a network. Our design provides strong guarantees regarding both the semantics of the measured values and their timeliness.
- We then present the design and implementation of Speedlight, a practical realization of the Synchronized Network Snapshot algorithm. Our prototype, built for Wedge100BF-series switches, is able to achieve microsecond-level synchronization of global network snapshots.
- Finally, we use our system to measure real workloads running on our testbed. This measurement study demonstrates both feasibility and usefulness of our approach.

2.2. Background and Motivation

Network measurement is a method through which we seek to understand network behavior. This can be in the context of designing new protocols/architectures, evaluating existing ones, or diagnosing issues in live networks. Over the years, a wide

range of network measurement tools and analyses have been created to assist in the aforementioned tasks.

Measuring path-level properties. One common approach is the use of end-to-end or flow/path-level measurement tools. Extremely flexible, these tools enable observers to evaluate, from the network edge, the aggregate effect of the network in the context of application-level measures like latency, throughput, and drop rate. An advantage to this approach is that it accurately reflects the overall experience of application traffic. They also often do not require additional network support, although there are recent exceptions [104, 20]. Though effective for some use cases, edge vantage points typically lack visibility into fine-grained network behavior and details of the network's structure [134].

Measuring devices. A much more fine-grained approach is to measure individual network components directly. This typically takes the form of counters or packet sampling/mirroring, but recent proposals have explored the use of more complex metrics like flow-based queries, heavy-hitter analysis, and TCP-level statistics [134, 123, 175, 184]. Direct measurement is precise, and with sufficient device support, quite expressive.

2.2.1. Whole-network Measurement

Path-level and device-level metrics form the foundation of today's measurement tools. Unfortunately, by themselves both approaches typically provide little to no guarantees about the relationship between measurements, or the effect of clock drift and other asynchronous behavior.

For bursty and/or high-capacity networks, even small amounts of unattended asynchronicity can lead to large inaccuracies in measurement. As an illustrative example, consider a datacenter network. A good NTP accuracy within a LAN is 1 ms; in con-

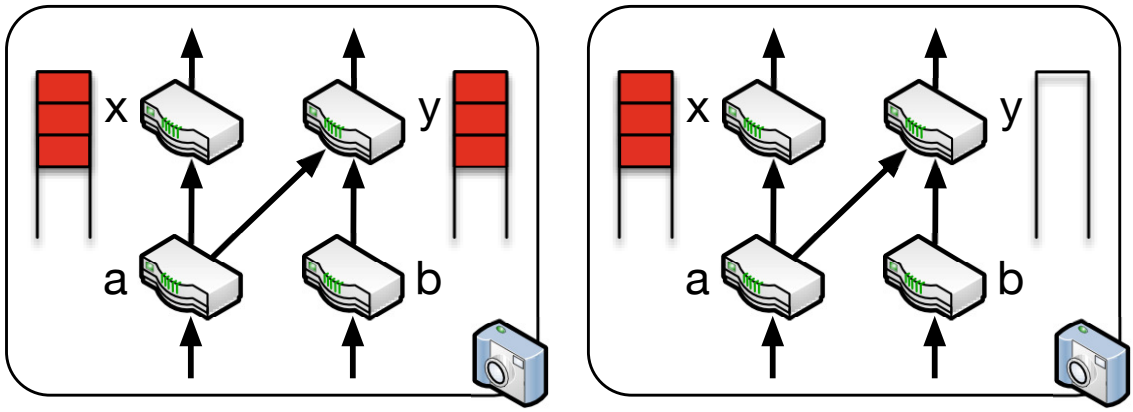
trast, typical datacenter RTTs are an order of magnitude lower, and there is evidence that traffic bursts can be even shorter ($O(10\,\mu s)$) [193]. In effect, for any two measurements of network behavior at different locations, their relationship is both tenuous and difficult to bound. This inaccuracy will only grow as network speeds increase.

Even within a single router, synchronized information is not always available. Counters may be on different line cards and most counter polling mechanisms are not optimized for polling more than one counter at a time. Without driver-level modifications, polling a single counter on a modern switch typically takes on the order of 1 ms [193].

For the above reasons, measurements are not often compared directly. Instead, when trying to examine network-wide behavior, most frameworks aggregate individual measurements, typically using statistical analysis over relatively long time periods so as to skirt the issue of unsynchronized clocks. Averaging and summation are particularly common mechanisms. An observer can compare average utilization of multiple components to determine how they differ over a given time span. They can also use a total path-level drop count in combination with network tomography to pinpoint lossy components. Network operators have become creative in their techniques to obliquely measure the whole network; however, as we will see in the next section, there are still fundamental limitations to existing tools.

2.2.2. A Case for Consistency

To illustrate the importance of consistent whole-network measurement, imagine we have the simple network depicted in Figure 2.1. The network consists of two ingress routers (a and b) connected to two egress routers (x and y) in an asymmetric fashion. Even for this simple case, many critical questions about network behavior are difficult to answer.



(a) 'Balanced' Queues

(b) 'Unbalanced' Queues

Figure 2.1: Asynchronous measurements can be misleading. These diagrams show two possible measurements of queue depth for $\boldsymbol{x}$ and $\boldsymbol{y}$. In both cases, the network could be perfectly balanced or arbitrarily unbalanced—the measurements fail to distinguish between the two cases.

1. Is the network load balanced? We begin with the question asked in Section 2.1. Imagine that an operator deploys a new load balancing protocol to a and b. How does she evaluate its efficacy? How would she know if there was a performance bug in the protocol? How does she quantify the room for improvement?

One possible solution is to sample the queue depth at x and y; however, on their own, these samples do not answer the above questions. Particularly in the presence of bursty traffic, asynchronous measurements can provide misleading results. For instance, the balanced queue measurements shown in Figure 2.1a could be a result of (a) a perfectly balanced network in which queue depths never differ, (b) an entirely unbalanced network in which one queue is always empty, or (c) anything in between. All of the above is still true if we observed unbalanced queues as in Figure 2.1b.

Common workarounds include averaging many samples (an approach that captures biases and long-term effects, but is not general) or only analyzing relative performance compared to a previous solution (an approach that is not always possible, and whose utility is limited). Instead, a set of contemporaneous measurements would give a

more meaningful view into the behavior of the network.

2. Where should we add capacity to the network? A related question is where an operator should add capacity to the network, i.e., the process of network provisioning. Today, they might examine tail utilization or drops over every link to identify bottlenecks in the network. Asynchronous measurements are sufficient for this, but fail to answer many followup questions. For instance, would adding a parallel path alleviate congestion or is a per-link capacity upgrade necessary? Balanced load among existing paths would indicate the former, while localized hotspots would indicate the latter. They provide similarly limited insight into whether alleviating one bottleneck would lead to others. Again, contemporaneous measurements would provide more insight into network behavior.

3. Is traffic synchronized? Synchronized measurements can also assist in application-level debugging, especially in the case of TCP incast and related performance problems. Many of the same issues from the previous questions also apply here. Today, detection of synchronized behavior is typically done either empirically (e.g., testing if added jitter in TCP sends alleviates the problem), or obliquely (e.g., testing for characteristics of incast like high flow count, TCP timeouts, and drops [134, 196]). These workarounds are both inaccurate and only possible after performance has already been impacted. We argue that a whole-network measurement primitive is a more natural and effective alternative.

4. What is the global forwarding state? Finally, a classic problem in networking is the detection of bad forwarding state. Forwarding loops are the canonical example of an undesirable network state that is difficult to detect, especially if the loops are transient and/or flapping. This class of problems have taken a newfound importance in the context of RDMA and RoCE. RoCE's PFC mechanisms can cause network deadlocks, not only when there are routing loops, but in many other cases as well [88].

For a general method of verifying and diagnosing these issues, a consistent snapshot is crucial—otherwise we can observe states that are impossible.

2.3. Overview

We seek to design a measurement primitive that captures a set of measurements representing a meaningful view of the whole network as it appears at a single point in time. We note that in pursuit of this goal, a truly simultaneous network-wide snapshot is impossible without either freezing the network or using prohibitively expensive hardware like atomic clocks. Instead, our goal is a snapshot primitive with the following two properties:

- *Causal consistency:* If a measurement in snapshot S includes the effect of event e (e.g., a packet reception), S also includes the effects of every event that led to e.

- *Near synchronicity:* The time difference between every pair of measurements in the snapshot is guaranteed to be at most d, where $d < RTT$. Our prototype guarantees $d < 100\,\mu s$, even for large networks.

Rather than capturing the true instantaneous behavior, i.e., what one would have seen if we froze time to examine the network, causal consistency provides a record of what *could* have happened. Augmented with a tight bound on the maximum jitter of the snapshot, we argue that the combination of these two requirements preserves most useful metrics.

Architecture. Our design for synchronized network snapshots involves three types of entities: ***data-plane processing units*** of which each switch/router can have many, ***control planes*** running at each device, and ***snapshot observers*** running on hosts connected to the network. Our design allows for partial deployment (Section 2.9) as well as a wide range of networking technologies and configurations. It is also

agnostic to the measured data—*any value accessible at line rate in the data plane can be snapshotted*. It is achieves all of this with minimal additional state and overhead.

Protocol. At the core of our design is a modified version of the Chandy-Lamport snapshot algorithm. Originally created in the context of distributed systems, snapshots seek to capture the global state of a system without a common clock or shared memory, and without affecting the operation of the system itself. What Chandy and Lamport proposed was a protocol in which an initiator can trigger a cascade of messages that, with causal consistency, partitions the system's events into 'pre-snapshot' and 'post-snapshot', then collects the state of every node and channel at the boundary.

There are two key differences between our version and the original. First, while most snapshot implementations begin with a single initiator, snapshots in our system are initiated at *all nodes simultaneously*. Second, our design is necessarily bipartite. Modern control planes, typically running on a general purpose CPU, can easily implement a fully featured snapshot protocol, but in terms of consistency, they are no better than a remote host. Recent proposals for programmable data planes, on the other hand, more closely adhere to the assumptions of the Chandy-Lamport protocol, but today's ASICs have limited functionality/resources. By leveraging both, we seek to mask each plane's deficiencies with the other's strengths—neither is sufficient on its own.

Operation. A Synchronized Network Snapshot begins humbly: with a host acting as a snapshot observer. The observer broadcasts a request to every device in the network to take a snapshot of a given metric at a given time in the future. The control planes running on every device then coordinate among themselves using a protocol like PTP to achieve the synchronized, network-wide initiation of a data plane snapshot. The data plane, where processing elements most closely adhere to the requirements of

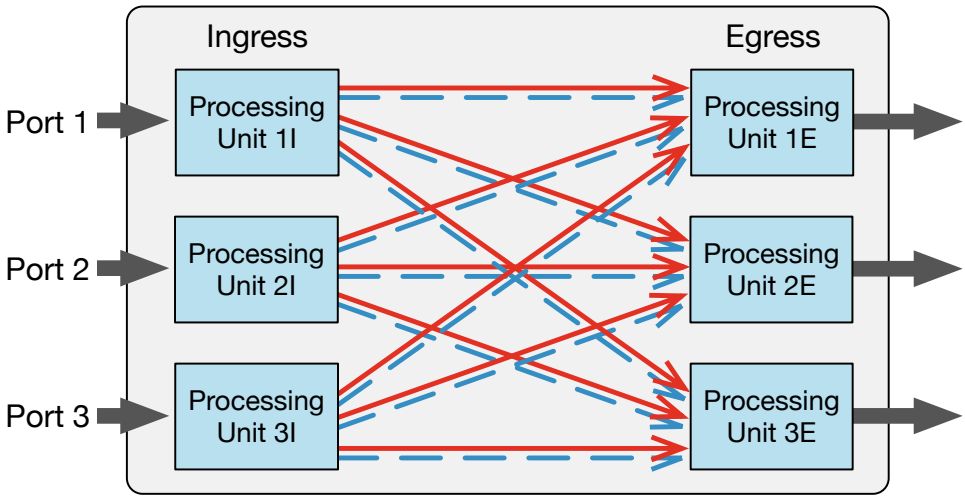


Figure 2.2: A conceptual model of a router in a network snapshot. At the lowest layer are the ingress/egress processing units of individual ports. Connecting the ingress and egress ports are unidirectional FIFO channels. Multiple channels may exist in the case of Class-of-Service queues (two in the diagram, represented by solid red and dotted blue arrows).

Chandy-Lamport, implements the core of a multi-initiator snapshot protocol, while the control plane fills in missing pieces of the protocol as necessary.

2.4. Network Snapshot Algorithm

Before we describe the detailed design of our system for Synchronized Network Snapshots, we first introduce the design of an idealized data plane snapshot algorithm. In Sections 2.5 and 2.6, we describe how we adapt this algorithm to current hardware.

2.4.1. System Model

Abstractly, a network is a collection of switches and routers. Each switch or router can be subdivided at many different levels. At the highest level, modern switches will often contain one or more line cards, each line card being responsible for one or more ports. The ports can be subdivided further into ingress and egress processing units (see Figure 2.2), although in some designs, multiple logical processing units can be implemented with a single physical unit. Regardless, the *per-port, per-direction processing unit* forms the fundamental building block of packet processing: despite aggressive amounts of parallelism, for a single port and single direction, processing is

guaranteed to be linearizable.

Connecting the ports are unidirectional communication channels. Within the device, the ingress processing unit of each port is logically connected to the egress processing unit of every other port.[2] These connections can potentially contain multiple sub-channels when the switch is configured to prioritize certain traffic. In those cases, individual classes of service (CoS) obey FIFO ordering, but the packets of different service classes can be interleaved. Between devices are physical links that connect the egress processing unit of one port to an ingress unit on a different network device.

In modern Ethernet, there is only one device sitting on either end of the channel. In other types of networks or in partial deployment scenarios, a logical channel exists between every connected egress and ingress.

2.4.2. Protocol

The original Chandy-Lamport algorithm relied on a few key assumptions: linearizable nodes, simplex FIFO channels, no message drops, and bounded delay. When considering a network of routers, few if any of these assumptions hold. Instead, our network snapshot protocol operates over the network of per-port, per-direction processing units connected by logical communication channels (either a physical link or an internal, logical CoS queue). This formulation gives us a distributed system of linearizable nodes connected by FIFO channels. To handle drops and delays, we take inspiration from subsequent work (e.g., Li et. al. [114]) and classical network assumptions. While snapshot protocols exist for other, more relaxed system models, they typically require massive storage requirements, delaying of messages, or they limit the gathered state to packet/byte counts.

Figure 2.3 depicts our algorithm in pseudocode. Every processing unit keeps track of

[2]Some devices allow for more complex internal packet communication, e.g., recirculation. If configured, those channels can be handled by adding additional logical channels to our model.

```
• state: Local state to be snapshotted.
• snaps[]: Set of snapshotted state.
• sid: Current snapshot ID. Starts at 0.
− lastSeen[]: The last IDs seen from each upstream neighbors.
Function onReceiveCS(pkt):
    if pkt.sid > sid then
        /* New snapshot */
        for i ← sid + 1 to pkt.sid do
            snaps[i] ← state
        sid ← pkt.sid
    else if pkt.sid < sid then
        /* In-flight packet */
        for i ← pkt.sid + 1 to sid do
            Update channel state of snaps[i] with pkt
    lastSeen[pkt.sender] ← pkt.sid
    All snapshots up to min(lastSeen[*]) are complete
    Update state and set pkt.sid ← sid

Function onReceiveNoCS(pkt):
    if pkt.sid > sid then
        for i ← sid + 1 to pkt.sid do
            snaps[i] ← state
        sid ← pkt.sid
    All snapshots up to sid are complete
    Update state and set pkt.sid ← sid
```

Figure 2.3: Per-processing-unit pseudocode for our idealized network snapshot protocol (w/ and w/o channel state). The match-action approximation and other details are described in Section 2.5 and 2.6. Global state preceded by '−' is only necessary for channel state.

its current snapshot ID, s, initialized to 0. They also keep track of the local state that is the target of the snapshot. Note that this requires snapshots of shared state (e.g., a switch-wide packet counter) be taken as a set of local snapshots or re-implemented as local state.

Every packet carries a snapshot ID field, s_p that indicates the epoch from which it was sent (similar to [114]). 'Piggybacking' of markers on every packet ensures that snapshot ID updates are resilient to packet loss. On receipt of a packet, processing units compare the packet's carried snapshot ID with their local ID. If $s_p > s$, the upstream neighbor has begun a new snapshot, and the current node should as well. The local state is immediately saved and the local ID is updated ($s \leftarrow s_p$). If, on the

other hand, $s_p < s$, the packet was in-flight when the snapshot occurred, and should therefore be included in the channel's state.

The specifics of how channel state should be recorded is metric-dependent. For instance, a network-wide packet count might require processing units to record the number of packets in their queue, then add in-flight packets to the count as they arrive. In other cases, the operator may not care about channel state at all (e.g., instantaneous queue depth measurements), and can omit this step. Either way, the processing unit sets $s_p \leftarrow s$ before forwarding. When packets arrive with $s_p = s$, no actions are necessary. The above process ensures causal consistency of recorded states.

Initiating a snapshot. Snapshots can be concurrently initiated at any number of processing units by incrementing their local snapshot ID. The affected processing units will tag all subsequent packets with the incremented ID. Assuming that the network is strongly connected and there is regular traffic flowing along every channel, even a single initiator will eventually cause all processing units to take the snapshot. When those assumptions break down, re-initiations may be necessary to ensure liveness. We discuss the details and practical challenges of snapshot initiation in Section 2.6.

Completing a snapshot. If channel state is not important to the measurement, a processing unit is finished with its snapshot as soon as it records its state and updates its local snapshot ID. Otherwise, it is finished when it sees that all of its upstream neighbors have updated their ID. At that point, there is no possibility of receiving additional in-flight packets ($s_p < s$). To detect this, each processing unit stores an array of the last seen ID from every upstream neighbor. Lines 11 and 12 in Figure 2.3 implement this process. With or without channel state, a network-wide snapshot s' is complete when all nodes in the system are finished with snapshot $s > s'$. As with snapshot initiation, we discuss the practical concerns of snapshot completion in real

networks in Section 2.6.

Proof sketch. The proof of correctness for our algorithm mirrors that of prior work, but we provide a brief sketch of the proof here. For each state-affecting event e on node n, $e \in PRE$ ('pre-snapshot') if it occurs before the local snapshot on n. The algorithm is correct if, for all $e \in PRE$, if e' happens causally before e, then $e' \in PRE$.

1. If e and e' are on the same processing unit, the above is trivially true.
2. Otherwise, $e \in PRE \Rightarrow e' \in PRE$ by contradiction.
   (a) Assume for snapshot i that $e' \notin PRE$ is a send of packet p and $e \in PRE$ is the matching receive.
   (b) Since $e' \notin PRE$, p must be carrying snapshot ID i.
   (c) That is not possible since $e \in PRE$, thus there is a contradiction.
   (d) Similar logic can be applied to other relationships between e and e'.

2.5. Data Plane Coordination

This section is the first of two that describes in detail the design of Speedlight. Speedlight leverages the match-action stages and stateful memory found in emerging programmable ASICs such as the Barefoot Tofino [33]. Using these tools, each processing unit can execute limited computation over packet headers/metadata using state in the form of register arrays.

Though the ASICs are powerful, their limitations and other network-specific concerns make the translation from the preceding snapshot algorithm to Speedlight difficult. This section describes the design of Speedlight's data plane while Section 2.6 describes the control plane that complements it.

2.5.1. Packet Headers

As mentioned in Section 2.4.2, network snapshots require additional header information. Speedlight does not require host cooperation, so headers are added by the first snapshot-enabled router, and removed before delivery to hosts. The required fields are as follows. If channel state is not desired, items preceeded by a — may be omitted.

- **Packet Type** can take one of two values: initiation or data. Most traffic is classified as data; initiation packets are special control messages that we describe in Section 2.6.

- **Snapshot ID** is set at each hop to be the processing unit's current snapshot ID. Conceptually, it specifies the snapshot to which the send of the packet is a member, and informs the current processing node whether the packet is part of a new one, or in-flight from an old one.

— **Channel ID** uniquely identifies each upstream neighbor. If there are multiple channels between neighbors, there should be an ID for each. Our reference implementation assumes switched Ethernet and no packet re-submission, so for ingress processing units, there is only one upstream neighbor (the external neighbor), and for egress units, the number of upstream neighbors is bounded by the number of ingress ports on the local router.

2.5.2. Stateful Variables

Some amount of inter-packet persistent state is also required in each processing unit. These mirror the state in Figure 2.3.

- **Counters** store target local state of the snapshot. These are managed separately from the snapshot protocol. This variable corresponds to *state* in Figure 2.3.

- **Snapshot ID** is an integer representing the node's current snapshot ID. This

value corresponds to *sid*.

- **Snapshot Value[max snapshot id]** stores the snapshoted state and, if necessary, channel state. These must be encoded into a value that fits into available register space. Equivalent to *snaps*.

– **Last Seen[# of neighbors]** tracks the last snapshot ID from each upstream neighbor. See definition of Channel ID for a discussion of what constitutes an upstream neighbor in our system. Corresponds to *lastSeen*.

2.5.3. Packet Processing Procedure

Figures 2.4 and 2.5 show the operation of ingress and egress processing units in Speedlight. Both approximate the algorithm presented in Section 2.4 with a few notable differences.

In both types of processing units, the first step is to read the target state and update it. The update process is orthogonal to the snapshot logic, only intersecting if the target state requires it (e.g., to ignore snapshot traffic). The next step is to examine the snapshot header.

The core of the snapshot processing procedure is similar to the one described in Section 2.4.2. The processing unit updates the neighbor's last seen value and then tests to see if the packet's snapshot ID is less than, greater than, or equal to the processing unit's local ID. As mentioned in Section 2.4.2, in-flight packet handling is metric-specific and configured by the network operator, and much of the algorithm can be elided if channel state is not necessary.

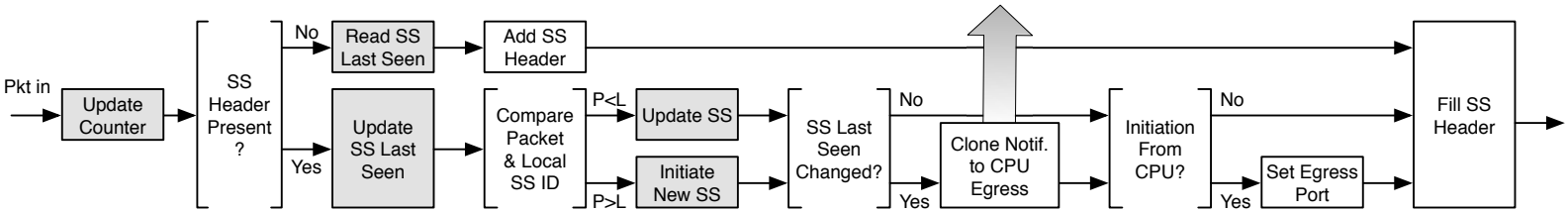


Figure 2.4: Pipeline of an ingress snapshot processing unit. Shaded boxes involve stateful registers.

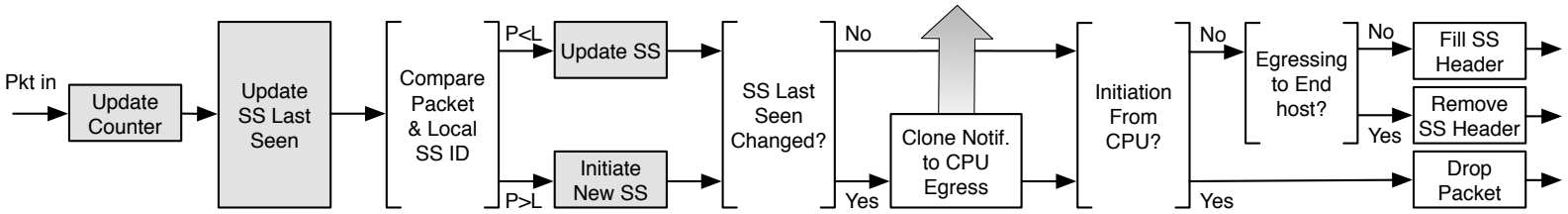


Figure 2.5: An egress processing unit. Shaded boxes involve stateful registers. Not shown is the CPU egress path.

Differences from the idealized algorithm. The primary differences between Speedlight's data plane and the algorithm in Section 2.4.2 derive from hardware limitations in high-speed programmable switches. One key limitation is that today's switches do not have the ability to loop through (at line rate) intermediate snapshot IDs when the packet's ID and the local ID differ by more than 1. Re-circulation loops are not possible as they would violate FIFO ordering. Instead, our implementation produces a complete and consistent snapshot iff the ID of all upstream neighbors and the local processing unit differ by at most 1. The following section describes how we detect and mitigate inconsistency.

Another is that the space of possible snapshot IDs and storage of the snapshot state are tightly constrained. As such, Speedlight enables rollover of the snapshot ID to 0 after reaching the maximum ID. For this, we assume that no ID in the system is ever 'lapped', i.e., that the maximum difference between any two snapshot IDs in the system is $(max\ snapshot\ id - 1)$. This can be enforced by the snapshot observers out-of-band. This assumption allows us to rely on the contents of the Last Seen array as a reference to detect if the packet's ID and/or the local ID have rolled over.

Snapshot Notifications. We mask the above deficiencies using the control plane. Supporting that process is a notification channel between the two planes. After any update of either the local *Snapshot ID* or of any *Last Seen* array entry, the data plane exports a notification to the CPU to assist in determining snapshot progress/completeness. For an upstream neighbor n, this notification includes the former value of LastSeen[n] along with the former and new Snapshot ID. Depending on the case, the former and new values may not be distinct. It will become clear in the following section why we need all four values.

2.6. Control Plane Coordination

Speedlight's data plane is augmented with a control plane to form a two-tier, mutualistic system in which each is responsible for masking the limitations of the other. This section examines some of the key scenarios in which the control plane is necessary.

Synchronized snapshot initiation. One of the primary responsibilities of Speedlight's control plane is to initiate snapshots in a timely fashion. At a high level, it does this by (a) synchronizing clocks between the control planes of different network devices, and then (b) executing a global, coordinated network snapshot initiation. Clock synchronization is a well-studied field, and Speedlight leverages this existing work. In our implementation, we use PTP, although the choice of protocol is orthogonal to our design.

Coordinated snapshot initiation, (b), is executed using the synchronized time. A snapshot observer first schedules a snapshot i for a given time in the future by registering the event with all device control planes. When the time comes, the control planes broadcast a message to all local ingress processing units. The message includes a snapshot header with *snapshot ID* set to i, the newly initiated snapshot. The ingress unit will process this snapshot header much like a regular packet—initiating a new snapshot if i is larger than the current snapshot ID. The control plane in this case is treated as an additional neighbor for the last seen array, though this value is only used for rollover detection and not to detect snapshot completion. After processing is complete, the ingress processing unit forwards the initiation to the egress unit of the same port, which drops the packet after processing. Unlike regular snapshot header processing, the packet is not included in the update counter stage and is never considered an in-flight packet.

Including control plane initiation, there are three ways by which a processing unit

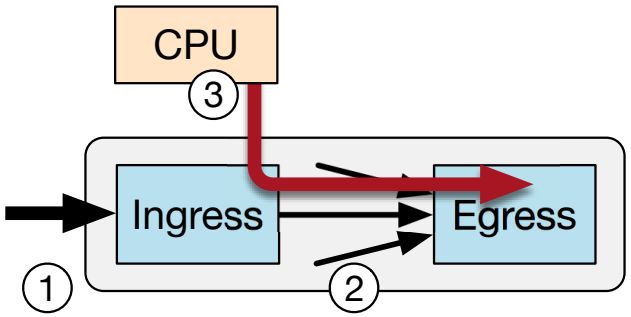


Figure 2.6: The three ways in which a processing unit can be induced to take a new snapshot. The initiation can come from: (1) a neighboring device, (2) another processing unit within the same device, or (3) from a control plane initiation message that, for every port, travels CPU→ingress→egress.

can be induced to take a new snapshot. The three methods, illustrated in Figure 2.6, cover normal snapshot-enabled packets from external (1) and internal (2) neighbors that have already begun the snapshot, as well as the control plane initiation messages (3). With these three initiation methods, Speedlight ensures a level of start-time synchronization beyond what a similar counter polling framework could achieve (see Section 2.8). That is in addition to the consistency provided by the snapshot protocol itself.

Detecting snapshot completion and inconsistency. In a classical distributed snapshot, a node's local state is valid as soon as it takes a local snapshot, and the state of the channel is valid when it receives an up-to-date snapshot marker on that channel. The global snapshot is complete when all such state is valid. In Speedlight, the control plane is responsible for gathering state and detecting the completion of snapshots. It is also responsible for detecting when snapshotted values become inconsistent. This scenario only occurs when channel state is required, and is not present in the original Chandy-Lamport algorithm. Rather, it is the direct result of the hardware limitations described in Section 2.5.

Figure 2.7 shows how a Speedlight control plane processes snapshot notifications to

```
• lastRead[unit]: Latest finalized snapshot for each unit.
− ctrlSnapID[unit]: Controller's view of units' current IDs.
− ctrlLastSeen[unit][neighbor]: Controller's view of the last seen array for each processing
  unit.
1  Function OnNotifyCS(unit, currentID, neighbor, currentLS):
2      if currentID ≠ ctrlSnapID[unit] then
3          /* New snapshot ID */
4          done ← min(ctrlLastSeen[unit][*])
5          for i ← done + 1 to currentID do
6              Mark i as inconsistent
7          ctrlSnapID[unit] ← currentID
8      if currentLS ≠ ctrlLastSeen[unit][neighbor] then
9          /* New last seen ID */
10         ctrlLastSeen[unit][neighbor] ← currentLS
11     toRead ← min(ctrlLastSeen[unit][*])
12     for i ← lastRead[unit] + 1 to toRead do
13         if i is not inconsistent then
14             Read snapshot value for i from unit
15     lastRead[unit] ← toRead

16 Function OnNotifyNoCS(unit, currentID):
17     if currentID ≠ lastRead[unit] then
18         validValue ← Read value for currentID from unit
19         for i ← currentID to lastRead[unit] + 1 do
20             value ← Read snapshot value i from unit
21             If value is uninitialized use validValue, otherwise validValue ← value
22         lastRead[unit] ← currentID
```

Figure 2.7: Control plane detection of complete and inconsistent snapshots with and without channel state. Note that min() must be rollback aware, but *lastRead* can be used as a reference. Global state preceeded by '−' are only necessary for channel state.

detect completion/inconsistency both with and without channel state.

1. *w/ Channel State:* Recall that in the common case, a processing unit is finished with snapshot i when $\forall u : lastSeen[u] \geq i$. Hardware limitations introduce an extra requirement: that the snapshot ID advances by exactly 1 each time. For example, if unit's snapshot ID is 5 and it receives a message from the snapshot 2 epoch, ideally the data plane would increment associated channel state for snapshots 3–5. Unfortunately, current ASICs cannot execute (at line rate) the required instructions to keep those intermediate snapshot values consistent. Speedlight marks them as inconsistent and handles notification drops conservatively.

2. *w/o Channel State:* The simpler case, a processing unit is done with a snapshot as soon as it increments its ID, records its local state, and sends a notification to the CPU. The snapshot ID can still skip forward; however, in this case, the CPU can infer the proper snapshot value. See lines 19–21 in Figure 2.7. Note that we must check for value initialization to account for notification drops.

All values are shipped to the snapshot observer, which assembles snapshots from all the devices with which it registered the snapshot. The observer computes completion and executes retries. If a device fails, it may timeout and be excluded from the global snapshot.

Ensuring liveness. An extension of the above two responsibilities, the control plane is also responsible for ensuring that snapshots are eventually initiated and completed at every processing unit. There are two reasons why this may not happen without assistance.

The first is packet drops of either the initiation message or update notifications. Especially for ingress processing units whose upstream neighbor is not snapshot enabled (e.g., a unit connected to an end host), a dropped initiation means that the processing unit will never advance its snapshot ID. Dropped notifications can also be problematic as they may cause snapshots to be incorrectly marked as inconsistent. To address both issues, Speedlight control planes will resend initiations for incomplete snapshots after a timeout. This is safe as duplicate and outdated control plane initiations are ignored by the data plane, and duplicate notifications are dropped at the control plane. Speedlight's control plane can also proactively poll the data plane registers to help recover from simple cases of notification drops.

The second is a lack of traffic when channel state is required. As completion of the snapshot is gated on receiving an up-to-date snapshot marker from all upstream

neighbors, if there is no such traffic on which to piggyback, the snapshot may never complete. This can happen due to traffic patterns, or it can be a natural consequence of the routing configuration (e.g., when using spanning trees or up-down data center routing). Speedlight has separate mechanisms for each situation. For a traffic-related absence of packets, we can inject broadcasts into the network that force propagation of snapshot IDs. For a lack of traffic due to network structure, operators can configure the removal of non-utilized upstream neighbors from *ctrlLastSeen* consideration.

Node attachment. Finally, we discuss the process of adding new devices to the network. For every snapshot, the snapshot observer keeps a list of all currently active devices. When adding a new device, it must be registered with the snapshot observer before it is included in the next snapshot. New devices will not start with the current snapshot ID. Instead the control plane initializes all state (registers in the data plane and tracking state at the control plane) to 0. As soon as traffic arrives from neighboring devices, the snapshot will jump ahead to the current value, if it is not 0. If it does jump ahead, the snapshot observer will ignore any spurious snapshot completions as the device would not have been in its expected device set when initiating the snapshot.

2.7. Implementation

We implemented a prototype of Speedlight with all of the data plane and control plane functionality described in Sections 2.5 and 2.6 for Wedge 100BF-series switches [97]. Wedge 100BF switches are driven by the Barefoot Tofino, a commodity multi-Terabit data plane ASIC that integrates recent designs for programmable line rate packet parsing [69], match-action forwarding [40], and stateful processing [168].

2.7.1. Data Plane

The Speedlight data plane is a pipeline of P4 match-action tables that compiles to the Tofino. We created multiple versions for different metrics, with and without

Variant	Packet Count	+ Wrap Around	+ Chnl. State
Computational Resources			
Stateless ALUs	17	19	24
Stateful ALUs	9	9	11
Control Flow Resources			
Logical Table IDs	27	35	37
Conditional Table Gateways	15	19	19
Physical Stages	10	10	12
Memory Resources			
SRAM	606 KB	671 KB	770 KB
TCAM	42 KB	59 KB	244 KB

Table 2.1: Resource usage for the Speedlight data plane on the Tofino. Numbers are for a snapshot of per-port packet counters and 64 ports.

wraparound and channel state support. Each implementation contains around 1000 lines of P4-14 code. Figures 2.4 and 2.5 show the logical ingress and egress match-action pipelines, assuming a snapshot that requires channel state.

Table 2.1 summarizes the key resources required by our prototype, broken down by the resources' logical functionality. We make no guarantee of the optimality of our prototype; the statistics represent a rough upper bound on the resource utilization of Speedlight. Even so, the prototype occupies less than 25% of any given type of dedicated resource—the remainder can be used for other data plane functionality.

As Table 2.1 shows, the prototype utilizes 10 to 12 physical processing stages in the Tofino to satisfy sequential dependencies in its control flow. It does *not* prohibit those stages from also implementing other ingress or egress data plane functions. Anything independent of the snapshot logic, such as forwarding or access control, can be compiled into the same stages and operate in parallel.

Speedlight fits well with other switch responsibilities. Its data plane is most expensive in terms of stateful ALUs (sALU), used to implement operations on register arrays, e.g., updating or initializing a snapshot. This is opposite of typical data plane

functionality, which tends to apply mostly stateless operations to packet headers.

Resource requirements for Speedlight increase with the use of wraparound and channel state, features that require more complex logic. Memory requirements also grow with the number of ports in the snapshot, as the data plane must allocate larger register arrays and tables to store and address the per-port statistics. The configuration shown in Table 2.1 is for 64 port snapshots, the maximum number of ports that a single processing engine in the Wedge100BF's Tofino can support. A configuration with wraparound and channel state for 14 port snapshots, as used for evaluation in Section 2.8, requires 638 KB of SRAM and 90 KB of TCAM.

2.7.2. Control Plane

We wrote the snapshot control plane in Python (∼2000 lines of code) and ran it on the switch CPU, which has a PCIe-3.0 X4 link to the Tofino ASIC. The control plane uses a compiler generated Thrift API to initialize tables, set up mirroring, and poll register arrays. Time synchronization was done via `ptp4l` and `phc2sys`.

The snapshot control plane receives notifications from the Tofino using a raw socket implemented by a kernel-level DMA packet driver. It listens for notifications, which trigger its main event handler as depicted in Figure 2.7. There are alternatives to this approach, e.g., a P4 digest stream, but we found that raw sockets made the implementation straightforward and offered significantly better performance.

2.8. Evaluation

We evaluated Speedlight in a hardware testbed and used it to perform measurement campaigns that study widely used distributed applications and protocols. Our testbed consists of a Barefoot Wedge100BF-32X programmable switch with 128 25 GbE ports connected to six servers with Intel(R) Xeon(R) Silver 4110 CPUs via 25 GbE links. We emulated a small leaf-spine topology in our testbed, as depicted in Figure 2.8.

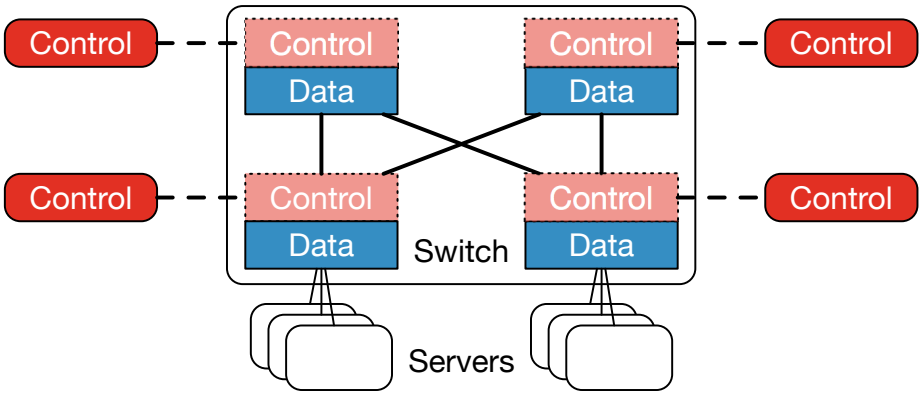


Figure 2.8: Depiction of our testbed topology.

We did this by splitting the 128 port switch into 4 fully isolated logical switches with lower fan-outs.

As in a real deployment, the virtual switches were connected with 100 GbE passive copper links. At the data plane, all forwarding tables were replicated for each virtual switch. At the control plane, we ran duplicate versions of the protocol. To emulate clock drift between switch control planes, snapshots were initiated based on the local system clock of four distinct PTP-synchronized servers. With the inclusion of network latency, our synchronization numbers therefore represent an upper bound.

To load balance traffic along the multiple paths in our testbed, we implemented two different algorithms alongside the snapshot logic in the switch data plane ASIC: ECMP [87] and flowlet switching [99].

Workload. We used three distributed applications in our testbed. The first is Hadoop running a Terasort [26] benchmark workload with 5B rows of data. Our Hadoop instance ran version 2.9.0 with YARN [27] on 10 mappers and 8 reducers. The second is Spark's GraphX [29] running a PageRank [28] synthetic benchmark workload with 100,000 vertices. Our Spark instance ran version of 2.2.1 with Yarn on 5 servers. Finally, we implemented memcache [24], running an `mc-crusher` 50-key

multi-get workload [57]. We populated the Hadoop and memcache instances with data during a setup phase that was not measured.

Counters. We implemented a variety of performance counters including per-port packet and byte counters along with queue depth measurements. However, in this section we primarily focus on an exponentially-weighted moving average (EWMA) of packet interarrival time. The EWMA counter was implemented in two phases due to hardware limitations on register computation:

```
interarrival = pkt_timestamp - last_ts[port]
last_ts[port] = pkt_timestamp
if packet_count[port] is even:
    temp_ewma[port] += interarrival
else:
    temp_ewma[port] /= 2
    ewma[port] /= temp_ewma[port]
```

Underlined variables are implemented with stateful registers. The EWMA updates on every other packet with the average interarrival of the last two packets. As shown in the code, our implementation is functionally equivalent to an EWMA with a decay factor of .5.

2.8.1. Synchronization of Network Snapshots

We begin by evaluating the synchronization properties of Speedlight. For this, we configured processing units to tag snapshot notifications with the current timestamp. Recall that notifications are sent on any update of either the local snapshot ID or the last seen array, i.e., on any progress in the algorithm. In the experiment, we sent a command to each of the four virtual control planes in our testbed to schedule a snapshot. At the scheduled time, they sent initiations to every processing unit (ingress and egress) under their control as described in Section 2.6. *Synchronization*

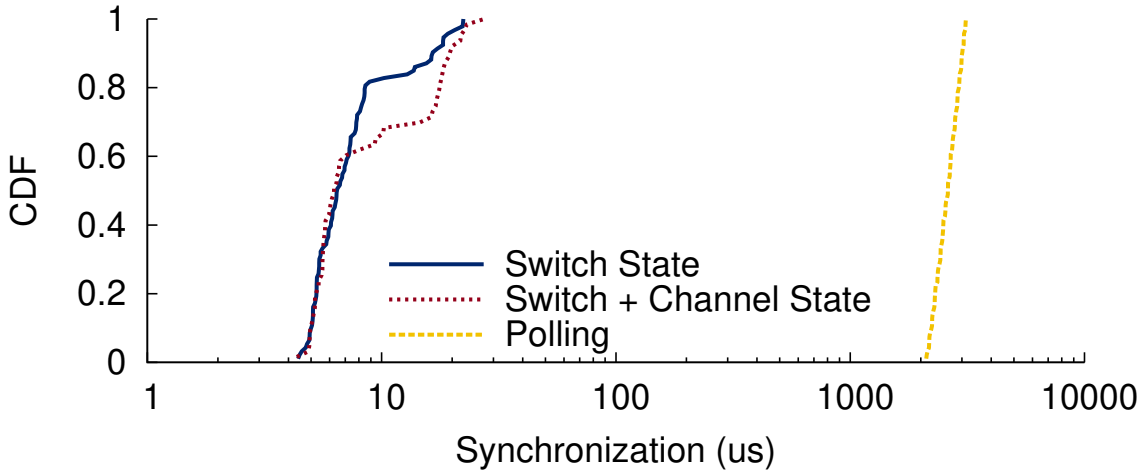


Figure 2.9: Synchronization of network-wide measurements using snapshots and traditional polling.

of a snapshot ID is defined as the difference between the earliest and latest timestamps on any notification with that ID.

Figure 2.9 shows a CDF of synchronization for three different approaches: (1) traditional counter polling, (2) Speedlight w/o channel state, and (3) Speedlight w/ channel state. In both configurations of Speedlight, median synchronization was $\sim$6.4 µs. The maximum synchronization delta we observed was 22 µs w/o channel state, and 27 µs w/ channel state, likely due to randomness in PTP, queuing, and scheduling. These values are well-within a single RTT for most networks. As one might expect, channel state synchronization has a longer tail as completion depends on all upstream neighbors advancing to the current snapshot.

For comparison, we also measured the synchronization of a typical counter polling framework where an observer polls the statistic for each port individually via a control plane agent that reads and returns the value on-demand. For a full sequence of network-wide measurements, the median difference between the first and last poll was 2.6 ms.

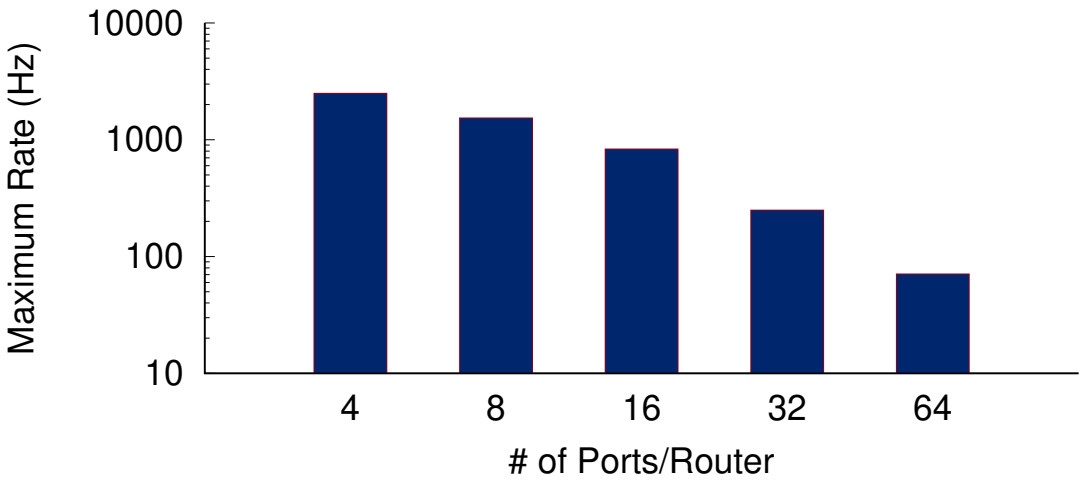


Figure 2.10: Max. sustained snapshot rate before notification queue buildup. Results are shown for a range of router port counts and assume no channel state.

2.8.2. Scalability of Speedlight

We also evaluate how Speedlight scales with the size and complexity of the network. In particular, we ask two questions: (1) how does the scale of the network affect the *frequency* with which Speedlight can take snapshots, and (2) how does the scale affect the *time synchronization* of those snapshots. Storage scalability was briefly addressed in Section 2.7.1.

Speedlight's architecture lends itself well to scalability; control planes are responsible for their own switch, and each processing unit has at most one external neighbor regardless of how many routers are added to the network. Instead, the primary factor in performance is number of ports per router.

Figure 2.10 shows the maximum sustained snapshot frequency versus router port count. In the experiment, we initiated a series of snapshots on a single switch with fixed interval. Snapshot frequencies that were too high eventually resulted in notification drops. The graphs plot the highest frequency without drops. Even for 64 ports (a full linecard), Speedlight can sustain over 70 snapshots per second. Note

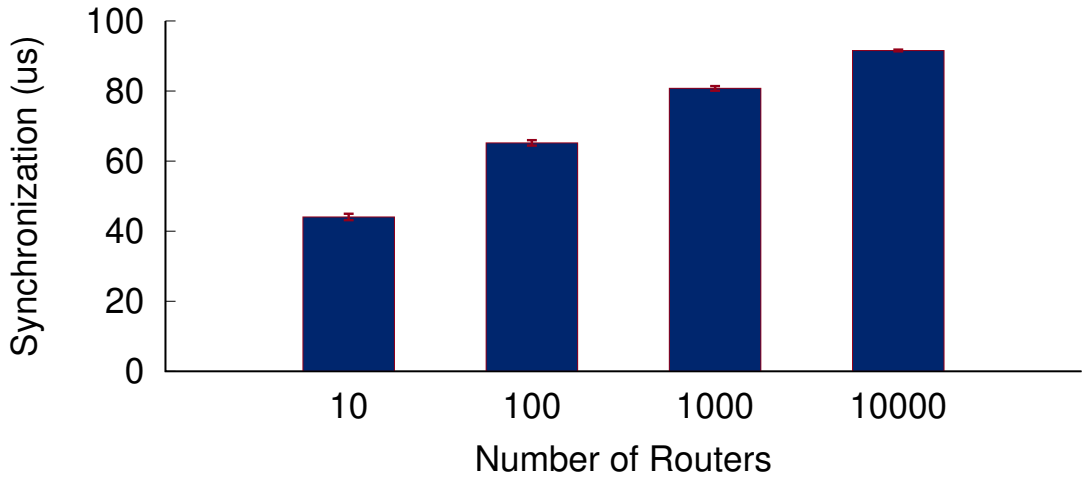


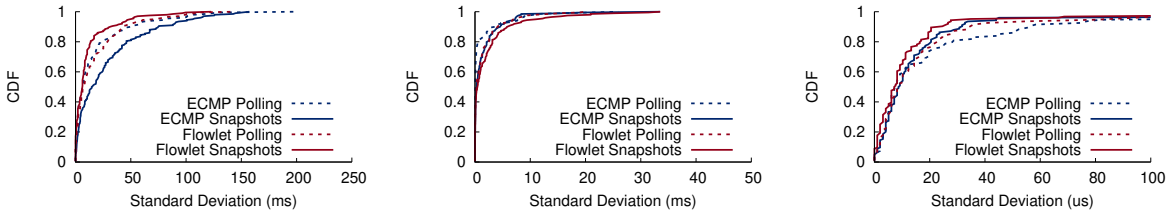
Figure 2.11: Average synchronization of Speedlight snapshots in larger network deployments. The snapshot assumes 64-port routers and no channel state.

that the ASIC-CPU channel is more than sufficient; rather, the bottleneck is in our unoptimized control plane processing latency. Thus, Speedlight supports bursts of higher frequency snapshots given a sufficiently large socket receive buffer.

Network size primarily affects Speedlight's synchronization. Figure 2.11 shows average whole-network synchronization for several large simulated networks. Our simulation included PTP time drift, OpenNetworkLinux scheduling effects, and the latency between initiation and data plane snapshot execution. Distributions for all of these values were collected from our hardware testbed. While Speedlight's multi-initiator design limits time drift, additional routers and ports can make encountering tail effects more likely; however, this effect is asymptotic and still stays under typical RTTs.

2.8.3. Use Case: Evaluating Load Balancing

We began this chapter with a running example of a question an operator might want to ask about a network: how well is my load balancing protocol working? We demonstrate Speedlight's ability to answer this question by comparing the performance characteristics of ECMP and Flowlet load balancing algorithms in the presence of Hadoop, GraphX, and memcache. In theory, Flowlet forwarding should balance load



(a) Hadoop (b) GraphX (c) Memcache

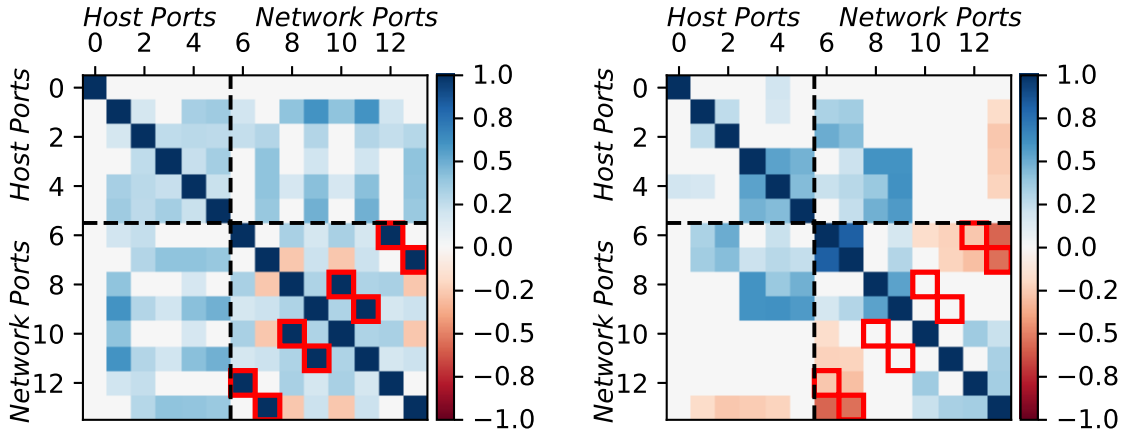
Figure 2.12: Standard deviation of uplink load balancing in our leaf-spine topology. We compared two approaches: flow-based ECMP and flowlet load balancing. We tested Hadoop, GraphX, and memcache as well as polling versus snapshots. Note the difference in units on the x-axis.

more fairly because it splits traffic at a finer granularity [99]. In practice, our understanding of the impact of flowlets on load balance is limited to average utilization, drop rate, flow completion time, and other carefully crafted proxies for the property in which we are actually interested.

In this experiment, we took a series of snapshots, and computed the standard deviation of the EWMA of packet interarrival times across uplink ports. To account for workload deviations, uplinks were compared only to other uplinks on the same switch. Figure 2.12 shows CDFs of the standard deviations for our Hadoop, GraphX, and memcache workloads taken with both snapshots and traditional polling. The three workloads showcase three different behaviors.

For Hadoop, polling shows little-to-no gain for flowets, when in reality flowlets improve balance significantly. For GraphX, polling consistently underestimates the imbalance in the network. Our Memcache workload is very evenly distributed, but exhibits the opposite behavior—polling consistently overestimates the imbalance.

Together, these experiments illustrate an important point. For measures of whole-network behavior, the issue is not just that polling might provide an incorrect view of the network, but that it is difficult to place a bound on the inaccuracy.



(a) Snapshot

(b) Polling

Figure 2.13: Pairwise correlation coefficients for egress ports while running GraphX. The red boxes highlight port pairs on the same ECMP paths, which are expected to have high positive correlations.

2.8.4. Use Case: Synchronized Traffic

The second use case we target is the detection of synchronized application traffic patterns for understanding behavior or debugging performance issues. For this experiment, we measured EWMA of packet rates at egress of all ports, in 100 snapshots taken 1 second apart. We then calculated pairwise correlation between ports using Spearman [51] tests.

Figure 2.13 shows the statistically significant ($\rho < 0.1$) correlation coefficients found while running GraphX. With snapshots, the Spearman test found correlations for 43% more of the port pairs. To validate correctness, we analyzed the output for evidence of two ground truths related to the application and network topology. First, we expected to see no significant correlations between the port egressing to the master server (server 0) and any other port because the master server did not participate in the distributed computation. Second, we expected to find high correlations between possible ECMP next-hops.

With snapshots, the correlation coefficients matched both expected ground truths. Polling, on the other hand, failed to identify the positive correlations between ECMP ports. As shown by the red boxes in Figure 2.13, the correlations found with polling were either statistically insignificant or, worse, statistically significant but *negative*. Results were qualitatively similar for other applications and ρ values.

2.9. Discussion

Measuring Forwarding State. In Section 2.2.2, we remarked that it may be useful to snapshot forwarding state. While ASIC data planes are not typically able to record table entries directly, they *can* record version information. Specifically, the control plane can ensure every FIB rule and version tags passing packets with a unique ID that is then stored back into processing unit state. A snapshot of the state would then give hints as to the entire network's forwarding state.

Partial Deployment. Speedlight is amenable to partial deployment. In this case, the snapshot would be of participating devices and the communication channels between them. For instance, in a data center, an operator might want for only ToR switches or a particular cluster to be snapshot-enabled.

For snapshots without channel state, the only requirement is that the snapshot header is appended and removed at the proper time. The simplest method is to append the header whenever an ingress processing unit encounters a packet without one, and configure the remaining hosts to ignore IP options in which the snapshot header is contained. If that is not possible (e.g., due to security concerns with IP options), the header should be removed at the last snapshot-enabled device in the packet's path. Causal consistency is maintained even when there are multiple paths between devices.

Snapshots with channel state are slightly more complex. In order to gather channel state, devices must be able to reduce communication to FIFO channels. More specif-

ically, devices must tag packets with the physical path they take between snapshot-enabled devices. We note that in the case of data centers and snapshot-enabled ToRs, this requires only minor modifications to the configuration of existing devices [155].

2.10. Summary

The technique described in this chapter, Synchronized Network Snapshots, and its realization, Speedlight, provide unprecedented visibility into the behavior of the network as a whole. Whether for evaluating a design, diagnosing an issue, or simply trying to understand an existing network, these techniques help to answer critical questions. We demonstrate that this approach is practical by implementing and deploying on a testbed a working version of our system, then using it to collect interesting measurements of real workloads.

CHAPTER 3

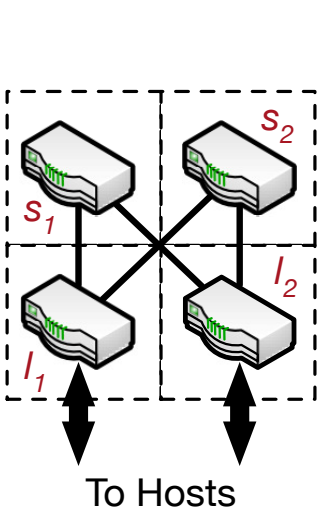
`tpprof`: A Network Traffic Pattern Profiler

3.1. Introduction

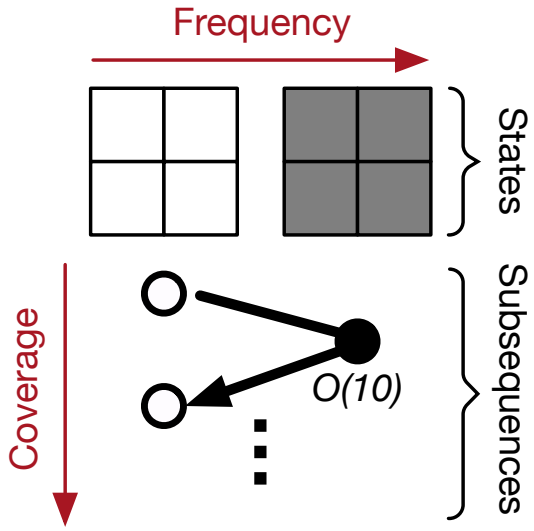
When designing, understanding, or optimizing a computer network, it is often useful to identify common patterns in its usage over time. Often referred to as a **network traffic pattern**, identifying the patterns in which the network spends most of its time can result in useful insights:

- *All-to-all traffic*, which might manifest as uniform utilization of all paths between a set of application nodes, might suggest the importance of bisection bandwidth and guide future provisioning decisions.
- *Chronic stragglers*, where we expect all-to-all traffic but a significant amount of time is spent with only a few flows active, might suggest the need for better sharding and mitigation techniques.
- *Elephant flow dominance*, in which utilization is dominated by isolated path-level hotspots, might guide future provisioning decisions.
- Finally, *synchronized requests/responses*, indicated by repeated bursts of cross-network communication all originating at a single node, might motivate changes in the application or network architecture.

While there are a number of existing tools that capture flow- and switch-level trends (e.g., heavy hitter analysis [194], network tomography [77], or the vast array of net-



(a) Network topology



(b) Common Patterns

(c) Traffic Pattern Scores

Figure 3.1: `tpprof`'s visualizations for (b) common traffic patterns and (c) the TPS score over time for a simple leaf-spine topology, (a). We describe these in more detail later, but in (b) , states are heatmaps of common utilization patterns over the network in (a); darker is hotter. Subsequences are common transition patterns between the aforementioned states. These are ranked by their frequency of occurrence and their cumulative coverage of the profiled run, respectively. The subsequence shows an all-to-all pattern: the network starts unutilized (left state), becomes fully utilized (right state) for 10s of samples, then returns. In (c), `tpprof` is tracking three different known traffic patterns. When the score of any of them crosses the alerting threshold (twice in the figure), `tpprof` deduces that the pattern has occurred in the network.

work analytics suites on the market [79, 1, 2, 3, 4, 5, 6]), identifying prevalent *network*-level patterns typically requires a significant amount of manual effort and specialized

analyses. To determine the presence of synchronized requests/responses, for instance, an operator might need to instrument the start and stop times of all flows in the system, correct for the time drift of different machines, compute the cluster tendencies of the data (e.g., with a Hopkins statistic or heuristic), and distinguish it from all-to-all traffic by examining the sources and destinations of synchronized flows. To determine whether this pattern is a particularly common one would require additional analyses.

Our goal in this work is a tool for the automatic identification of the most prevalent traffic patterns in a network. To that end, we present the design and implementation of `tpprof`, a network traffic pattern profiler.

Similar to traditional application profilers like gprof [75] or Oprofile [50] that have helped programmers understand and improve their software for decades, `tpprof` automatically measures, extracts, and ranks common traffic patterns of individual applications within running networks. It also facilitates the monitoring of known patterns so that, when specific patterns appear in the network, the operator is informed. It does both of these things without modifying applications and without affecting existing network traffic—the only changes we require are to switch monitoring configurations. Examples of both of the above tools in action are shown in Figure 3.1.

Traffic patterns are, unfortunately, significantly more challenging to profile than applications. Traditional profilers benefit from well-defined building blocks (functions or lines of code) connected by well-defined call graphs. In contrast, networks offer little such structure: switch and link utilizations are noisy and measured in real values (Bps); their evolution over time is even less constrained. In the end, two different instances of something as simple as all-to-all traffic will never look exactly the same.

Thus, `tpprof` is built around two novel abstractions: (1) *network states*, which capture an approximate snapshot of a network's device-level utilization and (2) *traffic pattern subsequences*, which represent a finite-state automaton over a sequence of

network states. As hinted above, subsequences serve as both output and input to our system: output in the case of profiling an existing network, input in the case of specifying a traffic pattern alert. In both cases, classification of network states and sub-sequences is approximate and implemented through specialized clustering techniques.

We implement and deploy `tpprof` to a small hardware testbed in order to monitor and profile the traffic patterns of real distributed applications like memcache, Hadoop, Spark, Giraph, and TensorFlow. We demonstrate that, using `tpprof`, we can find meaningful patterns and issues in their behavior. Further, we demonstrate `tpprof`'s utility on larger and more complex networks by profiling a trace taken from one of Facebook's frontend clusters. While our evaluation focuses on data center networks (where interesting and impactful distributed applications are plentiful), `tpprof` and its techniques generalize to arbitrary networks.

Specifically, this dissertation chapter makes the following contributions:

- **Novel abstractions for describing common traffic patterns:** We introduce two abstractions, network states and traffic pattern subsequences, that together enable network operators to easily describe and reason about common traffic patterns. Network states capture similar configurations of approximate utilization of a specific application or set of applications running in a network. Subsequences are then strings of states with bounded repetition that summarize traffic pattern changes over time.

- **Domain-specific algorithms for clustering and ranking both network states and subsequences:** Through empirical analysis of a variety of application traffic patterns, we identify and design algorithms that transform a network trace into the building blocks of traffic patterns. Specifically, we demonstrate through PCA and waypoint analysis of real application traffic that GMMs are

well-suited to capturing first-order similarities between different network utilization patterns. In the case of subsequences, we create a domain-specific clustering algorithm that extracts sequences that are both common and that provide broad coverage of the measured network traces.

- **A language and mechanism for expressing and fuzzily matching known traffic patterns in observed traces:** Finally, to complement the above, we develop a simple grammar for describing traffic patterns and introduce an algorithm that automatically identifies approximate occurrences of known traffic patterns within network traces. Our scoring engine outputs a confidence score that can be used to generate alerts when known traffic patterns appear in observed traces.

Taken together, `tpprof` is, to the best of our knowledge, the first profiling tool for network-wide traffic patterns. Our implementation is in Python and the code is open source.[3]

3.2. The Anatomy of a Traffic Pattern

We begin by introducing the definitions and abstractions on which `tpprof` is built. First and foremost, we define the overall traffic pattern of a network as follows:

Network Traffic Pattern — A function $f(x, t)$ that represents, for an entire network N across a time span $[t_0, t_1]$, the utilization of device $x \in N$ at time $t_0 \leq t \leq t_1$.

We also define, for each application in the network:

Application-specific Traffic Pattern — $f_A(x, t)$, equivalent to the network traffic pattern, but only accounting for a single application or set of applications, A.

For the purposes of our clustering and ranking algorithms, the distinction is unimpor-

[3] https://github.com/eniac/tpprof.

tant; unless otherwise specified, we use 'traffic pattern' to refer to both. Instead, the choice of whether to filter by application is entirely the user's (with the mechanisms in Section 3.4); Regardless, for a given network and overall workload, we note that the traffic pattern of both the network and its constituent applications will typically exhibit predictable and repeated characteristics given a sufficiently long measurement period. These patterns can occur over short time spans of individual packets and flows, or over longer time spans in the form of diurnal or weekday/weekend effects.

A contribution of this dissertation is the decomposition of traffic patterns into a more convenient low-level primitive:

Network Sample — $|N|$ real values that capture an approximate snapshot of $f(x, t)$ for all devices $x \in N$, at a particular time t, and averaged over the last t_Δ seconds.

Network Sample Sequence — A chain of network samples that sample $f(x, t)$ over increments of t_Δ, where t_Δ is bounded by the measurement granularity of the system.

Any traffic pattern can be described in these terms. For the network in Figure 3.1a, the all-to-all pattern in Figure 3.1b is one example. Another is chronic stragglers, which we can describe as a transition between two configurations, assuming a load balanced network: (1) all switches at high utilization and (2) only l_1, l_2 and s_1 at high utilization; or the same but replacing s_1 with s_2.

We can perform a similar exercise for all of the many (possibly application-specific) traffic patterns in the literature, e.g., rack-level hotspots in data centers [166, 78, 63, 121], synchronized behavior of distributed applications [48, 193, 24], and stragglers in data-intensive applications [115, 49, 120]. We do the same for the link- and switch-level traffic patterns that are the focus of most existing automated profiling tools [79, 1, 2, 3, 4, 5, 6].

While not necessarily the way these patterns were described originally, sequences of

network samples provide a general primitive with which we can represent arbitrary patterns.

3.3. `tpprof` Design Overview

Our goal in this work is the design and implementation of a *profiler* for network and application-specific traffic patterns. Our system, `tpprof`, is intended to identify traffic patterns, rank them in prevalence, and assist network operators in monitoring for their recurrence. Like other profiling tools, `tpprof` is not intended to improve networks directly; rather, its focus is on assisting users with designing, understanding, and optimizing them.

On that note, we take inspiration from traditional sampling profilers like gprof [75], Oprofile [50], and Valgrind [137]. These profilers take an unmodified application and they periodically sample system state (e.g., stack traces) to produce a statistical profile of the target application. Early instantiations solely sampled program counters; over time, they expanded to capture trends in function utilization and call graph traversal.

`tpprof` uses a similar approach to construct profiles of traffic patterns. To that end, network samples and sequences of samples present an attractive substrate. In principle, a sequence of network samples creates a statistical profile of an application's network utilization. Unfortunately, these samples are unlikely to ever repeat: small differences in application processing time, workload, and background traffic can cause substantive differences in traffic, as can slight noise in the sampling frequency of the measurement framework. Extracting patterns from raw samples is challenging.

Core abstractions. To address this challenge, we introduce two additional abstractions:

Network State — A class of network samples defined by a single, n-device network

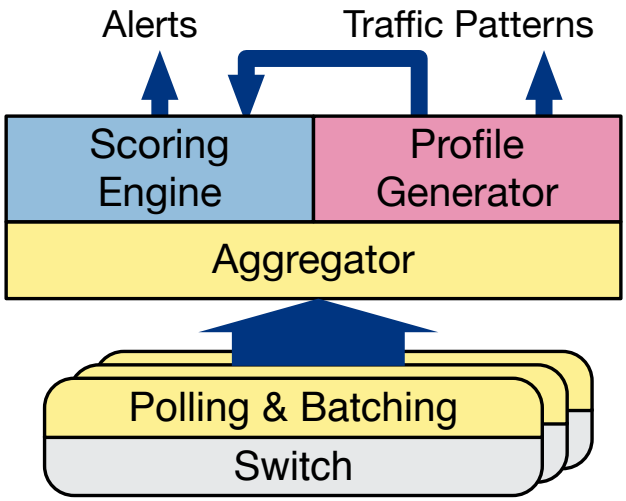


Figure 3.2: The overall architecture of `tpprof`. `tpprof` polls, batches, and aggregates switch counters from the network. These are fed into (1) a scoring engine that alerts on detection of known patterns and (2) a profile generator that extracts common traffic patterns from the gathered trace.

sample, S, and n variance values, $\bar{v}$ such that S is a centroid of the multivariate normal distribution with shape defined by $\bar{v}$.

Network State Subsequences — A class of sequences of network states that allows for bounded repetition of states. A state subsequence can be represented as a regular expression or finite state automata of network states.

Multiple network samples can be mapped to a single network state and multiple sample sequences can be mapped to a single state subsequence. These abstractions are tolerant to noise by design: variations of link utilization from sample to sample are smoothed by our method of extracting network states; variations in the evolution of those samples over time are smoothed by our method of extracting subsequences. The precise construction of both of the above elements is described in Sections 3.5.1 and 3.5.2.

Components. `tpprof` consists of three primary components:

1. A configurable *sampling framework* that periodically samples the device-level utilization of a specified application, set of applications, or the full network (Section 3.4).

2. A *profiling tool* for the automatic extraction and visualization of the most common states and state subsequences in the captured data (Section 3.5).

3. An *alerting system* that scores incoming traces against a set of user-defined patterns using a fuzzy string search in order to facilitate network automation (Section 3.6).

Of the above, only (1) affects the network itself; (2) and (3) occur out-of-band. As such, the overhead of `tpprof` is minimal: in the case of an non-application-specific traffic pattern, little is required beyond an SNMP poller; application-specific patterns only require simple `iptables` and switch configuration changes on top of that.

Our current implementation leverages programmable switches and a recently proposed network-wide monitoring framework [183]. This provides slightly more control and accuracy than an implementation based on top of traditional switches, but it is not a strict requirement; we detail both approaches in Section 3.4.

Workflow. `tpprof` profiles production networks. A typical workflow thus proceeds as follows. First, users specify three configuration parameters: the start time a, the end time b, and the sampling interval i. The network can optionally be configured to track certain applications separately. Regardless, a centralized service periodically polls the byte counters of the entire network between time a and b, with interval i.

The centralized service will stream the data through a set of scoring algorithms that quantify the prevalence of a set of target patterns in the measured trace. If the score of the trace exceeds a threshold for a given pattern, an alert will be generated. By default, the measurement data is not stored. This changes when users request

a profile, i.e., a visualization, of common traffic patterns in the network. In this case, raw network samples are stored for a specified profiling duration for clustering and analysis. The resulting profile can be used to construct additional pattern alerts or analyzed separately. The remainder of this chapter describes each of the three components of `tpprof` in more detail.

3.4. Sampling Framework

`tpprof`'s sampling framework continually polls a custom set of switch counters to capture traffic patterns. Most production networks already implement some form of this—`tpprof` can piggyback on these existing polling suites. `tpprof` is, however, parameterized by at least two configuration options.

- *Application filters:* To profile application-specific traffic patterns, users must provide a proper filter for the traffic in question. In `tpprof`, this takes the form of `iptables` rules. Any filter that can be expressed as an `iptables` rule is allowed. Thus, multiple applications can be captured by a single filter and different flows from the same application can be split into different filters by port, packet type, etc. All traffic matching installed filters are marked with a special set of bits, e.g., in the DSCP field of the packet header. We term the value of these bits a *filterid*.

- *Sampling interval:* Users must also specify an interval, t_Δ, at which `tpprof`'s sampling framework will poll all devices in the network. This interval is common to the entire system, so the network and all application-specific traffic patterns will be read at this rate. Though this is a user-defined value, we anticipate that it should be set to the minimum value feasible for the target network without incurring sample loss. We note that, because the raw data is discarded after alert pattern matching, measurement data storage capacity is not a bottleneck in `tpprof`.

3.4.1. Counter Implementation and Sampling

Network devices in a `tpprof`-enabled network track a set of device-level application-specific byte counters corresponding to the space of possible *filterid*s. For every packet traversing the switch, the counter associated with the specified *filterid* is incremented by the size of the packet; all categories summed will give the cumulative byte counter of the device. In this design, the network is never reconfigured; instead, users associate applications to *filterid*s directly through the `iptables` rules at every end host.

`tpprof` samples these counters at an interval of t_Δ via a recently proposed measurement primitive, Speedlight. For brevity, we omit the details of its operation and refer interested readers to its non-channel-state variant [183, 182]. At a high level, the primitive is that of a synchronized, causally consistent snapshot of network-wide switch counters. Compared to SNMP and other naïve polling tools, Speedlight provides increased accuracy and low minimum sampling interval, both of which are useful when profiling network traffic patterns.

Alternative implementations. We note that, at its core, the only requirement of `tpprof` is for configurable counters and a method to periodically poll all such counters in the network. There are other implementations that satisfy this requirement.

For instance, most modern switches typically include support for configurable ACL entries with per-entry counters. This approach has the advantage that it can be implemented without end host cooperation.

Class of Service (CoS) counters are similarly promising. Note that, if application-specific tracking is not required, periodic SNMP polling is sufficient.

3.4.2. Batching and Aggregation

While it is possible to directly transmit polled counter results to a centralized profiling service, the scale of measurement data collected by `tpprof` necessitates careful

handling. In particular, there are two issues we must address: decreasing overhead and handling sample loss.

For the first, to decrease the number of messages and the overhead per sample, `tpprof` agents running on each network device assemble results locally before shipping batches of size B in the following format:

```
sampleBatch: {
  switch: <SWITCH_ID>,
  indexes: [i : <SAMPLE_ID> for i from 0 to B],
  app1_bytes: [i : <BYTE_COUNT> for i from 0 to B],
  ...
  appM_bytes: [i : <BYTE_COUNT> for i from 0 to B]}
```

`indexes[k]` and `*_bytes[k]` should correspond to a single network sample. Gaps in the samples, e.g., from failures or measurement packet drops, will manifest as gaps in the `indexes` array. In these cases, `tpprof` attempts to interpolate values by taking the difference between byte counters before and after any gap and averaging the difference over the length of the gap. If the device has rebooted or if it stays down for too long, we will treat the device as 'failed' during the missing measurement intervals. 'Failed' devices are excluded from profiling and treated as wildcards during alerting. Note that reboots are also excluded from interpolation as we do not know how much traffic was sent before the counter was reset.

Storing data for profiling. While `tpprof` does not store raw counter values in the common case, raw values are necessary for generating profiles. Profiles are, therefore, executed on-demand using the API:

```
start_profile(start, end, filter_id)
```

The duration of collection should be long enough to capture a representative slice

of behavior. In general, longer is better, but this may be subject to limitations of sample storage space and the user's timeline. `filter_id` $= -1$ indicates the sum of all application-specific counters.

3.5. The `tpprof` Profiling Tool

We first discuss how `tpprof` extracts and ranks traffic patterns before delving into the scoring and alerting system in Section 3.6.

To that end, the output of the previous subsection (3.4.2) is a network sample sequence, i.e., a sequence of n-device samples of network utilization. Using that, the output of the `tpprof` profiler is a ranked list of network states and a ranked list of state subsequences, as sketched by Figure 3.1b and demonstrated in Section 3.7. `tpprof` achieves this using a pair of domain-specific clustering techniques that are designed to capture first-order patterns in network traffic.

The first challenge in identifying meaningful traffic patterns is the inherent noise present in a trace of network samples. Small variations in workload, TCP effects, background traffic, or any number of other factors mean that, most likely, no two network samples will look exactly alike.

To de-noise the data, `tpprof` summarizes network samples into a small number of distinct network states. We can naturally frame this as a clustering problem, where the points to be clustered are the n-element vectors representing network samples. Clustering has been used to great effect in a number of fields, from image segmentation to recommendation systems and anomaly detection; each of these has its own set of challenges and associated clustering algorithms.

Network state extraction is no different in that regard. In this work, we leverage empirical analysis of a variety of applications and traces to identify and design algorithms suited to the domain. Applications observed include Hadoop, Giraph, TensorFlow,

Spark, Memcache, and a trace from a production Facebook frontend cluster (see Section 3.7 for their details).

Dimensionality reduction. Before delving into `tpprof`'s clustering algorithm, we note that, in general, networks present a particular challenge to clustering because of their high device counts. Profiling the ToRs of a 48-rack data center cluster, for instance, might result in a 48-feature input vector, which prior work has indicated might be too many dimensions for typical distance metrics [37].

The general solution to this well known 'curse of dimensionality' [35] is transforming the data into a lower-dimensional space before clustering. The simplest approach is to cluster on only a subset of features. While this works in other domains, it is not well suited for our problem because the load on *every* device may be important. Instead, `tpprof` preprocesses data with Principle Component Analysis (PCA) [66], which derives a small set of features that are an orthogonal linear transformation of the original features. Said differently, PCA removes redundancies in the original data by creating a new set of independent features that explain most of the variability in the original data.

PCA is most effective when features are strongly correlated, and there is good reason to believe that this is true in our domain. Recent work [49] shows that network usage is highly correlated, driven by the data-parallelism in distributed systems [83, 187]. Analysis of the Facebook traffic trace verifies this: each ToR had strong and statistically significant correlations ($r > .7$, $p < .001$) with an average of 3.25 other ToRs. The applications we profiled showed similar results.

Figure 3.3 measures the effect of PCA on that data, gauged by plotting the number of PCA dimensions (i.e., features) versus explained variance. All other traces we obtained showed similar results. A value of 1 means that a PCA transformation to

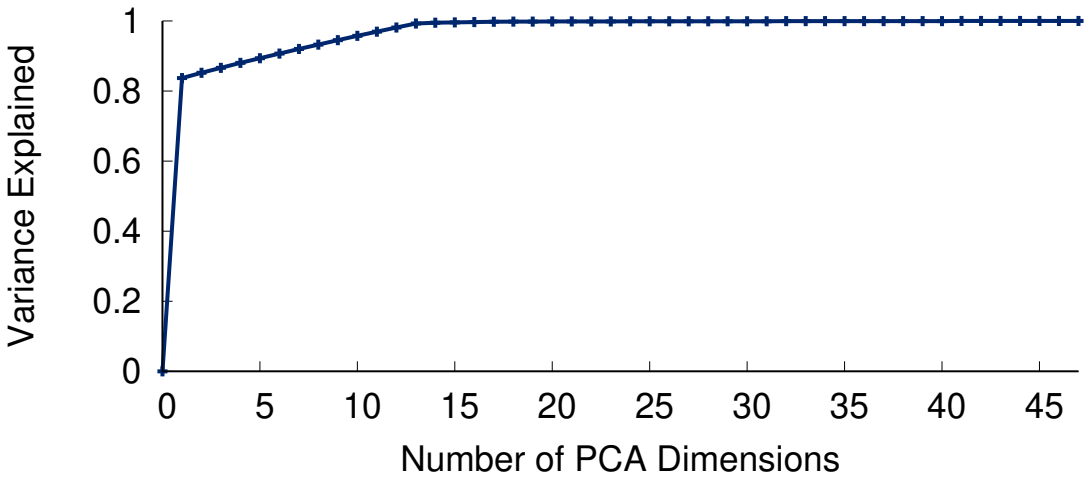


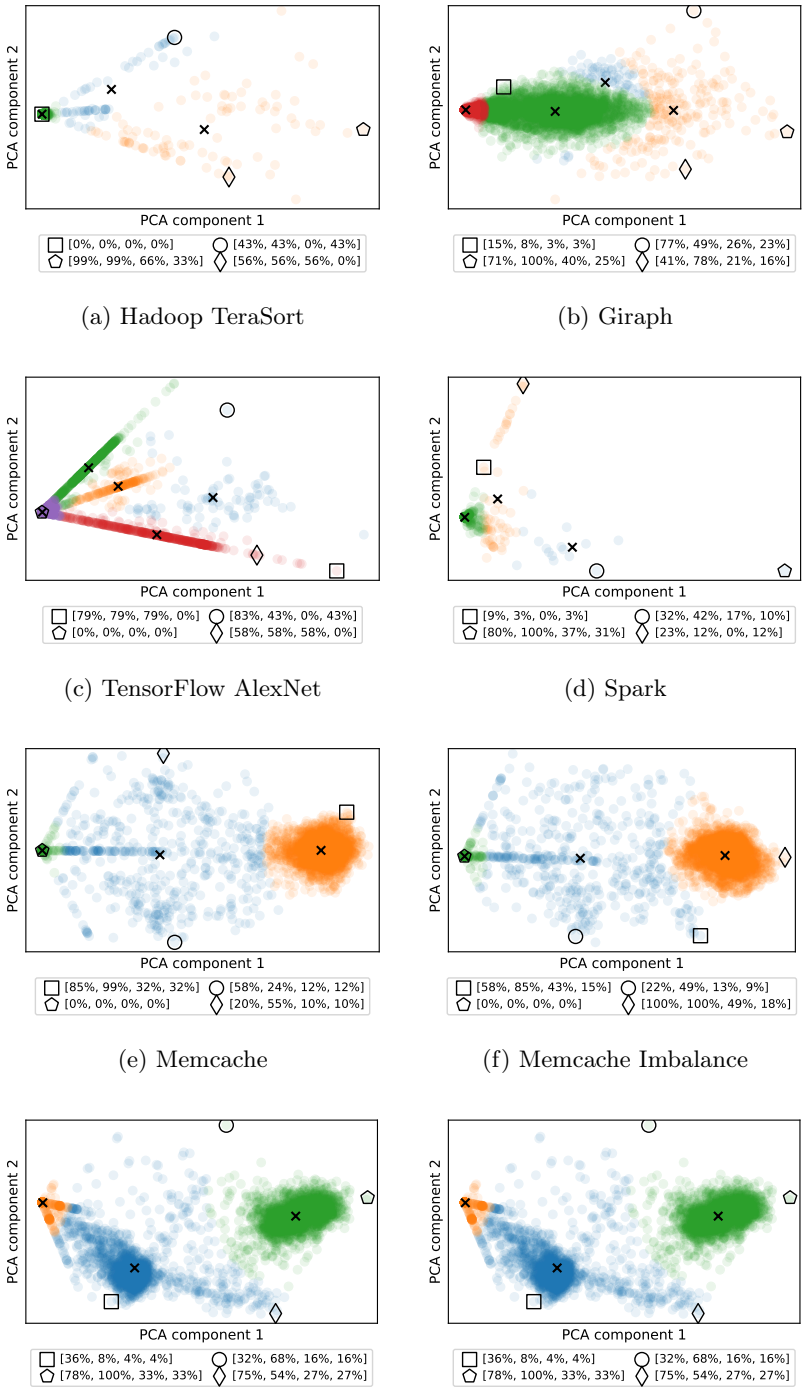
Figure 3.3: Covariance explained by different numbers of PCA dimensions. Dataset is a trace of utilization over 48 ToR switches in a Facebook frontend cluster.

K dimensions preserved all the information contained in the original data with 48 dimensions. Even for this large and complex trace, one dimension already explains over 80% of the variance and two dimensions explain ∼85%. Striking a balance between clustering efficacy and explained variance, `tpprof` projects all data into 2D by default. This can be adjusted depending on the data.

3.5.1. Network States

Gaussian Mixture Models (GMMs) for sample classification. `tpprof` clusters around typical network variations through its use of GMMs. To demonstrate this effect, we consider the 2D PCA projections described above and visualized, for our set of profiled applications and traces, in Figure 3.4. To help interpret points in the PCA space, we also plot network load at 4 extreme *waypoints* along the convex hull of each trace. We observe two general cluster shapes in the projected data: 'rays' and 'clouds'.

- **Rays**, like the ones prominent in Figures 3.4a and 3.4c, are typically associated with rising or falling utilization on a set of highly correlated nodes. We can see this effect most clearly in Figure 3.4c through the relationship between ⬠, ♢,



(a) Hadoop TeraSort

(b) Giraph

(c) TensorFlow AlexNet

(d) Spark

(e) Memcache

(f) Memcache Imbalance

(g) Memcache Straggler

(h) Facebook Trace

Figure 3.4: Network samples projected into a 2-dimensional PCA space. Cluster centers are marked with x's. Shaped-markers map points in the space to sample vectors [l1, l2, s1, s2] (see Figure 3.1a) or, for the Facebook trace, average utilization.

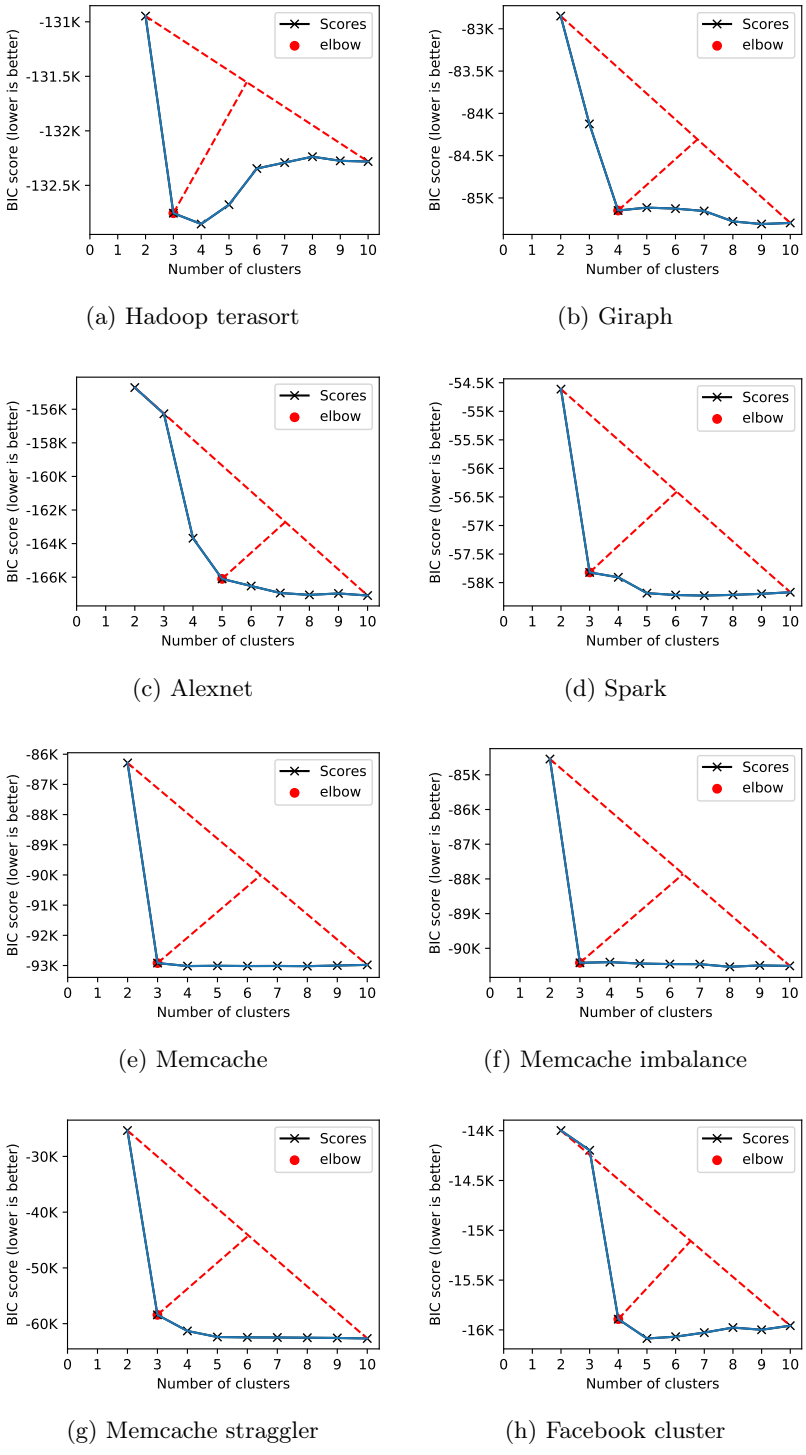
and □. Compare their utilizations with that of ○.

- **Clouds**, like the ones in Figures 3.4b and 3.4g, typically characterize samples that are similar in configuration, but separated by noise that offsets points by a small amount in all directions of the PCA space. These clouds can be more or less dense, depending on the coherence of the pattern. The memcache variants, for instance, exhibit strong all-to-all behavior, which manifests as dense clouds to the right of the PCA plots.

Synchronized behavior and noise around a specific configuration capture most of the key behavior in our empirical tests. For these two types of clusters, GMMs are known to perform well. GMMs model a cluster as a multivariate Gaussian with independent parameters for each dimension of the data. This independence provides the flexibility for clusters to fit both types of clusters with arbitrary densities. We fit GMMs to the data using the expectation-maximization algorithm from Scikit-learn [146], which finds clusters that are each defined by a centroid sample and a vector of per-feature variances.

Automated detection of cluster count. GMMs are defined in terms of a fixed number of clusters, K. `tpprof` selects K automatically by using a Bayesian Information Criteria (BIC) score. Informally, a better (lower) BIC score means that a specific clustering, if used as a generative function, is more likely to produce the observed data.

We note, however, that BIC scores tend to improve as K increases, but a high number of clusters can overfit the data. To overcome this issue, we select a K value at which the benefit gained by adding an extra cluster starts to diminish. Finding such "elbows", or points of maximum curvature, is a common problem in machine learning and systems research. We use the Kneedle method [159], a simple but general



(a) Hadoop terasort

(b) Giraph

(c) Alexnet

(d) Spark

(e) Memcache

(f) Memcache imbalance

(g) Memcache straggler

(h) Facebook cluster

Figure 3.5: Selecting the number of clusters with Bayesian Information Criteria (BIC) and the elbow heuristic.

algorithm based on the intuition that the point of maximum curvature in a convex and decreasing curve is its local minima when rotated θ degrees counter-clockwise about (xmin, ymin) through the line formed by the points (xmin, ymin) and (xmax, ymax). Specifically, we plot the BIC score versus K and draw a line segment connecting the points for $K = 2$ and a configured maximum of $K = 10$, which we set based on the typical working set capacity of humans. The optimal value of K is given by the point furthest from that line. Figure 3.5 shows the results of this analysis for the applications and traces introduced above.

3.5.2. Network State Subsequences

Network state subsequences extend network states to capture temporal patterns in traffic. Like states, subsequences require compression of the full sequence of samples taken during the profiling run into a small set of representative patterns. Unlike states, existing sequence-based clustering algorithms are a poor fit for network traffic patterns.

To see why identifying and ranking network state subsequences is challenging, consider a strawman solution: take all possible subsequences of the trace and count their frequencies, e.g., the trace ABC would result in the following subsequences $\{A \times 1, B \times 1, C \times 1, AB \times 1, BC \times 1, ABC \times 1\}$.

Challenge 1: (a) $A^5 = AAAAA$ versus (b) $A^5B \ldots AAB \ldots AAB$

> Intuitively, the interesting bit of sequence (a) is that there is a long run of A's. The strawman solution will instead output that the most common subsequence and frequency is the single state $A \times 5$, followed by $AA \times 4$, etc. With the naïve approach, short subsequences will always take priority; in fact, we can prove that subsequences will *never* beat their member states. On the other hand, sequence (b) demonstrates a case where it might be useful to be able to observe the shorter subsequences. In this case, greedily setting aside the A^5 would miss the third

```
param stateSequence[# samples in the trace]: Full sequence of network states.
param minFreq: The minimum number of subsequence occurrences before it is counted
  as a 'common' subsequence.
Function getSubSequences:
    for targetLength : len(stateSequence) to 2 do
        maxStart ← len(stateSequence) − targetLength
        for start : 0 to maxStart do
            end ← start + targetLength
            /* Skip taken ranges */
            if [start,end] contained in takenRanges then
                continue
            /* Add if it meets minFreq */
            subseq ← log10Merge(stateSequence[start:end])
            if (# subseq observations) ≥ minFreq then
                Add (start, end) to takenRanges after loop
                Increment subseqs[subseq]
            else
                Hold subseq until the threshold is reached
    subsequenceCoverage ← computeCoverage(subseqs)
    totalCoverage ← computeTotalCoverage(subseqs)
    return subseqs, subsequenceCoverage, and totalCoverage
```

Figure 3.6: Pseudocode for finding common subsequences in a sequence of network states.

occurrence of AAB, which is arguably the more important pattern.

Challenge 2: $XA^{39}Y \ldots XA^{40}Y \ldots XA^{41}Y$

The strawman solution also performs poorly with similar, but not identical ranges. While it may find here that there are long strings of As, or even that X is typically followed by As, or that Y is typically preceded by As, it will fail to find that As are typically sandwiched between X and Y. Variance in duration is common in networks, where measurement timing, available capacity, and workload size changes frequently.

Challenge 3: $(AB)^3(CDEFGHIJKLMNOPQRSTUVXYZ)^2$

Finally, we note that frequency itself is not an ideal metric. Consider the above trace. The longer trace is much rarer and more interesting, but a pure frequency analysis will rank AB higher in importance.

`tpprof`'s subsequence extraction (outlined in Figure 3.6) addresses these challenges through a series of rules, which we describe below. Line numbers reference Figure 3.6.

Only consider subsequences of length 2+ [Line 2]. While knowing the most frequent single states is useful, the goal of extraction is to capture patterns in traffic. We, therefore, prune subsequences of length 1 from consideration and list the relative frequency of single states separately.

Ignore strict subsequences [Lines 6–7]. To better summarize cases like Challenge 1(a), we exclude any subsequence that is wholly contained within another subsequence. We implement this efficiently using two data structures: (1) *takenRanges*, a list of existing subsequences sorted by *start*, and (2) a *heap*-based index of the currently overlapping subsequences, sorted by *end* (not shown).

Frequency threshold before a subsequence is counted [Lines 9–13]. The above rule, applied directly, might produce a single subsequence encompassing the entire trace. To account for this, we set a minimum frequency threshold, *minFreq*, before which the subsequence is not counted. We note that a lower value of *minFreq* promotes the inclusion of longer but sparser subsequences, while a higher value favors many, shorter subsequences. `tpprof` automatically tunes this value using the `hyperopt` library to optimize for 'total coverage', a metric we describe at the end of this section.

Log10 repetition frequencies [Line 8]. To handle cases like that of Challenge 2, common in network traces, we compress repetitive states into the nearest power of 10. Doing so ignores small differences in duration while still retaining the length's rough magnitude.

Coverage rather than frequency [Lines 14–15]. As evidenced in Challenge 3,

differences in the length of subsequences and the ability of subsequences to overlap diminish the utility of frequency as a way to reason about the relative importance of different subsequences. Instead, we propose *coverage* as a metric for ranking subsequences and for hyperparameter-tuning minFreq. Coverage measures, for either a single subsequence or the union of all subsequences, the cumulative fraction of states in the stateSequence that are included in at least one subsequence.

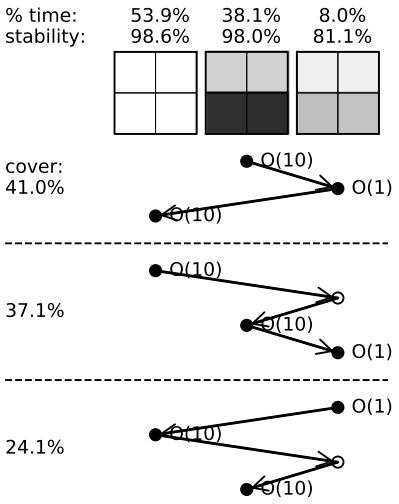
We encourage the reader to step through several short examples of network state sequences to see why the above rules produce intuitive results.

3.5.3. Example Visualization: memcached

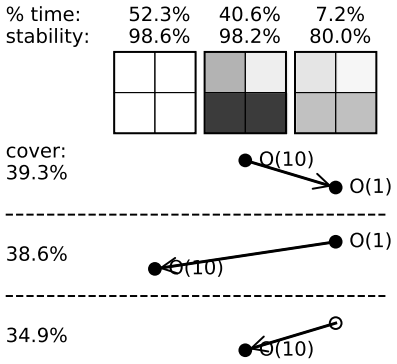
To tie the above discussion together and showcase the utility of `tpprof`, we present to the reader several real profiles produced by the `tpprof` tool suite. See Section 3.7 for a description of the hardware testbed used in these tests.

As a baseline, we first look at a memcache workload generated using the `memaslap` [7] benchmarking utility, running in isolation. Each machine in the testbed was configured as a memcache server with 64 B keys and 1024 B values. Gets and sets were randomly generated from two machines—one in each rack—with a ratio of 9:1. In this simple test, the two memcache clients, every 6 s, will simultaneously begin performing 290k get/set operations.We profiled this behavior, collecting a total of 7000 network samples at a 50 ms interval.

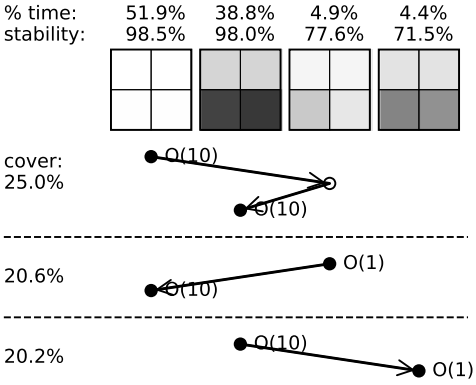
Visualization structure. Figure 3.7a shows the `tpprof` profile for this run. Like Figure 3.1b, heatmaps of *network state* are at the top of the figure and the most common *network state subsequences* are below. Each heatmap shows the centroid of the sample clusters it represents. We add to this the state's *% time* (the amount of time the network spends in the state) as well as its *stability* (the probability that the network, once in the state, will stay there); states are sorted by % time.



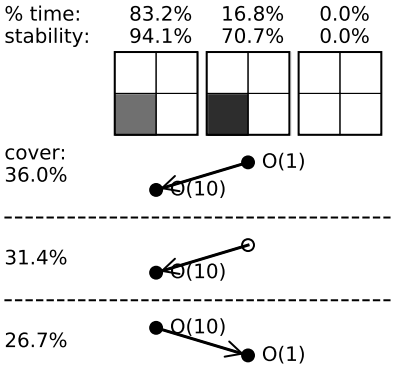
(a) memcache – Baseline



(b) memcache – Imbalance



(c) memcache – Straggler



(d) Cross-traffic during Figure 3.7c

Figure 3.7: `tpprof` profiles of memcache in three different environments (Figure 3.7a– 3.7c), plus a profile of cross-traffic (Figure 3.7d) active during Figure 3.7c.

For subsequences, we include the top three by coverage; more can be generated on demand. Subsequences are depicted with a series of points (representing states) connected by arrows (denoting transitions between states). The points align horizontally with the states they represent. Solid points accompanied with an O(x) label indicate an x–$10x$ repetition of that state. The number on the left of each subsequence is the percentage of the trace that it covers. Note that coverage can add to more than 100%

due to overlapping subsequences.

Observations. We can observe several characteristics in Figure 3.7a. First, we can see that there are three states in which the network spends its time. In the one that accounts for more than half the trace, the network is unutilized. The other two show different states of even leaf and even spine utilization, indicating that the network is relatively balanced when it is being used. Note that the leaves of the network are consistently hotter than the spines due to rack-internal communication.

As expected of the workload, the subsequences of the profile show a trace composed of *on-off* periods of *all-to-all* traffic. We can also deduce from the duration of repetitions that the on and off periods both last on the order of seconds. Further, we can infer that the network takes time to ramp up/down from full utilization. This is inferred from the presence of the (L-to-R) 3rd state and the absence of direct transitions between states 1 and 2. Ramp ups seem to be an order of magnitude faster than ramp downs.

`tpprof`'s observations can inform network and application changes. For example, if an operator were to see a similar profile in practice, she could conclude that load balancing is not an issue. Instead, a more promising approach would be to either desynchronize traffic to spread out utilization over time or augment the leaf switches with additional capacity.

Case Study #1: Detecting Load Imbalance

`tpprof` can also help to detect acute problems in networks. As a case study, we artificially introduced an ECMP misconfiguration [196] into the network. Specifically, we configured one of the ToR switches to only use the left spine; otherwise, the workload is identical to Figure 3.7a. Figure 3.7b shows the output of our `tpprof`'s Python-based visualizer. An operator comparing this profile to that of the baseline

would be able to see the new and stark differences between the two spines in all states with load and conclude that ECMP was not doing its job. While imbalance can also be due to elephant flows and hash collisions, the fact that this happens consistently and always with the same spine points to a structural issue.

Case Study #2: Debugging a Noisy Neighbor

As another case study, we use `tpprof` to debug an apparent straggler in the system. In this experiment, we add a heavy background flow between two hosts connected to the lower-left leaf, l_1. Figure 3.7c shows the profile in question. From this profile, an operator can observe that, in 5–10% of samples, there is a slight bias toward l_1 while the other leaf is largely un-utilized. These samples are summarized in the right two network states. If the operator is expecting an even all-to-all pattern like the one in Figure 3.7a, these states would lead her to suspect that a task in the system is straggling.

`tpprof`'s ability to profile concurrent applications independently can also help to diagnose this issue. In particular, she can view the profile of non-memcache traffic present during the same profiling period. In this case, `tpprof` would provide her with Figure 3.7d, which clearly shows a competing flow or set of flows within l_1.

3.6. Traffic Pattern Scoring

The `tpprof` components described in prior sections allow users to profile their networks and find prominent traffic patterns. In many cases, after finding certain patterns, users are likely to want to know if (or when) they occur in the future. The `tpprof` traffic pattern scoring engine solves this problem. The key challenge is designing both a language that makes it simple for users to specify pattern signatures and also an algorithm efficient enough to detect those patterns in realtime.

Traffic pattern signatures. A traffic pattern signature describes the approximate

⟨*signature*⟩ ::= **{ (**⟨*target state set*⟩**) ;** ⟨*target sequence*⟩ **}**

⟨*target state set*⟩ ::= ⟨*target state*⟩ **,** ⟨*target state set*⟩

⟨*target state*⟩ ::= $\vec{utilization}$

⟨*target sequence (P)*⟩ ::= ⟨*target state*⟩ | ∼⟨*P*⟩
| ⟨*P*⟩∧⟨*P*⟩ | ⟨*P*⟩∨⟨*P*⟩
| ⟨*P*⟩* | ⟨*P*⟩**{** min repetitions **,** max repetitions**}**

Figure 3.8: Definition of a traffic pattern signature.

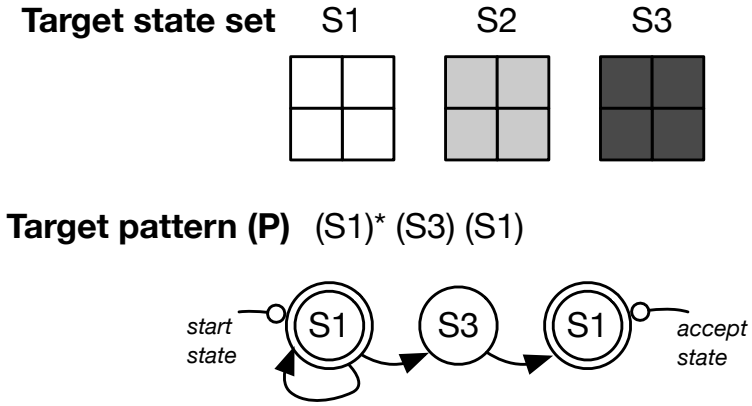


Figure 3.9: An example traffic pattern signature that detects a synchronized all-to-all burst.

spatial and temporal characteristics of a traffic pattern. It is defined by the grammar in Figure 3.8 and has two components.

- A set of *target network states* describe the approximate samples that are likely to be observed during the traffic pattern. These can be generated from prior profiling runs or manually specified.
- A *target subsequence*, written as a regular expression, that estimates how the network transitions between the target states during the pattern.

As an example, Figure 3.9 illustrates a signature to detect a synchronized all-to-all burst of traffic in our example topology. The target states in the signature are: S_1, 0% utilization on all links; S_2, 50% utilization on all links; and S_3, 100% utilization on all links. The signature's target subsequence is, thus, one in which the network

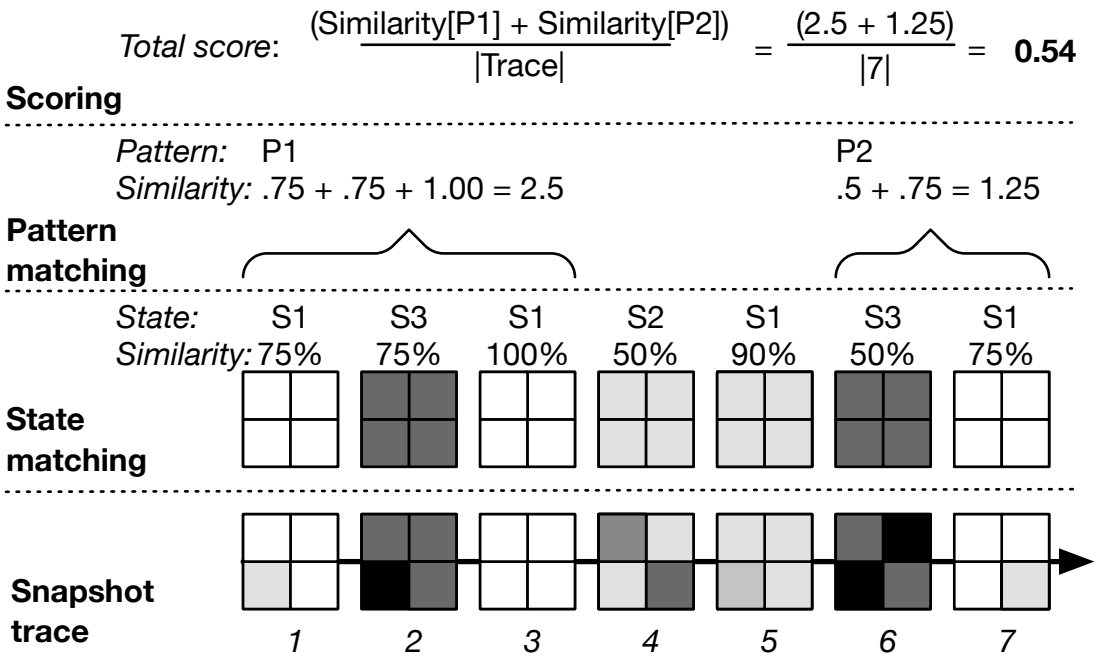


Figure 3.10: Matching and scoring a sample trace against the all-all signature in Figure 3.9.

is in S_1 before transitioning to S_3 (i.e., high, all-to-all utilization) and immediately going back to S_1, signaling a quick end to the all-to-all utilization.

Scoring signatures. `tpprof`'s Traffic Pattern Score (TPS) algorithm quantifies a signature's prominence in a network sample sequence by finding and scoring subsequences that are similar to it. This amounts to a streaming fuzzy string search. Figure 3.10 illustrates the scoring algorithm for the all-to-all signature in Figure 3.9, while Figure 3.11 provides psuedocode of our streaming implementation. There are three steps.

1. *State matching:* The TPS algorithm first maps each incoming sample to the most similar target state, transforming the stream of samples into an *intermediate stream*.

2. *Pattern matching:* It then scans the intermediate stream for the target subsequence using a finite automata [90]. A match occurs when the automaton

```
1  signature ← (targetStates, regexp)
2  Function TPSGrep(signature, sampleStream):
3      Initialize matchStream
4      compile_patterns(matchStream)
5      scoreBuf ←[]
6      offset ← 0
7      for each (sample, timestamp) in trafficPattern do
8          /* Identify most similar target state.*/
9          stateSymbol ← nearestNeighbor(sample,targetStates)
10         similarity ← |netState − sample|
11         /* Track scores for up to BUF_LIM of the last samples. */
12         scoreBuf.append(score)
13         if len(scoreBuf)>BUF_LIM then
14             scoreBuf.pop()
15             offset ← offset + 1
16         /*Invoke HyperScan to update stream. */
17         (begin, end) = scan(matchStream, stateSymbol)
18         /*If a match occurred, calculate and emit a score. */
19         if end ≠ NULL then
20             emit sum(scoreBuf[begin-offset:end-offset]
```

Figure 3.11: The streaming TPS algorithm.

reaches an accept state, at which point it is executed in reverse to identify the start point of the longest matching subsequence.

3. *Match scoring:* A match indicates that the exact target subsequence has been found in the intermediate stream; however, how this relates to the underlying sample stream is unclear. Thus, the final step is to score match strength by calculating the average similarity between the two streams during the subsequence.

Writing signatures. There are two sources for signatures. First, they can be automatically generated by the profiler, from the network state subsequences it identifies. This allows the TPS algorithm to automatically identify future reoccurrences of events identified with the `tpprof` profiler.

Second, users can manually write signatures that characterize the most important attributes of a traffic pattern. Since TPSes use a fuzzy algorithm, patterns do not need to be exact. Instead, they can be defined programmatically. With the three

Pattern	Signature	State Definitions
Short all-all	$S_1^* S_2 \{1, 10\} S_1$	$\{S_1\}$=`N:0.0`,$\{S_s\}$=N`:0.5`
Long all-all	$S_1^* S_2 \{10, 100\} S_1$	$\{S_1\}$=`N:0.0`,$\{S_s\}$=N`:0.5`
Hotspots	$(S_1 \mid S_2 \mid S_3 \mid S_4)\{10, 100\}$	$\{S_1, ..., S_4\}$=`{(x:1.0, -x:0.0) for x` $\in N$`}`
Imbalance	$S_1^* \mid S_2^*$	$\{S_1, S_2\}$=`{(x:1.0, -x:0.0) for x` $\in (s_1, s_2)$`}`
Stragglers	$(S_1 \mid S_2 \mid S_3)^* S_3$	$\{S_1,S_2,S_3\}$=`{(`l_1`:v, -`l_1`:0.0) for v ∈ (0.1,0.01,0.0)}`

Table 3.1: Traffic pattern signatures for a leaf-spine network N with spines (s_1, s_2) and leaves (l_1, l_2).

primitives described below, users can express simple but powerful signatures.

- *State definition*, e.g., `(x:v, y:u)`, which defines a state with switch `x` having utilization `v` and switch `y` having utilization `u`.
- *Set assignment*, e.g., `X:v`. This sets every switch `x` ∈ `X` to utilization value `v`.
- *Iteration (over sets or switches)* e.g., `{(x:v) for x ∈ X}`, which defines a set of states: one state for each switch in `X`, defining that switch to utilization value `v`.

Table 3.1 lists five example signatures written with these primitives. We evaluate them later in Section 3.7.3.

3.7. Implementation and Evaluation

`tpprof` is implemented in Python/C++ as a standalone service that aggregates samples, profiles them, and scores them for the presence of known traffic patterns, as described in the previous sections. Each of the profiles shown in this section is a real output of `tpprof`, generated programmatically using Python and Matplotlib 3.1.1. The requisite counters and polling/batching components that run on each device are implemented in P4 and Python, respectively. Traffic Pattern Scoring is implemented in C++ using `hyperscan` [90].

Hardware testbed. To verify the utility of `tpprof` and its outputs, we used it to profile and score the traffic patterns of real applications running on a small hardware testbed consisting of a Barefoot Wedge100BF-32X programmable switch connected to six servers with Intel(R) Xeon(R) Silver 4110 CPUs via 25 GbE links. The testbed is configured to emulate a small leaf-spine cluster like the one in Figure 3.1a. To implement this network, we split the Wedge100BF switch into 4 fully isolated logical switches. Each logical switch runs ECMP to balance load across paths.

Application workloads. On our hardware tested, we profile four popular networked applications, in addition to the memcache evaluation in Section 3.5.3:

1. Hadoop running a TeraSort [26] benchmark workload with 5B rows of data. Our Hadoop instance ran version 2.9.0 with YARN [27] on 10 mappers and 8 reducers spread across the 5 servers (and 1 master).

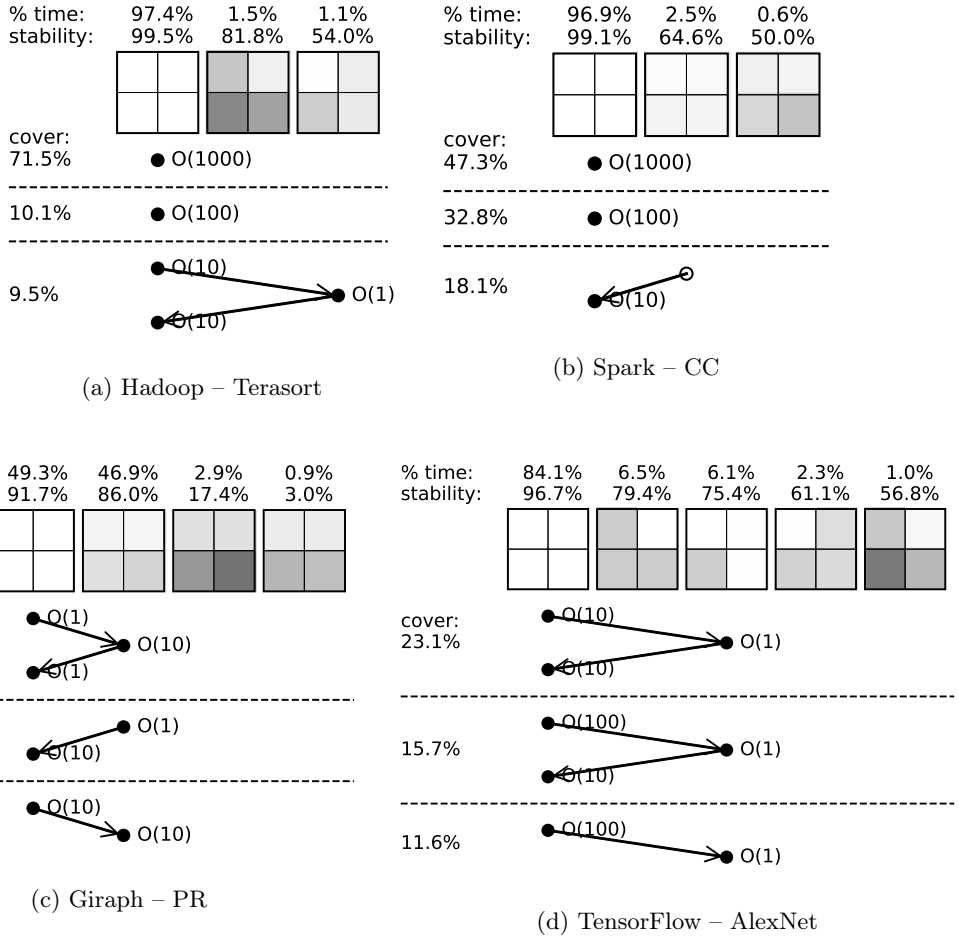
2. Spark's GraphX [29] running a connected components benchmark workload with 1.24M vertices. We ran Spark 2.2.1 with Yarn on 5 servers (and 1 master).

3. Giraph [25] running a PageRank synthetic benchmark workload with 120,000 vertices and 3,000 edges on each vertex. We used 23 workers in total across our 6 servers.

4. TensorFlow running the AlexNet [106] image processing model with 1 server managing parameters and 5 workers. We used ILSVRC 2012 data for training.

Unless otherwise specified, these applications were run in the presence of background TCP traffic derived from a well-known trace of a large cluster running data mining jobs [21]. Profiles are of the target application only.

Large-scale trace. To augment our small testbed, we also profile packet traces of 48 Top-of-Rack switches from three of Facebook's production clusters: a frontend



(a) Hadoop – Terasort

(b) Spark – CC

(c) Giraph – PR

(d) TensorFlow – AlexNet

Figure 3.12: Profiles of more complex applications running with realistic background traffic.

cluster, a database cluster, and a Hadoop cluster. As the datasets are sampled by a factor of 30,000, we divide the timestamps by 30,000 to obtain an approximate representation of a full trace. Note that multiplying traffic by 30,000 would have given a more accurate distribution, but resulted in artificially stable patterns.

3.7.1. Profiling More Complex Applications

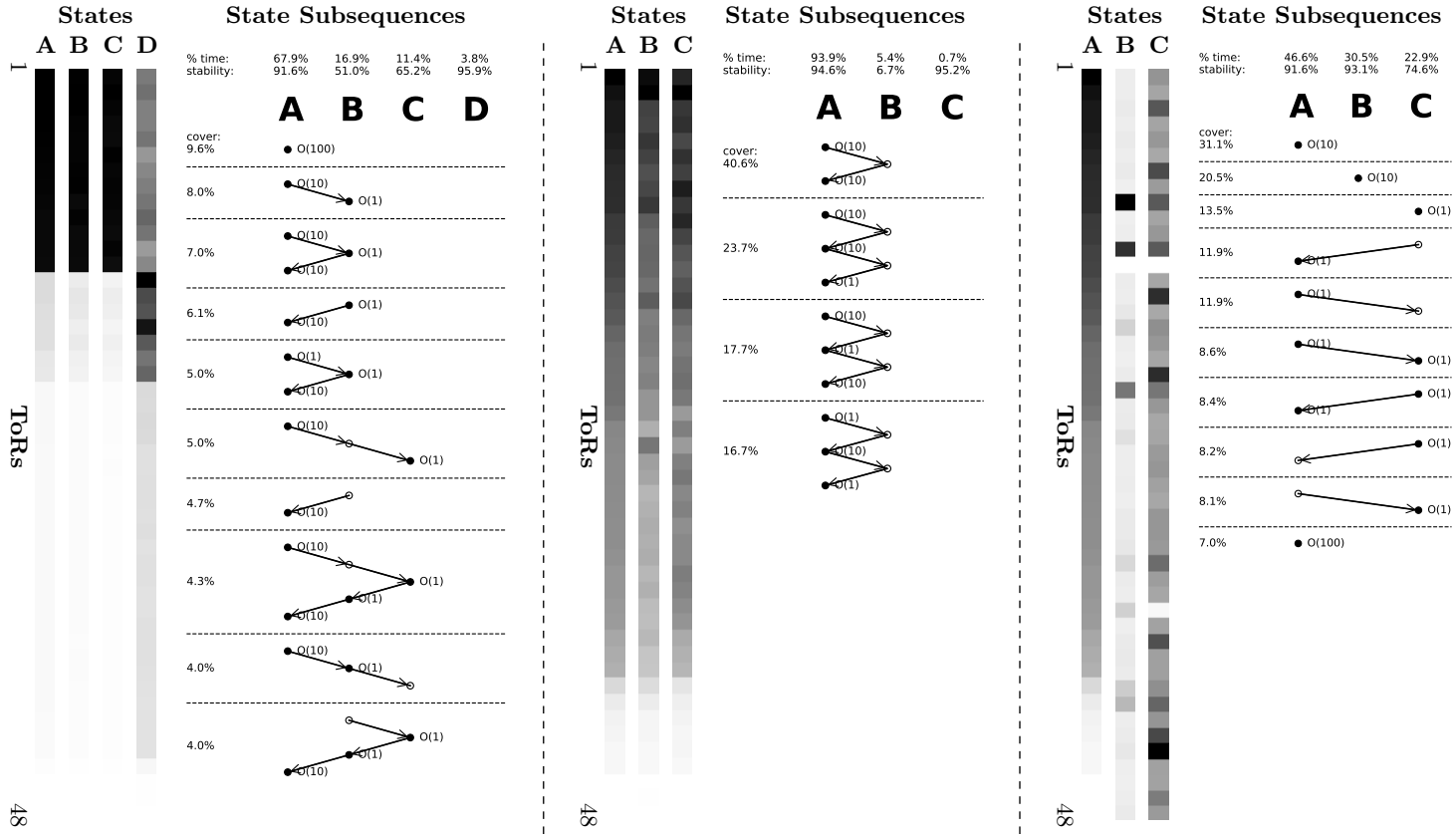
To evaluate how `tpprof`'s algorithms deal with more complex applications, we profile each of the application workloads we introduced earlier in this section. These applications all ran in the presence of background traffic, but we only show profiles of the application-specific traffic.

From the resulting profiles in Figure 3.12, we can see that, for the most part, the network was only lightly utilized during these tests. In Hadoop and Spark, for instance, the network spent $> 96\%$ of the time unutilized, indicating that our particular testbed tends to be CPU-bound. Giraph is the notable exception, spending about equal time utilized and not.

The states reveal some interesting behavior of the applications. For Hadoop and TensorFlow, we see heavy skew in spine utilization, but not to a consistent spine. This likely indicates the presence of a few large flows that dominate the network and sidestep ECMP's flow-level balancing. We also see in these two workloads a slight bias toward the lower-left switch. This is due to task placement: for Hadoop, that switch is home to the controller and name server; for TensorFlow, it holds both the chief worker and the parameter server.

3.7.2. Profiling Large Production Networks

`tpprof` is able to profile more complex networks as well. To demonstrate this, we run `tpprof`'s profiler over large-scale traces of the combined traffic for three production Facebook clusters and show the output in Figure 3.13. We separate the states and subsequences for readability.



(a) Frontend Cluster (b) Database Cluster (c) Hadoop Cluster

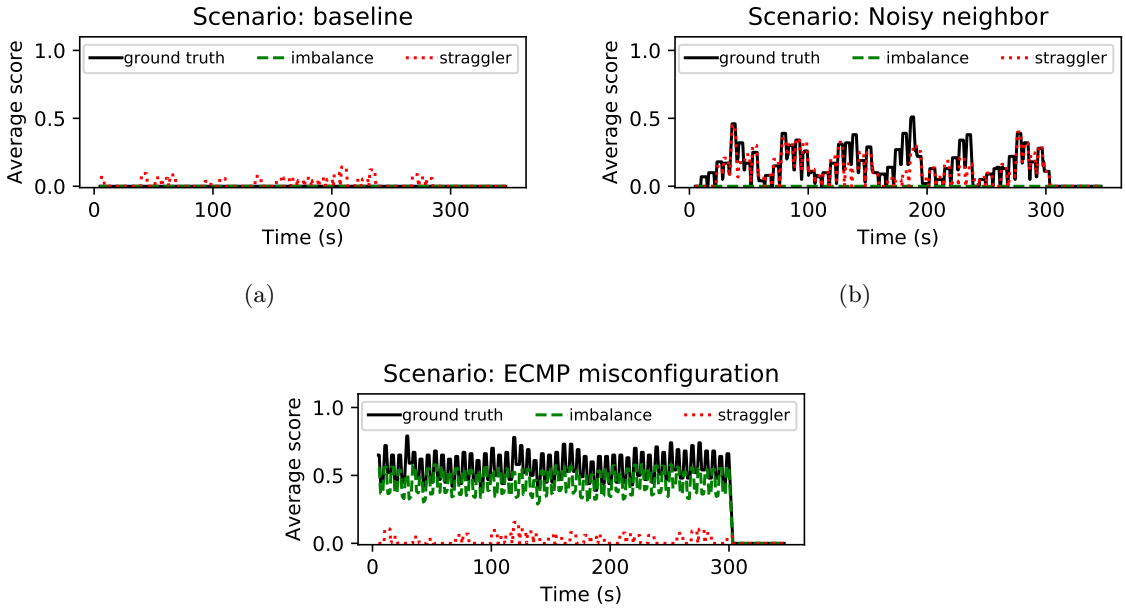
Figure 3.13: `tpprof` profile of three 48-rack Facebook clusters. Figures include both (1) a collection of states (A–D) organized as a 1×48 heatmap, and (2) a list of the most common state subsequences. Letters map between the two representations.

Figure 3.13a shows the profile for the frontend cluster. As in the original paper describing this trace (Figure 5 of [154]), we can observe a clear split between the average utilization of cache, multifeed, and web servers. States A–C show memcache at full utilization, webservers at low utilization, and varying levels of multifeed traffic. Diverging from the original paper, we find an additional network state (occurring 3.8% of the time) in which the multifeed server utilizations spike. The stability of this state indicates that this may manifest as small, but intense and correlated bursts. Subsequences further show frequent transitions between states A and B, with state C representing a short-lived relative lull in multifeed traffic.

Figure 3.13b and Figure 3.13c show the profiles of a database and Hadoop cluster, respectively. Notably, the database cluster is very uniform and stable across the trace, indicating a steady workload and good load balancing properties. The Hadoop profile is also notable in that it diverges substantially from the averaged results in Figure 5 of the original paper, which showed balanced utilization across racks. While the traffic is balanced across longer timescales, our results match more closely with their more granular findings of on-off periods and significant variance at medium timescales.

3.7.3. Efficacy of the TPS Module

We showcase Traffic Pattern Scores by demonstrating how they can help answer an important question: *is my network performing poorly due to load imbalance or stragglers?* For this, we use the straggler and network imbalance signatures from Table 3.1 to diagnose issues in the memcache deployment from Section 3.5.3. We run the deployment in baseline, noisy neighbor, and ECMP misconfiguration scenarios. We then generate labeled network sample traces by manually identifying the precise time windows during which each undesired behavior occurred. Finally, we run `tpprof` on each of the traces and compare signature scores against ground truth scores calculated from sample labels.



(a) (b)

(c)

Figure 3.14: Signature scores for memcache in a baseline configuration, with noisy neighbors, and with an ECMP misconfiguration.

Signature	**Accuracy**	**Precision**	**Recall**
Straggler	0.943	0.867	0.720
Imbalance	0.936	1.000	0.868

Table 3.2: Classification performance of signatures in the memcached testbed.

Figure 3.14 plots the rolling average of ground truth and signature scores in each of the three scenarios. The signature scores are highly correlated with the ground truth. Table 3.2 lists the classification performance. Both signatures have high accuracy and precision, with slightly lower recall—a desirable tradeoff in an alerting system. We note that `tpprof`'s per-scenario precision and recall are 100%: no signature's score is high in the baseline scenario; only the straggler signature's score is high in the noisy neighbor scenario; and only the imbalance signature's score is high in the ECMP misconfiguration scenario.

Filter count	0	128	256	512	1024
CPU Util (%)	0.44	0.65	0.84	1.18	1.77

Table 3.3: `tpprof`'s `iptables` CPU utilization.

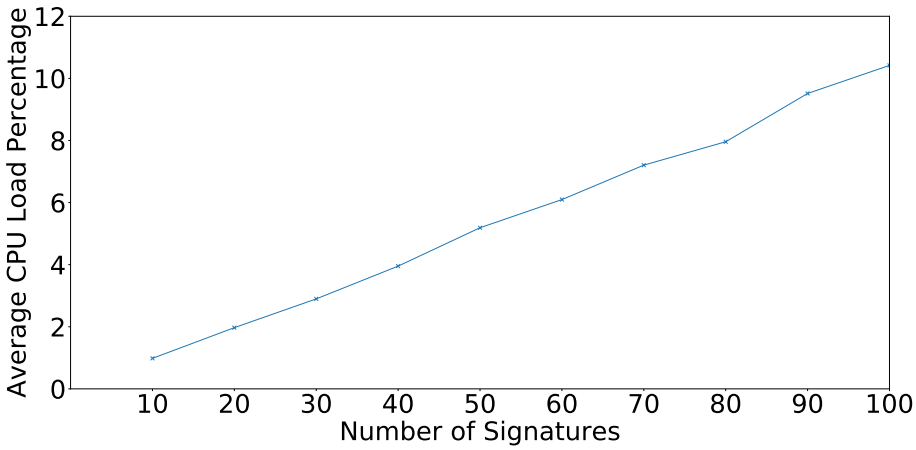


Figure 3.15: Signature vs CPU Load

3.7.4. Overhead and Performance of `tpprof`

Finally, `tpprof` is designed for efficiency and minimal overhead. Only two components in the sampling framework can potentially impact traffic: sample collection and `iptables` tagging.

Analytically, snapshots of all ports on a 128 port switch at a 50 ms interval generate only 0.1 Mb/s of measurement data. As Table 3.3 shows, the `iptables` rules used to construct application-specific profiles also have low overhead.

In addition to measuring overhead, we also benchmark the Hyperscan [90]-based TPS scoring engine, which operates online in parallel with the network. Specifically, we measure average CPU load while operating on the Facebook trace. Figure 3.15 shows single-core CPU load. It increases linearly with the number of signatures, but even in this large network with 100 signatures and a 50 ms sampling frequency, average load for real-time processing is only around 10%.

3.8. Discussion

Other metrics. While we focus on utilization in this chapter, we note that `tpprof` easily extends to any metric collectible from the network. These include simple extensions like packet counts to more advanced metrics like buffer depth and high-water marks. As these metrics are generally correlated with utilization, we anticipate that `tpprof`'s techniques will extend intrinsically, but we leave an exploration of these extensions to future work.

Canned reactions. We also note that the ability of `tpprof`'s scoring engine to distinguish different traffic patterns presents an attractive substrate for building network-level reactions to different traffic patterns. This can also work in reverse: `tpprof` can identify common patterns for which operators should pre-compute reactions. We leave an investigation of this class of applications to future work as well.

3.9. Summary

We present `tpprof`, a network traffic pattern profiler. Just as tools like gprof made it easy for programmers to design, understand, and optimize their programs, `tpprof` does the same for profiling the utilization of large networks. `tpprof` leverages recent advancements in programmable networks and network-wide measurement to capture packet-accurate snapshots of utilization over time. On top of that, `tpprof` builds user-centric profiling, visualization, and automation tools. `tpprof` is agnostic to the application set running over the network and can profile networks in situ, making it an ideal fit for multi-tenant or transit networks. We profile several classic applications in order to demonstrate its utility.

CHAPTER 4

Aragog: Scalable Runtime Verification of Shardable Networked Systems

4.1. Introduction

An emerging bottleneck to correctness and availability in modern cloud systems are the various network functions (e.g., firewalls, NATs, and load balancers) that interpose on the majority of application requests flowing to, from, and between servers in the cloud. Over time, these network functions (NFs) have become increasingly complex. Today, many of these functions are full-fledged distributed systems whose correctness depends on the coordination of multiple devices as well as on stored state and system timing.

Configuration errors and software bugs in these components can have an outsized impact on SLAs [11] not only because of the complexity of these systems, but also because they are on the critical path of most application requests. For instance, a production NAT gateway we verify in this work manages (replicated) states for millions of flows and errors in this system can lead to black holes, broken connectivity, forwarding loops, and more. Public incident reports from providers show multiple outages due to errors like these [74, 11].

To improve availability, recent proposals suggest using *static verification* to prove the correctness of these systems [192, 188, 125, 103, 178, 144, 185, 81]. While powerful, the need to reason about every possible interleaving of inputs and control flows presents a significant obstacle to the application of these techniques in today's network functions.

Attempting to explore the full space of control flow paths often leads to state/path explosion [125, 103, 178]. Mitigations to this problem, broadly speaking, can be categorized in a few ways. The first is to require the use of special programming languages or other types of programmer interaction [189, 81]. The second is to use model checking techniques to more efficiently explore all possible system behaviors. Finally, many systems—to reduce the state space they must verify and to make verification more tractable—limit the set of verifiable behaviors, e.g., to those that are unordered [144], abstract [22], or restricted to a single machine [192, 188].

While effective in many cases, each of these approaches also comes with significant drawbacks. With the first, programmers are saddled with a substantial burden that can overwhelm the development of the system. With the second, model checking still typically relies on hand-written models of functionality, which may be difficult to provide for a rapidly evolving or complex system. Finally, limiting the scope of verification fails to extend to the increasingly complex services found in modern networks—services that arguably need verification the most.

An alternative approach to static verification is runtime verification of distributed systems. In runtime verification, a tool extracts information about the current state of a running system (testbed, canary, or production) to verify that invariants hold throughout execution [136, 54, 151, 126, 58, 127, 122, 177]. Compared to static verification, runtime verifiers only test inputs and control flows that are seen in practice, thus improving scalability and enabling verification of actual deployments running over actual data. In return, they sacrifice a principled exploration of the system's behavior and the ability to catch bugs early. We argue that these tradeoffs are a better fit for our operators' requirements.

We find today's runtime verifiers cannot be applied as-is to deployed network functions. The challenge (for network functions) is the need, at runtime, to: (1) reason

about the coordination between events issued at different locations, (2) efficiently aggregate global state after each event, and (3) scale sub-linearly with the size of the original system—after all, a verifier that requires the same amount of resources as the system itself is untenable for most production environments.

In this dissertation, we present the design of a scale-out, runtime verification tool for network functions called *Aragog* that overcomes the above challenges. *Aragog* provides a simple, but expressive language for describing violations of invariants, with a focus on supporting network functions. Examples of network-centric language features that are found in *Aragog*'s Invariant Violation (IV) specifications, but that are uncommon in other runtime verifiers are support for properties that are parametric over the "location" of events, properties that reference stateful variables, the ability to execute partial matches over packet fields, and support for temporal predicates.

Aragog translates these IV specifications to a set of symbolic automata that can efficiently verify the current global state of the system. In addition, to ensure that the system can scale out to a near-unlimited number of machines, *Aragog* implements the core of these checks on top of production stream processing systems [9, 10]. To efficiently coordinate between distributed verifiers, *Aragog* relies on hardware-supported time synchronization protocols like PTP. Finally, to minimize the overhead of the verification system, *Aragog* leverages observations that network events/invariants are typically:

Flow- or connection-based: For most network functions, correctness is defined on a per-flow or per-connection basis. From the IV specification, *Aragog* derives sharding keys that allow it to distribute the verification task across independent workers. These shards also expose boundaries on which we can gracefully scale down to a sampled subset of the input.

Partially suppressible Rather than aggregate all events in the system to a logically

centralized verifier, most network events have limited windows of relevance depending on the state of the system, e.g., only if the connection has recently been closed. *Aragog* includes an optimization scheme to suppress such messages before they ever leave the NF instance.

Aragog does not guarantee perfect accuracy under asynchrony—to do so would require atomicity guarantees in the critical path of the network functions. *Aragog* instead handles these situations speculatively and notifies users after-the-fact[4] about transient inconsistency (§4.7.3). Despite this, *Aragog* identified at least four bugs in an early (limited) deployment of a real distributed network function: Azure's new NAT gateway (NATGW). These bugs were detected within ∼100 ms of occurrence. Compare this to the hours our operators typically spend searching for similar bugs.

To summarize, our work makes the following contributions:

- We present a case study of the needs of a large modern network function from Microsoft's Azure. The system exhibits several interesting characteristics and suggests key requirements for verifier design.

- We synthesize ideas from timed regular expressions, symbolic automata, and parametric verification. To the best of our knowledge, ours is the first to demonstrate a concrete need and method for combining these concepts.

- We introduce the design and implementation of *Aragog*, a system for at-scale runtime verification. When needed, *Aragog* can also run on traces (offline) and therefore complement static verification to find implementation bugs in distributed networked systems. Among other innovations, *Aragog* includes a novel method for computing location-dependent suppression of network events.

[4]This reporting happens in under 1 s. This delay is on the same order as other alerting systems used in our production networks.

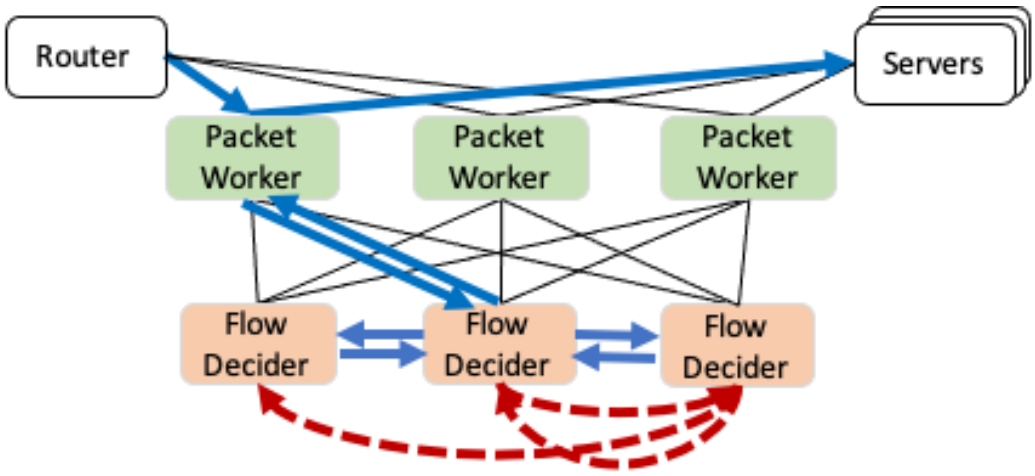


Figure 4.1: The architecture of our NATGW. The bolded blue arrows show the sequence of communication to handle the SYN packet of an incoming flow: it is sent to a random packet worker, which forwards it to the flow decider in charge of that flow. The flow decider chooses a target server and replicates the mapping to other deciders, then installs it in the original packet worker. The three dashed red arrows trace the allocation of the mapping for the reverse flow.

- We introduce a collection of *Aragog* invariant violations for a set of distributed network functions, and we evaluate *Aragog* on NATGW and a distributed firewall.

4.2. Motivation: A Cloud-scale NAT Gateway

Our work is grounded in experience with Azure's large-scale NF that we call NATGW. NATGW is a cloud-scale NAT gateway that balances incoming requests over available servers and supports almost all external traffic.

Like many other NFs of similar scale [145, 62], NATGW is implemented entirely in software, is distributed across a pool of servers, and replicates state for fault tolerance. Routers use ECMP-based anycast to randomly direct packets to NATGW workers, which then rewrite the destination IP and port to point at a target server. A similar translation occurs for packets in the reverse direction (from the server to the client).

Figure 4.1 depicts the NATGW architecture. It is composed of two types of nodes:

packet workers and flow deciders. Packet workers process every packet passing through the NATGW, parsing its header, looking up the target server, and rewriting the packet header to point to that target. The mapping of a flow to a target server is decided with the help of a sharded set of flow deciders. The deciders cache and replicate these mappings to other deciders to ensure availability.

Flow allocation When a packet worker receives the first packet of a new flow, it uses a hash of the 5-tuple to identify the "primary" flow decider that owns the flow and forwards the packet to that decider. The primary then:

1. Decides the target server to which to send the new flow and installs the mapping in the local flow cache.

2. Sends the reverse mapping to the flow decider that "owns" the other end of the flow. Together, these two mappings cover translation for both incoming and outgoing traffic.

3. With its counterpart primary, greedily copies the mappings to the cache of other flow deciders in a manner akin to chain replication: decider i will try to copy to deciders $(i+1) \bmod N$ and $(i+2) \bmod N$, where N is the number of deciders. If one is down, it switches to $(i+3) \bmod N$.

4. Installs the mapping into the originating packet worker.

After the above flow allocation, the packet worker can process all subsequent packets of the flow without coordination with any other node. If the packet worker fails, anycast redirects the packet to another worker; the new worker will send the packet to the primary flow decider, fetching the existing mapping. If the flow decider fails, packet workers will query the next deciders in the sequence until they find the mapping.

Flow mapping timeouts All components time out their flow mappings to ensure

stale entries are eventually removed.

To ensure NATGW maintains mappings for active flows, packet workers periodically send a keepalive message to the primary decider. The primary forwards the keepalive to all replicas, refreshing the timeout on every instance of the mapping in the system. In parallel, the primary forwards the keepalive to the primary in charge of the reverse mapping.

Eventual consistency This NATGW design exhibits some interesting properties. One of them is a choice to allow for temporary inconsistency in the presence of node failures in order to satisfy certain practical and performance constraints.

For example, consider three replicas of a flow mapping R_P, R_{P+1}, and R_{P+2}, where R_P is the primary. To delete the mapping, R_P would send a delete request to both of the other nodes. Now imagine the message to R_{P+1} is dropped. Rather than waiting for R_{P+1}, the others will go ahead and delete f. If, later, R_P fails, packet workers will contact R_{P+1} for the mapping, which will return a stale/inconsistent result until a timeout or periodic sync eliminates the inconsistency.

There are known mitigations to the above behavior (e.g., querying a quorum on every packet or initiating a view change algorithm on R_P's failure); however, these come with significant performance costs. Instead, the NATGW is an example of a *deployed* architecture that chooses eventual consistency after careful consideration of its drawbacks and alternative solutions. Our work is motivated by our operators' experience with such behaviors.

4.3. Design Goals

Our runtime verifier targets the following design goals:

Practicality Network functions are complex; written in a variety of languages; and

frequently rely on external libraries, drivers, and other components. NATGW, for example, is built using libraries like DPDK and interacts with an ecosystem of networking hardware and configurations. The intricacies of the systems, the richness of their dependencies, and the rapid evolution of all the associated components mean the system is not easily modeled or accurately simplified. Instead, verification should be of the end-to-end system, *in situ*.

In the same vein, *Aragog* should not place undue burden on developers, e.g., by requiring engineers to perform non-trivial proof writing (as mandated by many deductive reasoning techniques). NATGW has over 40 thousand lines of code—*Aragog* should avoid incurring a proportional overhead.

Expressiveness Prior work has observed a gap between state-of-the-art verification tools and the requirements of modern networks [136]. In particular, it is challenging to specify invariants related to: (1) parametric variables over values like locations or identifiers, (2) coordination between network devices, and (3) timing of events. Moreover, since the number of devices (e.g., flow deciders) may vary over time as the system scales out, it is useful to express properties in a way that does not require explicitly naming components. *Aragog* should provide syntax and semantic support for these behaviors.

Scalability Just as a single machine cannot handle all traffic entering a large network, it also cannot be expected to verify the correctness of the entire network. Rather, the verifier should scale out to arbitrary size and require fewer resources than the original system. Therefore, *Aragog* should attempt to minimize the number of messages exported from each NF, e.g., by exporting events (resulting from the execution of the NF) rather than packets (the inputs to the NF).

Graceful degradation of accuracy As we describe in Section 4.7.3, perfect preci-

sion and recall is impossible in an asynchronous system without substantial overhead. Instead, *Aragog*'s correctness goal is in the same spirit as NATGW's: perfect recall under the assumption of 'partial synchrony' [61] and notifications of potential false positives/negatives after-the-fact. Our operators find this is sufficient for most cases.

Near-real-time alerts Diagnosing bugs manually can take hours of operator time and the network could worsen the longer the bug persists: *Aragog* should raise alerts within seconds of observing the offending sequences of events.

4.4. *Aragog*'s Architecture

We present the design and implementation of a practical, expressive, and scalable verifier for large and complex NF deployments. Our system, *Aragog*, is a combination of a language for specifying invariant violations and a scale-out runtime system. *Aragog* takes a grey-box approach, requiring small changes to the underlying source code in order to export events of interest to the verifier. Thus, *Aragog* verifies by:

Specifying invariant violations over user-defined events To provide operators with sufficient expressiveness to check network-level events, *Aragog* comes equipped with a new language for specifying invariant violations that is based on writing symbolic regular expressions over a global trace of events (and their locations) in the system. *Aragog*'s language includes a notion of parameterized "variables" that allows violations to be described in a way that holds for any combination of variable instantiations subject to constraints.

Checking for invariant violations NF developers export any relevant events to *Aragog*. To scale up checking of the event stream, *Aragog* does two things. The first is to automatically analyze and split verification into local and global components. The local level resides at the NF instances themselves, where *Aragog* infers (only using the state of the local instance) whether it can safely suppress the event before exporting

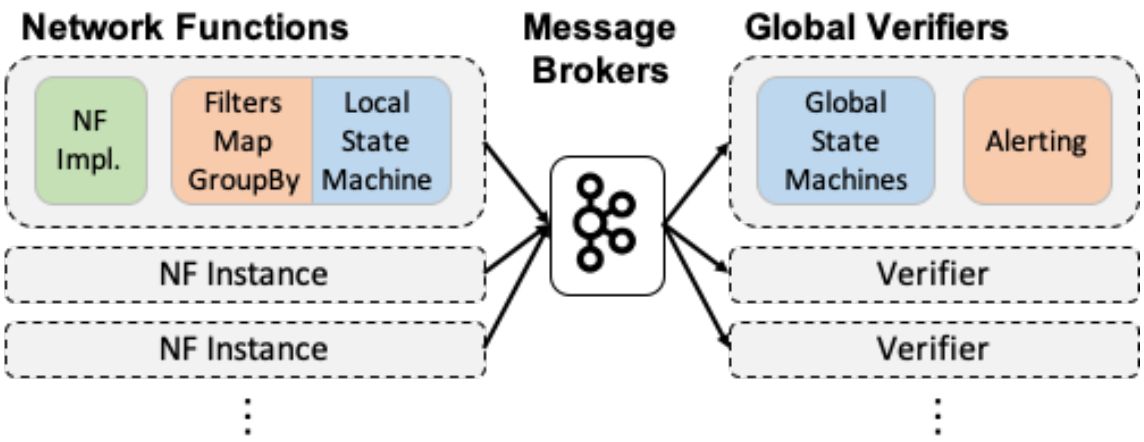


Figure 4.2: The architecture of *Aragog*. NF instances generate and feed events into a set of local state machines. The NF instances use these state machines to determine if they can hide unnecessary messages before exporting the rest to the global verifier. These messages pass through a Kafka cluster and are streamed to a set of Flink-based verification engines.

it to the global *Aragog* verifier. The second is to leverage the fact that most network invariants are defined across *related flows* rather than globally—for instance, on the granularity of a 5-tuple. As a result, events can be automatically sharded across a cluster of scale-out stream processing workers using Kafka [105] and Flink [44].

Note that, because the invariants are defined and checked only across related flows, we only need to know the correct ordering for events pertaining to those flows: event timestamps that use the sub-microsecond-scale synchronization of PTP suffices for our needs. For many production networks, these types of event exports are already common.

Overview Figure 4.2 shows *Aragog*'s design. Users describe a set of invariant violations that identify classes of incorrect behavior. *Aragog* translates these to a set of symbolic automata and then splits the automata into local and global components. It then deploys these to NF instances and global verifiers.

At runtime, NF instances stream events into the pipeline. The local *Aragog* agent

```
{ "fields" : [
    {"eventType" : 16},
    {"nodeType" : 8},
    {"sourceIPv4or6" : 8},
    {"sourceIPv4or6==4" : [ {"srcIP" : 32} ],
     "sourceIPv4or6==6" : [ {"srcIP" : 128} ]},
    ...
],
"constants" : {
    "NAT_ALLOCATION" : 1,  // eventTypes
    "FLOWCACHE_CONSENSUS" : 769,
    "PACKET_WORKER" : 0,   // nodeTypes
    ...
}}
```

Figure 4.3: A snippet of the NATGW JSON event schema.

filters, maps, and shards events The message brokers aggregate and compact those streams The global verifiers determine, for the shard, whether a violation occurred. Kafka and Flink will automatically allocate resources and load balance requests to ensure scalability.

4.5. Specification Language

Users define both events and policies over the events using two types of specifications that are inputs to *Aragog*: event definitions and Invariant Violation (IV) specifications. While both of these require the user to have some knowledge of the inner workings of the NF to specify how it can fail, our network operators determined that event-based violations struck a reasonable balance between precision and ease-of-use.

4.5.1. Event Definitions

Users specify the format of the event messages that arrive at the local verifier. *Aragog* expects these messages to be in the form of packed arrays of raw binary data whose format is defined with a JSON configuration file. For example, Figure 4.3 shows a selected subset of the definition for NATGW event messages. 'fields' contains the ordered list of expected fields in the message. Each field is defined by a JSON dictionary specifying the field's name and its length in bits—for instance, the first 16 bits

```
FILTER((eventType == FLOWCACHE_PRIMARY_ADD
      || eventType == FLOWCACHE_REMOVE_ENTRY)
      && workerType == FD)
GROUPBY(srcIP, dstIP, srcPort, dstPort, proto)
MATCH
(eventType == FLOWCACHE_PRIMARY_ADD) @ $X
((eventType == FLOWCACHE_REMOVE_ENTRY) @ NOT $X)*
(eventType == FLOWCACHE_PRIMARY_ADD) @ NOT $X
```

Figure 4.4: An example IV specification that ensures at most one primary is ever active for a given flow.

of the event message is an `eventType`.

Conditionals In addition to specifying the length of each field and their ordering, *Aragog* allows users to implement simple conditional parsing logic. The example event definition shows one such use where *srcIP* can be either IPv4 or IPv6. In the configuration shown, event messages include a 8-bit field that specifies the IP version number. Depending on the value of that version number, the next field is either a 32-bit or 128-bit `srcIP` field. These branches can define entire sub-headers and can contain nested conditionals.

Named constants *Aragog* also allows users to define named constants representing integer values represented in decimal, hexadecimal, or binary notation. We show four such constants in Figure 4.3: two for values of the `eventType` field and one for the `nodeType` field. These are intended for use in IV specifications to make them more readable.

4.5.2. Invariant-Violation (IV) Specifications

Aragog parses incoming event messages and checks them against a set of user-defined policies that describe sequences of events that violate the invariants of the system. Operators specify these policies using *Aragog*'s domain-specific language, which we detail in this subsection.

Figure 4.4 shows an example specification for our NATGW. The policy only pertains to a subset of events (lines 1–3), and *Aragog* verifies it on a per-5-tuple basis (line 4). A violation occurs when some node `$X` adds a primary mapping (line 6) and then a different node (`NOT $X`) adds the same mapping (line 8) without `$X` removing it. The full grammar for IV specifications is shown in Figure 4.5. Briefly, an IV specification consists of (1) a collection of event transformations followed by (2) a regex-like expression over the generated events.

4.5.3. Transformations

Aragog allows users to define a set of policy-specific transformations. In addition to enabling greater flexibility and expressiveness, *Aragog* also uses these transformations to perform an initial filtering and aggregation as well as to identify valid sharding strategies. *Aragog* currently supports three transformations: `GROUPBY`, `FILTER`, and `MAP`.

Operators can use `GROUPBY` to indicate which events need to be considered together and which can be considered separately. For example, when an operator wishes to guarantee at most one primary is active (Figure 4.4) for *each flow*, the `GROUPBY` is used to classify events into unique flows. *Aragog* uses this transformation to both simplify policy logic and to assist in the sharding of verification.

Operators can also use the `FILTER` transformation to indicate which events should be considered at all and which should be ignored. In the above example, we only care about flow deciders—specifically when they add a flow as a primary and when they delete the flow mapping from the cache; we can filter events of any other type or from any other type of node. `FILTER`s are critical for reducing the number of events handled by the verification framework.

Finally, operators can use the `MAP` transformation to generate entirely new fields based on mathematical expressions over existing fields of the event message.

⟨*IVspec*⟩ ::= ⟨*transformations*⟩ '`MATCH`' ⟨*events*⟩

⟨*transformations*⟩ ::= ⟨*transformations*⟩ ⟨*transformations*⟩
| '`GROUPBY`' '`(`' ⟨*fields*⟩ '`)`'
| '`FILTER`' '`(`' ⟨*filter_matches*⟩ '`)`'
| '`MAP`' '`(`' ⟨*field_expression*⟩ '`,`' ⟨*field_name*⟩ '`)`'

⟨*fields*⟩ ::= ⟨*field_name*⟩ ['`,`' ⟨*fields*⟩]
| '`LOCATION`' ['`,`' ⟨*fields*⟩]

⟨*filter_matches*⟩ ::= '`(`' ⟨*filter_matches*⟩ '`)`'
| ⟨*filter_matches*⟩ '`||`' ⟨*filter_matches*⟩
| ⟨*filter_matches*⟩ '`&&`' ⟨*filter_matches*⟩
| ⟨*filter_match*⟩

⟨*filter_match*⟩ ::= ⟨*field_name*⟩ ⟨*compare_op*⟩ ⟨*field_name*⟩
| ⟨*field_name*⟩ ⟨*compare_op*⟩ ⟨*value*⟩

⟨*events*⟩ ::= '`.`' '`@`' ⟨*location_spec*⟩
| ['`!`'] '`(`' ⟨*event_match*⟩ '`)`' '`@`' ⟨*location_spec*⟩
| '`(`' ⟨*events*⟩ '`)`'
| ⟨*events*⟩ ⟨*events*⟩
| ⟨*events*⟩ ⟨*regex_op*⟩
| '`SHUFFLE`' '`(`' ⟨*events_list*⟩ '`)`'
| '`CHOICE`' '`(`' ⟨*events_list*⟩ '`)`'

⟨*events_list*⟩ ::= ⟨*events*⟩ ['`,`' ⟨*events_list*⟩]

⟨*location_spec*⟩ ::= '`ANY`'
| ⟨*loc_matches*⟩

⟨*loc_matches*⟩ ::= ['`NOT`'] '`$`'⟨*loc_name*⟩ ['`,`' ⟨*loc_matches*⟩]

⟨*event_match*⟩ ::= ⟨*field_match*⟩ ['`,`' ⟨*event_match*⟩]

⟨*field_match*⟩ ::= ⟨*terminal*⟩ ⟨*compare_op*⟩ ⟨*terminal*⟩

⟨*terminal*⟩ ::= ⟨*field_name*⟩
| ⟨*value*⟩
| '`$`'⟨*variable_name*⟩
| '`TIME`'

Figure 4.5: Grammar for *Aragog*'s IV specification language. Tokens ending in '_name' are identifiers that must begin with a letter; the 'compare_op' token refers to the class of operators '==', '!=', '<', etc; 'value' indicates a constant number; and 'field_expression' is a mathematical expression over fields.

4.5.4. Event Expressions

Users define invariant violations over the transformed event streams by specifying sequences of events that result in a violation of a particular policy. Users specify these sequences with a regular-expression-like language, which describes patterns over pre-

defined elements. In *Aragog*'s case, the elements take the form of a set of matching operations over the fields of the event message; the example in Figure 4.4 shows matches on one such field, the `eventType`. A match can occur at any point in the stream of events and triggers on every occurrence of the match, not just the first. For example, if events $A \rightarrow B \rightarrow A$ form a violation and (at runtime) we observe the sequence $CABABAC$, *Aragog* will alert twice.

As in other regular languages, users can list the sequence of expected elements and use operators like '`*`', '`+`', and '`?`' to signify repetitions. Users can also leverage the functions `CHOICE` and `SHUFFLE`. In `CHOICE`, an occurrence of any one of the contained expressions matches. In `SHUFFLE`, the contained events can arrive in any order, but must all arrive.

Event expressions come after the set of transformations and must appear after a `MATCH` statement.

Locations In distributed NFs, an important feature is that correct behavior is defined not only on the events and their order, but on *where* the events occurred. Therefore, every event match is accompanied by a location specifiers. This is useful for specifying matches, but it is also important for determining how we might partition evaluation of the IV specification across both local and global verifiers (see Section 4.6). In both cases, the goal is to determine whether each pair of events are expected to occur at the same or at different NF instances.

Consider again the example in Figure 4.4. The example contains a single named location, `$X`, corresponding to the original primary node for the current flow. One way to use this named location is to specify that another event in the sequence must *also* occur at `$X`. Another, demonstrated in lines 7&8, is to specify that the event occurs at a location distinct from `$X`. Note that the syntax does not constrain the

```
1 FILTER(eventType == INIT || eventType == DROP)
2 GROUPBY(LOCATION)
3 MATCH
4  (eventType == INIT, srcIp == $S, dstIp == $D,    srcPort == $P, dstPort == $Q) @ ANY
5  (. @ ANY)*
6  (eventType == DROP, srcIp == $D, dstIp == $S,    srcPort == $Q, dstPort == $P) @ ANY
```

Figure 4.6: An example specification that checks that a stateful firewall does not drop reverse traffic for an open connection.

relationship between the locations of the events of lines 7&8.

Every event can reference one or more named locations, or it alternatively use the location `ANY`, which indicates no special semantic meaning of the location of the event. In the case of multiple locations, users specify multiple predicates (one per location). For example, to ensure three events with distinct locations: one could specify ev_1 at (`$X`, `NOT $Y`); ev_2 at (`NOT $X`, `$Y`); and ev_3 at (`NOT $X`, `NOT $Y`).

One possible method of implementing locations is to enumerate all possible locations in the system and expand the event expression accordingly. While this would allow the usage of more traditional state-machine evaluation techniques, it would also lead to an unacceptably inefficient implementation. Further, any change in membership would require us to fully recompile and re-install all IV specifications across the system. Instead, *Aragog* lazily tracks all potential candidates for location variables at runtime using a multi-leveled tree data structure, which we describe in detail in Section 4.6.

Variables *Aragog* generalizes the state tracking afforded to locations in order to track other types of state in the IV specification. Examples of non-location stateful properties include the IP/port NAT mappings of the NATGW and connection tracking in a firewall. An example of the latter is shown in Figure 4.6, which verifies that if an outbound flow from source IP `$S` and destination IP `$D` is properly initialized, then packets in the reverse direction are also allowed.

```
MAP(srcIP < dstIP ? srcIP : dstIP, IP1)
MAP(srcIP < dstIP ? dstIP : srcIP, IP2)
MAP(srcIP < dstIP ? srcPort : dstPort, port1)
MAP(srcIP < dstIP ? dstPort : srcPort, port2)
FILTER(flag == FIN || flag == ACK || flag == FIN_ACK)
GROUPBY(IP1, IP2, port1, port2)
MATCH
  (flag == FIN) @ $X
  SHUFFLE(
    (flag == FIN, TIME == $s) @ $Y,
    (flag == ACK, TIME == $t) @ $Y)
  (flag == SYN, TIME - min($s, $t) <= 30000) @ $X
```

Figure 4.7: An example of a timing violation specification that checks the behavior of TCP's TIME-WAIT state [89]. The SYN *must not* arrive by a deadline. This specification assumes that only packet sends are captured.

```
FILTER(flag == FIN || flag == FIN_ACK)
GROUPBY(IP1, IP2, port1, port2)
(eventType == FIN, TIME == $t) @ ANY
((eventType != FIN_ACK, TIME - $t <= 30000) @ ANY)*
(TIME - $t > 30000) @ ANY
```

Figure 4.8: An example of a timing-related IV specification that checks timely arrival of a FIN_ACK after a FIN. The FIN_ACK *must* arrive by a deadline.

As these variables do not indicate or impose restrictions on the location of the event, we do not use them for the partitioning procedure of Section 4.6.

Timing Timeouts and deadlines are also common in NFs. To specify them, users can use parameterized variables in conjunction with a builtin `TIME` field to compare the time between multiple events. For example, Figure 4.7 defines a violation of the TIME-WAIT semantics of a TCP flow in which SYN packets should not be sent within 30 s of a passive closer's FIN/ACK. The same SYN packet 31 s after the FIN/ACK would not be a violation. On the other end of the spectrum, Figure 4.8 defines a violation where a FIN-ACK does not arrive in time (within 30 s of the FIN). Any intervening FIN-ACK will mean that the violation does not match.

4.6. State Machine Generation

Aragog checks for invariant violations efficiently by translating each of the IV specifications into a state machine. In contrast to traditional finite-state automata, *Aragog* requires a combination of complex features, e.g., timing, arithmetic, field/location variables, and regular expression-event patterns.

Aragog, thus, generates its state machines in three stages. First, it creates a symbolic non-deterministic finite automaton (SFA) [52] whose alphabet is based around a theory of arithmetic and boolean algebra, and whose predicates can include the placeholder variables described in the previous section. Second, it determinizes the SFA to a deterministic symbolic finite automaton (DSFA) to reduce runtime overhead of state machine execution. Finally, it constructs localized versions of the DSFA that can be used to infer the global state of the system from only locally observed events.

4.6.1. Constructing the SFA

We first convert all predicates on events into boolean logic with equalities/inequalities by taking the conjunction of all event field matches and the location specifier. For example, we transform an event match `(A==B, C==D) @ NOT $X` to the predicate `(A==B` $\wedge$ `C==D` $\wedge$ ρ`!=$X)`, where ρ is the placeholder for the event's location, which we determinize at runtime. A '`!`' modifier on the event would negate this predicate.

Aragog performs an additional check on the sequence of generated predicates to facilitate efficient variable checking (Section 4.7.2). Specifically, it checks via reachability analysis that all uses of variables in either an arithmetic expression or non-equality comparison ($<$, $\leq$, $>$, and $\geq$) strictly follow after their introduction via an equality comparison.

With the resulting predicates, *Aragog* constructs the SFA by creating a start state, S, with a self-loop for any event (`TRUE`). This self-loop ensures the pattern will match

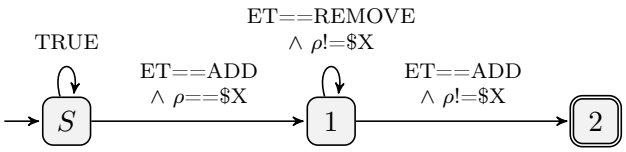


Figure 4.9: SFA for Figure 4.4 with some field names and constants abbreviated as well. ρ indicates location.

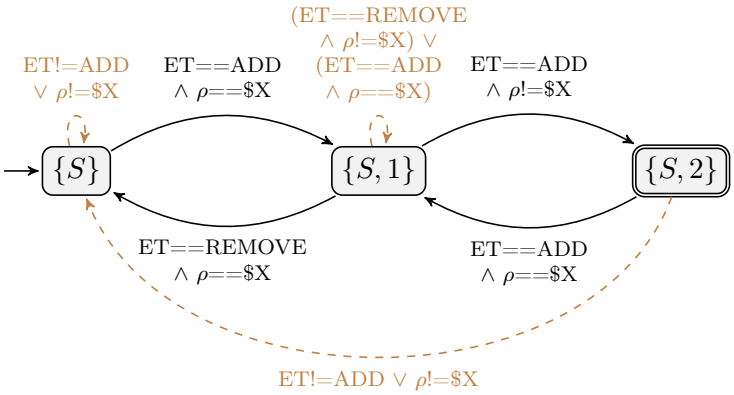


Figure 4.10: DSFA for the SFA in Figure 4.4. Colored, dashed edges represent suppressible transitions.

starting from anywhere in the event trace. From the initial state S, *Aragog* recursively builds out the state machine using Thompson's construction [176], treating `CHOICE` as a choice operator, and expanding `SHUFFLE` to all permutations. Figure 4.9 shows a (minimized) SFA for the example violation specification from Figure 4.4. We mark the final state in the SFA as the accepting state, which indicates a violation when reached.

The specified transitions may not cover the complete space of possible events. All events that do not match any transition out of the current state will never lead to a match.

Aragog next determinizes the SFA: it generates an efficiently executable DSFA from the SFA using standard symbolic automata techniques [52]. The result is a state

machine where all transitions are unambiguous and exhaustive. Figure 4.10 shows the DSFA for the example. Each state in the DSFA stores the correesponding set of SFA states the machine is in at that given point in time.

4.6.2. Local State Machines

Conceptually, the DSFA provides an efficient method for checking whether a stream of events leads to an invariant violation. In principle, we could simply funnel all events to a central verifier, which would then apply the relevant DSFA transition and report a violations upon reaching an accepting state. Unfortunately, doing so would require the verifier to process *all* unfiltered events in the system. Instead, we further improve *Aragog*'s scalability by generating a localized version of the state machine that is executed on the same machine as the NF before sending the event to the global verifier.

4.6.3. Suppressible Transitions

The local state machine needs to identify events that will not impact the detection (or lack of detection) of a user-specified violation whether or not it is sent to the global verifier. Our key observation is that there are transitions in the global DSFA that do not affect the end result of the state machine. We term these transitions *suppressible transitions*. More formally:

Definition 1. An event stream s is either empty $s = \epsilon$ or it consists of an event followed by another stream $s = e \cdot s'$.

Definition 2. $q \xrightarrow{e} q'$ indicates that, from state q, event e transitions to state q'. We lift this to event streams inductively as $q \xrightarrow{\epsilon} q$, and $q \xrightarrow{e \cdot s} q''$ iff $q \xrightarrow{e} q'$ and $q' \xrightarrow{s} q''$.

Definition 3. Transition t is suppressible if for any event e matching t from state q, then (1) $q \xrightarrow{e} q'$ means q' is not an accepting state, and (2) for any event stream s, and accepting state q_a then $q \xrightarrow{e \cdot s} q_a$ iff $q \xrightarrow{s} q_a$.

In the running example DSFA in Figure 4.10, the three dashed transitions are suppressible given the above definition. The two self-loops are clearly suppressible (satisfy Definition 3) since an event processed by such a loop will not change the global state—(not) observing the event has no effect, and the loops do not occur on accepting states. Perhaps less obvious is that the bottom-most edge is also suppressible since, from either state $\{S\}$ or $\{S, 2\}$, one needs to see the same two events to get back to the accepting state $\{S, 2\}$. For example, an `ADD` event at `$X` followed by another at `NOT $X` will take either state $\{S\}$ or $\{S, 2\}$ back to $\{S, 2\}$. We never mark transitions with time constraints as suppressible—we assume the timing of an otherwise irrelevant event might still be significant.

4.6.4. Local State Machine Construction

Aragog uses local knowledge to determine whether an event will be processed by a suppressible transition. Since each local component is unaware of what might be happening at other components, it must conservatively account for all possibilities. To determine (locally) whether an event is suppressible, we create a local state machine for every location variable in every IV specification such that each machine assumes it is playing the role of that location (e.g., one machine for "I might be `$X` in a violation" and another for "I might be `$Y` in a violation"). In the example from Figure 4.10, there is only a single local state machine: the one for `$X`.

The first step in creating a local state machine, L, is to model the uncertainty other locations may introduce (Figure 4.11). The algorithm takes the global state machine G, the location variable V (e.g., `$X`), and a predicate F corresponding to the user-defined `FILTER` statements. It returns a new localized DSFA.

The algorithm considers each transition T in G where T has predicate P, and checks whether the formula $(\mathrm{F} \wedge \mathrm{P}) \not\Rightarrow (\rho = \mathrm{V})$ is satisfiable (line 6). If it is, then there exists a potential event that makes it through the filter F and uses transition T but which

```
input: Global DSFA G, variable V, filter F
output: Local DSFA L
1  Function CreateLocalDFA(G, V, F):
2      L ← CopyStates(G)
3      for S ← States(G) do
4          for T ← Transition(G,S) do
5              P ← Predicate(G, T)
6              if SAT((F ∧ P) ⇏ (ρ = V)) then
7                  AddTransition(L, TargetState(T), ε)
8              P' ← Simplify(P, ρ == V)
9              AddTransition(L, TargetState(T), P')
10     return Determinize(L)
```

Figure 4.11: Create a local state machine for a variable

takes place at a location other than V. To model the fact that other NF instances might send events that use this transition, the algorithm adds to L an epsilon (ϵ) transition (line 7). An ϵ transition is one which the local SFA can take immediately and unconditionally. It accounts for the possibility of concurrent execution of other NF instances to represent that the global state could be in either state (the one before or the one after the ϵ transition).

In either case, the algorithm then adds a local transition to L by simplifying the existing transition predicate (P) to account for the fact that the location is known (line 9). It does so by partially evaluating the predicate with the assumption that ρ==V (line 8). In Figure 4.10, for example, the transition (`ET==REMOVE` $\wedge$ ρ `== $X`) is simplified to `ET==REMOVE`.

Figure 4.12 shows the local SFA for location `$X` and its determinized (DSFA) form. By executing the DSFA in Figure 4.12 locally, an NF instance can learn some partial information about the state of the overall system. For example, after seeing an `ADD` event, the NF instance recognizes that (if it is `$X`) the global state machine can be in any state: $\{S\}$, $\{S, 1\}$, or $\{S, 2\}$. However, after locally processing a `REMOVE` event, the local machine now knows it must be in state $\{S\}$ once more.

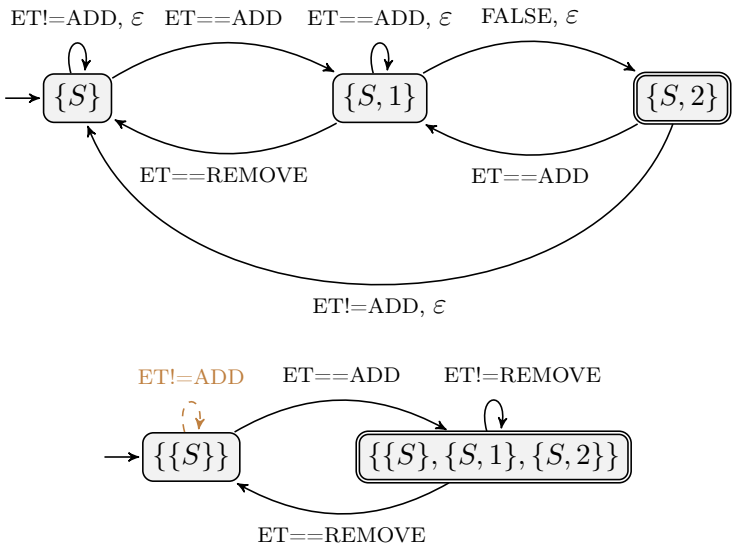


Figure 4.12: Local machine for `$X` from Figure 4.10. SFA is shown on top and its equivalent DSFA is shown below. Colored, dashed edges indicate locally suppressible transitions.

input: Global DSFA G, filter F
output: Local state $\Theta = \langle \{L_1, \ldots, L_k\}, NC \rangle$

```
1  Function Localize(G, F):
2      NC ← false, LS ← ∅
3      for V ← Variables(G) do
4          L ← CreateLocalDFA(G, V, F)
5          for S ← States(L) do
6              for T ← Transitions(L,S) do
7                  suppress ← true
8                  P ← Predicate(L,T)
9                  for S' ← GlobalStates(L,S) do
10                     for T' ← Transitions(G,S') do
11                         if CanSuppress(G,T') then
12                             continue
13                         P' ← Predicate(G,T')
14                         if SAT(P ∧ (ρ = V) ∧ P') then
15                             suppress ← false
16                 if suppress then
17                     MarkSuppressed(L,T)
18         LS ← LS ∪ {L}
19     for S' ← States(G) do
20         for T' ← Transitions(G,S') do
21             if CanSuppress(G,T') then
22                 continue
23             NC ← NC ∨ Simplify(Predicate(G, T'), ρ == Fresh())
24     return ⟨LS, NC⟩
```

Figure 4.13: Construct local state machines

4.6.5. Suppressing Events Locally

The local machine can hide events when it can prove they would otherwise be processed by suppressible transitions in the global machine. Algorithm in Figure 4.13 is used to create all the data structures needed to suppress events locally. It takes the global state machine G as input along with the user-defined filters F and produces, as output, a collection of local state machines (L_i) as well as a negated condition (NC), explained below.

The algorithm works by iterating over every location or variable in the IV specification (line 3) and calling CreateLocalDFA to build the local state machine (line 4). It then walks over each local transition (T) and attempts to mark the transition as locally suppressible. To do so, it looks up all the possible global states corresponding to this local state (line 9) and checks whether the local transition can process an event that is also processed by, and is not suppressible for, some global transition T′ from one of these states (line 14). If not, then all events that trigger T must be part of a suppressible transition in the global DSFA, so the event is suppressed.

In Figure 4.12, events matching `ET!=ADD` in state $\{\{S\}\}$ are suppressible: for each global state in the set ($\{S\}$), this event must be processed by a suppressible global transition.

Negated condition The final part of the algorithm (lines 19 to 23) computes a "negated condition." This condition captures the case where the local NF may not correspond to any named location in the IV specification, e.g., the NF instance is not `$X`, but it still may observe a relevant event as `NOT $X`. We observe, in such a case, the current machine can not possibly know anything about the global automaton state since the other NF instances that also are not `$X` may be sending events that match `NOT $X` transitions. The fix is simple: the algorithm computes the disjunction of all

the transition predicates in the global state machine subject to the knowledge that the location ρ does not match any variable (line 23).

In the running example, the algorithm computes: (`ET==ADD` $\wedge$ `Z==$X`) $\vee$ (`ET==ADD` $\wedge$ `Z!=$X`) $\vee$ (`ET==REMOVE` $\wedge$ `Z==$X`), where `Z` is a fresh variable that is guaranteed to not match any location in the predicate. The above condition simplifies to `ET==ADD`. This means that the local machine *must* send any `FLOWCACHE_PRIMARY_ADD` events to the global verifier regardless of its local state.

Note that non-location variables may introduce some uncertainty at the local verifier, which may not be sure what other NF instances have observed for their value. To address this, *Aragog* first tries to generate a predicate that accounts for any possible variable assignment by enumerating all possible assignments from their ==/!= expressions, replacing their occurrences in the negated condition, and computing the disjunction of the resulting predicates. If any variables or arithmetic operations remain in the disjunction, *Aragog* will simply not suppress any events, which is always safe.

4.7. Runtime System

We next describe the *Aragog* runtime.

4.7.1. Workflow Overview

We begin with the common case: NF instances synchronized via PTP send events—at runtime—to a co-located local agent via traditional IPC mechanisms. This local agent applies transformations, computes supressions using local state machines, and then sends any non-suppressible events to the global verifier via a set of Kafka brokers.

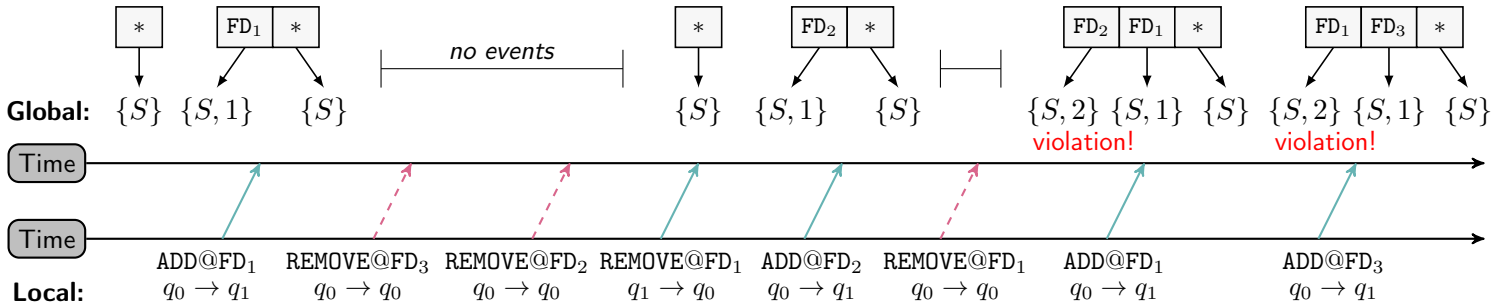


Figure 4.14: Distributed execution for the example from Figure 4.4 on an example sequence of events for N flow deciders. Time progresses from left to right. Local events are shown along the bottom line with the local state of the flow decider. We use $q_0 = \{\{S\}\}$ and $q_1 = \{\{S\}, \{S, 1\}, \{S, 2\}\}$. The global verifier's state is shown at the top. Red, dashed edges indicate suppressed events.

Filtering, mapping, and grouping After ingesting the stream of PTP-timestamped events, local *Aragog* agents co-located with the NF first apply any applicable transformations—`FILTER`, `MAP` or `GROUPBY`—to the raw stream. As each IV specification can have a different set of transformations, this may require *Aragog* to duplicate the incoming stream of raw events (it tries to avoid doing so when possible). The end result is a set of keyed event streams: one stream for each combination of policy and `GROUPBY` key.

Computing suppression The next step, also performed locally, is to determine whether events in each keyed stream are suppressible. *Aragog* passes the events through the localized state machines — one for each location referenced in each IV specification. For a given event and IV, *Aragog* suppresses the event when (1) all localized instances of the IV specification would take a suppressible transition when fed the current event and (2) the event does not satisfy the negated condition. If either constraint is false, *Aragog* sends the event to a Kafka queue for the given keyed event stream.

As a concrete example, Figure 4.14 shows processing of a series of events with the specification in Figure 4.4 and with the same `GROUPBY` key. The first event is an `ADD` event at flow decider $\texttt{FD}_1$. After seeing this event, $\texttt{FD}_1$ will transition locally from state q_0 ($\{S\}$) to state q_1 ($\{\{S\}, \{S, 1\}, \{S, 2\}\}$). Since this transition is not suppressible, the event is sent to the verifier. The next event is a `REMOVE` event that takes place at $\texttt{FD}_3$. This particular transition *is* suppressible and the negated condition (`ET==ADD`) is not satisfied, thus, the event is suppressed.

This suppression can substantially reduce the number of events received by the global verifier. For example, with three replicas (including the primary), a correct execution of Figure 4.4 *Aragog* would receive—after suppression—just 2 out of 4 events (the add and remove at the primary but not the 2 suppressed removes at nodes other than `$X`).

Global state machines Pulling from Kafka is a cluster of Flink instances running the global versions of the IV state machines. Both the Kafka and Flink instances are automatically provisioned, checkpointed, assigned `GROUPBY` keys, and load balanced to worker nodes. As Flink does not guarantee that events from different NF instances will arrive in order, *Aragog* temporarily stores and reorders events in the Flink workers with an efficient priority queue before passing them to the associated state machine.

One challenge is how long to wait for delayed events. One approach is to maintain a list of all NF instances along with the timestamp of the last event they sent to this partition and only process time t when we have seen events from all instances up to $t+latency$. Unfortunately, most NF instances do not interact with most flows/policies and sending 'null' events to advance the timestamps of every partition would be costly. Instead, *Aragog* relies on the assumption of a maximum latency t_{max} and handles violations of this assumption with the techniques in Section 4.7.3.

Aragog will hold each event for t_{max} time before running it through the global DSFA. While processing events for a given IV specification, the verifiers will track all of the possible states in which the associated state machine could be, as well as all potential values of the IV specification's variables (see Section 4.7.2 for details). If any of the possible states is a 'final' state in the IV's DSFA, *Aragog* will raise an alert.

Consistent sampling If scaling is still challenging despite sharding the verifier, filtering relevant events, and suppressing events locally, *Aragog* provides a final mechanism that lets users trade performance for completeness by sampling a consistent set of events with consistent hashing based on the `GROUPBY` key (e.g., a 5-tuple for NATGW). In this way, each group is itself complete though false negatives remain possible when violations occur for keys that are not sampled.

4.7.2. (Location) Variable Tracking

Aragog tracks *all* possible instantiations of variables (location or otherwise) at runtime using a multi-level tree data structure (shown at the top of Figure 4.14). Intuitively, the tree captures the state the global automaton would be in for every possible instantiation, with the leaves of the tree as the state and the interior nodes as variable assignments. Every variable is assigned a single level of the tree.

Let the number of variables (location or otherwise) for an IV specification be n. When the system starts, the DSFA is in the start state, $\{S\}$, for all possible variable assignments. This is represented as a degenerate tree with height $n+1$ and a single leaf pointing at the start state $\{S\}$. The interior nodes are all set to $*$, indicating no constraints on the n variables. For every incoming event, we advance the DSFA using the state and variable assignments of every leaf. Whenever a predicate is encountered that references a variable, V_i, if $V_i = *$ is an ancestor of the current leaf we split execution into a case where V_i satisfies the predicate and a case where it does not. The $(n-i)$-height subtree under $V_i = *$ may need to be cloned.

In the example of Figure 4.14, there is only one variable (`$X`) and, thus, only two levels in the tree. The system starts in the degenerate case where `$X` $= *$. After the first `ADD` event arrives at the verifier from $\texttt{FD}_1$, we fork the tree to separate out the old case and a new case for `$X`=$\texttt{FD}_1$. When `$X` is $\texttt{FD}_1$, the verifier takes the transition ($\texttt{ET} == \texttt{ADD} \wedge \rho == \texttt{\$X}$) to state $\{S, 1\}$: the current location ρ is $\texttt{FD}_1$, and `$X` is also $\texttt{FD}_1$. Otherwise if `$X`!=$\texttt{FD}_1$, it takes the self-loop transition to remain in $\{S\}$. For the next event from $\texttt{FD}_1$ (`REMOVE`), there is no new case to fork, and applying the transition to both cases in the tree leads to both being in state $\{S\}$ once more. Therefore, the states are collapsed together back to $*$. This process continues until the second to last event where a violation is detected for the case where `$X` $= \texttt{FD}_2$ due to a duplicate add at $\texttt{FD}_1$. The final event (`ADD` at $\texttt{FD}_3$) leads to a second violation,

where now $\texttt{\$X} = \texttt{FD}_1$, and is subsequently caught by the implementation.

4.7.3. Fault Tolerance

Failures and message drops/delays can cause *Aragog* to become desynchronized from the ground-truth state of the system. Even so, *Aragog* is able to guarantee both precision and recall of typical network violations under the assumption of 'partial synchrony' [61], i.e., that there exists a time, t_s, after which there is some upper bound on message delivery time.

- *Recall:* Under a partial synchrony assumption, *Aragog*'s practice of creating a self loop in the initial state of the SFA means all violations whose trace begins *after* t_s are accurately modelled in the state machine and detected.

- *Precision:* *Aragog*'s precision guarantees are less complete, but still hold in practice. Specifically, we observe that all of the IV specifications we studied contained some property where flow state would eventually be dropped in reaction to a `REMOVE_ENTRY` or TCP `FIN`/`RST` event; such transitions are common in networked systems and ensure that any desynchronized state machine instances will eventually transition back to the initial state.

In addition to the above, Flink provides guarantees that successfully pulled events are processed by the state machine exactly once. *End-to-end* guarantees of exactly once delivery between Flink and Kafka are also possible, but would incur the overhead of atomic exporting of NF events, transactions, and rollbacks. Instead, *Aragog* chooses to rely on partial synchrony and to alert users after the fact when false positives may have occurred. This can happen when an event arrives with a timestamp earlier than the last processed event, two events arrive from an NF instance with a gap in their sequence numbers, or an NF instance (and its local agent) fail. Upon restarting, the agent can immediately resume exporting events, but the local state machine may be out of sync. In this case, it can temporarily export all events (which is always safe)

until it can synchronize with the global verifier to rebuild the local state machines from the global verifier's state.

4.8. Implementation

We have implemented *Aragog* with more than 6,500 lines of Java 8 code, packaged with Maven v3.6 and more than 2,000 lines of C++ code. The implementation consists of two major components: the compiler and runtime system. It can be found at: https://github.com/microsoft/aragog.

The compiler takes as inputs an event format specification as described in Section 4.5.1 along with a set of IV specifications in the format of Section 4.5.2. For each IV specification, it generates the global state machine, the resulting local state machines, information about suppressible events, and a slew of other metadata about variables, filters, and partitioning. The lexer and parser use the ANTLR v4.7 [8] parser generator, and the SFA construction and determinization use the open-source `symbolicautomata` library [13], but with the addition of a custom Z3-based [14] theory of Boolean Algebra designed to support our IV specification language.

We built the runtime system on top of Apache Flink [9] and Kafka [10]. These frameworks are designed for scalable and robust stream processing and provide, intrinsically, fault-tolerant and stateful processing, exactly-once semantics, load balancing, flexible membership, checkpointing, etc. The local agents, implemented in C++, ingest events directly, then filter, map, and suppress events as necessary before sending them to Kafka. The global verifiers, implemented in Java using Apache Flink, pull from Kafka into a timestamp-based priority queue from which events are dequeued after waiting for a maximum delay; violations are logged to disk. We place the verifiers off of the critical path to avoid any impact on production traffic.

Network Function	Invariant Description	LoC	States	Transitions
NAT Gateway	`nat_decider_open`: After a PW goes into closed state, at least one replica also goes into closed state.	14	4	10
	`nat_consensus`: All TCP flows are open only after consensus.	5	2	4
	`nat_open_to`: Open flows are timed out after 4 minutes of inactivity.	5	4	12
	`nat_primary_single`: There is a single primary per flow.	10	3	7
	`nat_primary_to`: The NATGW does not start an idle timeout for active flows.	13	6	18
	`nat_same_consensus`: After TCP flow U is terminated, the next flow for U achieves consensus.	12	5	15
	`nat_syn_to`: Flows with a TCP handshake in progress timeout after 5 seconds of inactivity.	5	4	12
	`nat_udp_same_consensus`: If UDP flow U times out, the next flow for U achieves consensus.	12	6	17
Firewall [12]	`fw_consistency`: *all* Firewall instances should block suspicious IPs after a block rule is added.	6	4	12
	`fw_client_init`: Ensure a flow can only be open after a client initiates it.	4	2	4
	`fw_syn_first`: Data packets are only allowed after a SYN is sent.	4	2	4
DHCP	`dhcp_reuse`: Leased addresses are not re-used until expiration or release.	6	4	12
	`dhcp_overlap`: Leases should not overlap between DHCP servers.	6	3	7

Table 4.1: List of example invariants that *Aragog* can implement for several common network functions and systems.

4.9. Evaluation

We evaluate *Aragog* in CloudLab [152] with a number of network functions and along a number of dimensions.

The deployed NAT gateway (Section 4.2) We use two event traces captured from two different builds of the NAT gateway to evaluate *Aragog*. The builds capture the introduction of a set of bugs that arose from the change of an interface between two internal components, with V1 from before the change and V2 from after. The traces are both for 7 flow deciders over a 30 minute interval, but they export a different number of packets (V1: 23.7M; V2: 9.0M) owing to changes in the protocol. The production deployment of NATGW does not yet support fine-grained clock syn-

chronization, but our operators plan to add it in the system's next version. Instead, we capture the event traces and correct for time drift using a set of known synchronization points within the event stream. In total, there are eight IV specifications for NATGW (see Table 4.1).

A distributed firewall We also execute a collection of micro-benchmarks using an open-source, stateful, and distributed firewall implementation built on `iptables`, `conntrackd`, and `keepalived` [12]). On the firewall, we check various invariant violations, some of which were derived from [19]. The list of specific invariant violations we check are listed in Table 4.1.

We deploy this firewall on a topology with four clients, four internal hosts on a single LAN, and four firewall nodes interposing between the two groups. The firewalls are configured as two high-availability groups with one primary and one hot standby each. Each primary-standby group shares a virtual IP with the VRRP protocol. We base the traffic between external hosts and internal servers on the traces provided in [21].

DHCP To show the flexibility of *Aragog* and its language, we also give examples of DHCP invariant violations in Table 4.1. With our current implementation, the operator needs to write just 6 lines to express the invariant violations. Each of the state machines uses a small number of states and transitions.

Evaluation metrics We evaluate *Aragog* along a number of key dimensions: lines of code, throughput, latency, and CPU overhead. In addition, our micro-benchmarks show *Aragog*'s ability to scale as the number of nodes in the NF deployment increase by demonstrating the benefits of our event suppression scheme. Finally, we find *Aragog* is able to identify bugs in production systems. In particular, we were able to identify four bugs in the NAT gateway which were confirmed by our operators. Similarly, in the firewall, *Aragog* was able to find a series of injected configuration

Invariant Violation	Version 1	Version 2
nat_decider_open	0	0
nat_consensus	0	0
nat_open_to	1	45019
nat_primary_single	0	0
nat_primary_to	1	29964
nat_same_consensus	536	259
nat_syn_to	0	2697
nat_udp_same_consensus	0	0

Table 4.2: Violations found in traces for NATGW versions. Note that V1's trace contains more events than V2's, which may account for the difference in `nat_same_consensus` violations.

errors over real traffic traces.

4.9.1. Bugs Identified by *Aragog*

NATGW Bugs Running the traces through *Aragog*, we discovered violations of `nat_open_to`, `nat_primary_to`, `nat_same_consensus`, `nat_syn_to`, all of which were confirmed as caused by bugs by the NATGW team. Table 4.2 shows the absolute number of violations observed for each.

`nat_open_to` was by far the most frequent violator in V2. Discussions with our operators revealed that in V2, this violation (and that of `nat_syn_to`) were due to related bugs in the code: it had taken operators over an hour to identify the issues while *Aragog* identified it in under a minute. Although `nat_open_to` also had a violation in V1, further examination revealed that the violation in V1 was due to an expected consequence of eventual consistency—specifically one of the replicas was getting update messages from the packet worker but the primary did not and therefore started a timeout for the flow. This led us to start checking for `nat_primary_to`.

Also prominent in both systems were violations of `nat_same_consensus`. This violation occurred because the flow was not closed or removed properly from one of the replicas. The operators suspected this could be an issue, but never had a method to test that hypothesis. *Aragog* confirmed the problem and helped the developers to

formulate the test setup to reproduce the issue.

Bugs in the distributed firewall rules For the firewall, we manually injected bugs in the firewall configuration to test *Aragog*'s ability to identify this category of errors. The injected issues, for instance, always allowed external traffic from a particular address range into the internal network, violating `fw_client_init`. *Aragog* found all of them.

4.9.2. Throughput of *Aragog*

Aragog's global verifier keeps track of the set of possible states for each IV specification and the possible values for each variable/location. Thus, *Aragog*'s throughput is directly correlated with the number of IVs checked (Figure 4.15). To evaluate this scaling, we run the V1/2 traces through all the 8 NATGW IV specifications using a single Task Slot on the global verifier (running on an Intel(R) Xeon(R) E5-2450 processor CPU @ 2.10GHz machine). We upload the entire trace on Apache Kafka after local processing to measure the maximum throughput a single task slot of Apache Flink of the global verifier can process. In Figure 4.15 we randomly select n among the NATGW invariant violations and see the performance. As each type of invariant violation exhibits different resource requirements, we see more variance when the number of type of invariant violations selected is low.

With a single task slot, our optimizations allow *Aragog* to scale and process over 500,000 events per second for a single invariant violation type (over 30,000 for 8). Adding more task slots does not improve the performance as our implementation is parallel in nature and a single task slot is already using multiples core in a single machine.

Aragog scales linearly as we add more machines to the global verifier (Figure 4.16). Scaling with multiple machines avoids the bottleneck of CPU and I/O.

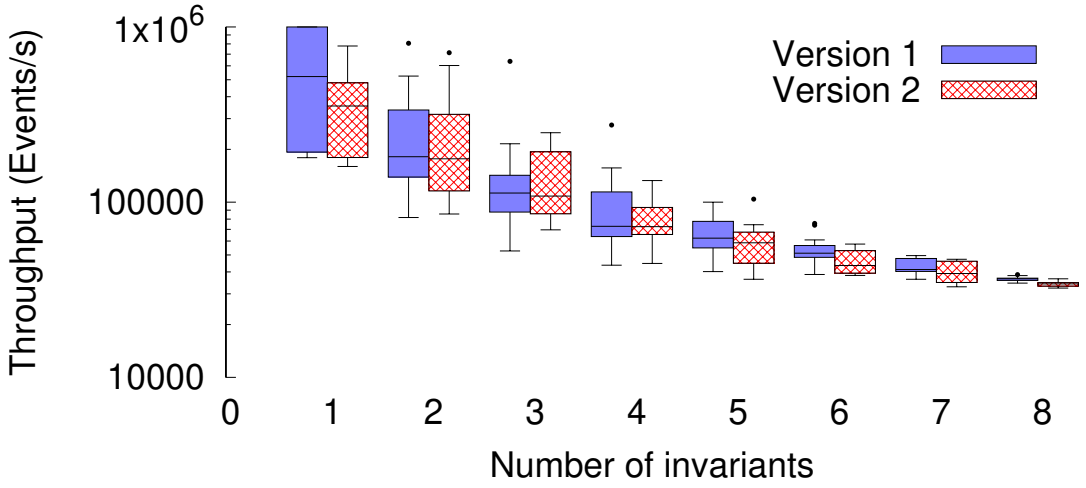


Figure 4.15: The throughput in events/second for an executor of *Aragog* on the trace.

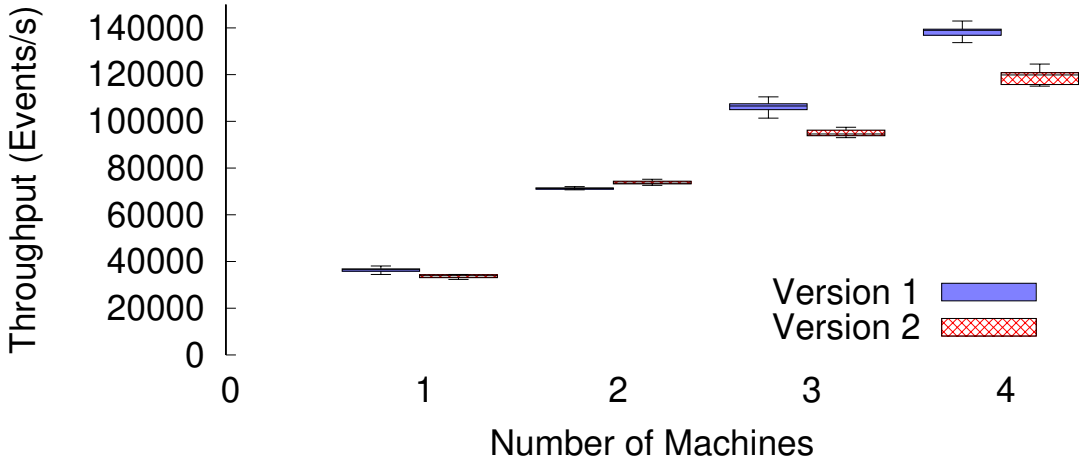


Figure 4.16: Throughput of multiple *Aragog* verification server checking all 8 types invariant violations

4.9.3. Overhead of *Aragog*

To measure the memory and CPU overhead of *Aragog*, we study its behavior while verifying the distributed firewall. In Figures 4.17, 4.18 and 4.19, data is divided into separate groups. 'Primary' represents the verifier running at the primary firewall. 'Backup' represents the verifier running at the hot-standby firewall. 'Manager' and 'executor' represent the Apache Flink job manager and executors, respectively. The

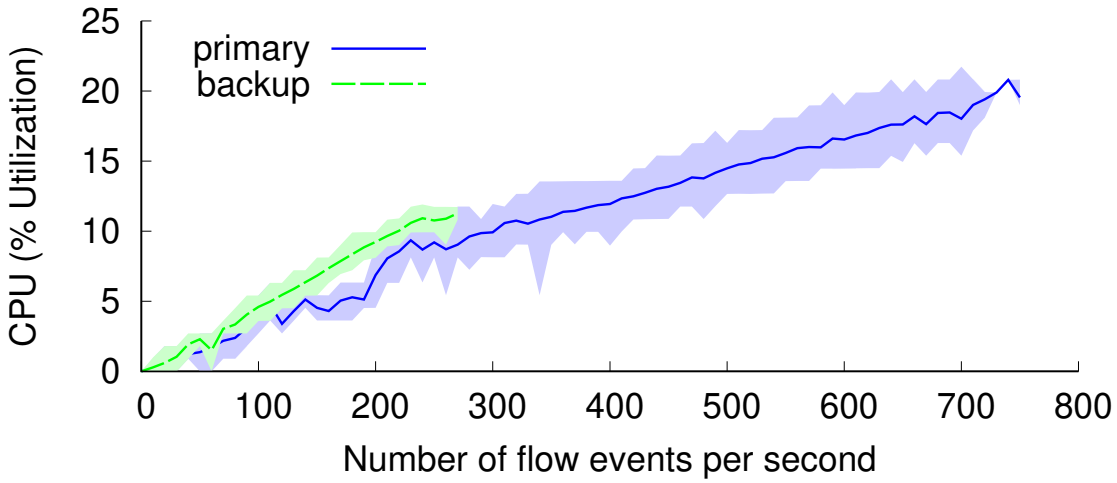


Figure 4.17: CPU utilization by *Aragog*'s local component. The graph shows CPU utilization of the local verifier at both the primary and backup firewall.

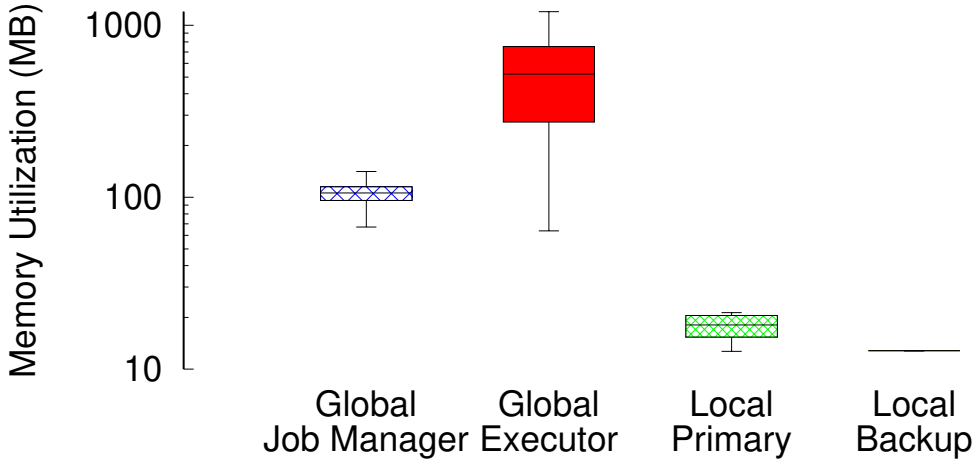


Figure 4.18: Memory utilization of verifier in MBytes.

global verifier runs on the executors.

We see that in Figures 4.17 and 4.18, the overhead of the local verifiers is low. This is important as the local components are co-located with the production NF instances. To that end, the CPU utilization of the local verifier increases linearly with the number of flow events per second. We also observe the CPU and memory usage for the local verifier is higher at the primaries as they tend to generate more events. Memory at

Process/Location	Resource	Spearman correlation
job manager	CPU	0.14700
job manager	memory	−0.59379
executor	CPU	0.78481
executor	memory	−0.38373
primary	CPU	0.88916
primary	memory	−0.18253
backup	CPU	0.93618
backup	memory	0.24768

Table 4.3: Spearman Correlation between number of events/s and resource utilization at different locations of verifier while running the firewall.

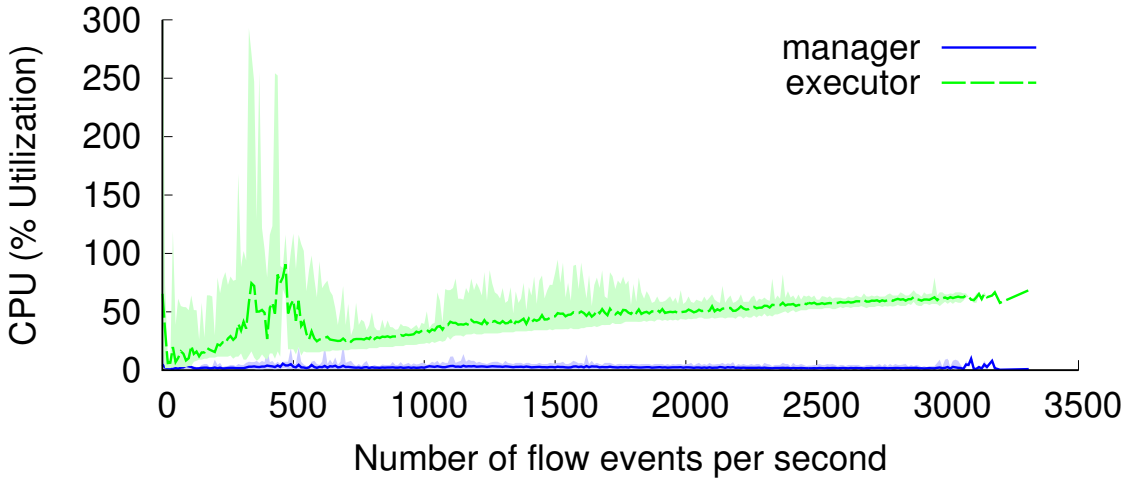


Figure 4.19: CPU utilization by *Aragog*'s global component. 'Manager' and 'executor' refer to the Flink node designations.

the local components is much less correlated (Table 4.3), partly due to *Aragog*'s small memory footprint (Figure 4.18).

The global verifier has higher CPU (Figure 4.19) and memory (Figure 4.18) than local verifiers as the global verifier is implemented in Java using Apache Flink. We have set the maximum memory of job manager to 1 GB and executor to 2 GB. In our graphs, we are plotting active memory in Java's heap for the global verifier rather than used memory to avoid including memory waiting to be cleaned up by the Java GC.

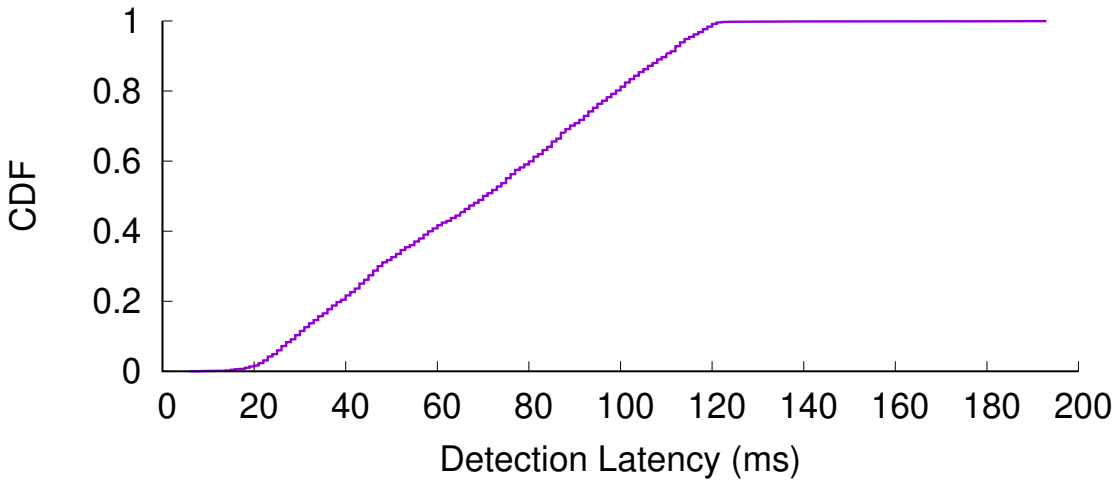


Figure 4.20: Latency (alert time – packet time) for detecting a violation in the distributed firewall.

Version	Generated	After Filter	After Suppression
V1	189M	92.9M **(49.1%)**	70.2M **(37.1%)**
V2	72.2M	36.7M **(50.8%)**	28.0M **(38.8%)**

Table 4.4: Total number of generated events, events processed after filtering, and events processed after filtering and suppression for the NAT gateway with all 8 IV specifications.

Figure 4.20 shows the CDF of *Aragog*'s *time to detection* for violations in the distributed firewall function. The time to detection is low: in the median it takes roughly 70 ms from the time the event was executed (the violation occurred) at the NF instance until *Aragog* raises an alert.

4.9.4. Efficacy of Suppression

Each optimization in *Aragog* improves scalability by reducing the number of events sent to the global verifier (reducing the network overhead and the number of events processed at the global verifier). Filters remove the need to send events that are not pertinent and reduce the number of events sent to the verifier by up to 61% for the NATGW (Table 4.4). Suppressible events can further reduce this number (by up to an additional 12% in our experiments).

4.10. Discussion

Aragog is a lightweight verification framework for verifying distributed network functions. To scale to large systems with minimal overhead, *Aragog* leverages a two-tiered setup with local monitors at each NF instance sending events to (and hiding events from) a collection of sharded global verifiers. While *Aragog* can verify any distributed system, its scalability will depend on whether the invariant violations of interest can utilize its sharding and suppression optimization effectively.

Finally, as *Aragog* is the first to verify distributed network functions at scale (and at runtime), there are a number of aspects where follow up work may be needed. Included in this set are explorations of other time synchronization protocols, e.g., [68] or some other lightweight and precise event ordering mechanisms. Also for future work are innovations in atomic event export and transactions over streams in *Aragog*.

4.11. Summary

Aragog is a lightweight verification framework for verifying distributed network functions. To scale to large systems with minimal overhead, *Aragog* leverages a two-tiered setup with local monitors at each NF instance sending events to (and hiding events from) a collection of sharded global verifiers. While *Aragog* can verify any distributed system, its scalability will depend on whether the invariant violations of interest can utilize its sharding and suppression optimization effectively.

Finally, as *Aragog* is the first to verify distributed network functions at scale (and at runtime), there are a number of aspects where follow up work may be needed. Included in this set are explorations of other time synchronization protocols, e.g., [68] or some other lightweight and precise event ordering mechanisms. Also for future work are innovations in atomic event export and transactions over streams in *Aragog*.

CHAPTER 5

FP4: Fuzz Testing for P4

5.1. Introduction

Computer networks have evolved to include more flexible platforms in which data plane functionality can be defined by programmers using domain-specific languages like P4 that describe devices' data plane processing. These devices are opening doors to better support applications [158, 94] and to improve network management [84]. Although the increased programmability offers great benefits, it also brings the increased risk of introducing new bugs that are difficult to catch, either in the data plane, control plane, language compiler, ASIC implementation, or any combination of the above.

Bugs may arise as a result of anything from simple programmer typos to divergent interpretations of the P4 specification [56]. They can even include behavior that spans multiple packets because switches can store and recall state in registers, trigger control plane transitions, and reconfigure match-action rules as a result of incoming packets. Combined, all of these factors increase the complexity of bugs in switches [119, 59].

To reduce bugs, recent work has suggested static verification to prove the correctness of P4 programs [59, 119, 65]. For pure, stateless data plane programs, static verification is often effective since there are no complex pointer-based data structures or loops, making analysis both more accurate and tractable. However, real forwarding behavior depends on many other components: the control plane; the exact past, present and future match rules; the compiler translation; the switch state including registers; and specifics of the hardware implementation. For example, in the process

of developing *FP4*, we discovered a subtle compiler/runtime bug in our installed SDE in which a multicast primitive in the default action of a table with no entries does not properly multicast the packet. All other commands in the default action execute correctly as does the same setup in the provided simulator. Static verification tools that only consider the P4 program itself cannot catch this class of bug.

We note that, in traditional programs, developers often rely on fuzz testing to catch this wider class of bugs. A fuzzer generates semi-random inputs to discover assertion failures, memory leaks, and crashes. Fuzz testing is able to evaluate applications in their natural environment (ignoring issues that are impossible to reach and catching issues that only arise in the presence of the application's surrounding components). Fuzzing, and particularly blackbox and greybox fuzzing, also tends to scale well with the complexity of control flow and state. As others have noted [42], these approaches often find more bugs than whitebox approaches like static verification and symbolic-execution-based test case generation as the latter either (a) spend significant time doing program analysis and constraint solving or (b) further sacrifice precision, e.g., by approximating functions that are hard to reason about analytically, such as hash functions.

In this work, we observe that, when applied to programmable switches, not only does fuzzing allow a developer to check the entire device *in vivo*—incorporating effects of the data plane, control plane, ASIC implementation, and compiler—it also allows the fuzzer to leverage the intrinsic hardware parallelization, pipelining, and acceleration of packet processing in today's network devices. Explicitly optimized for fast packet processing, switch-based fuzzing potentially enables input testing that is orders of magnitude faster than is possible in a CPU.

To that end, we present *FP4*, a greybox fuzz testing framework for P4-programmable network devices that is both (a) full-stack and (b) line-rate. *FP4* feeds semi-random

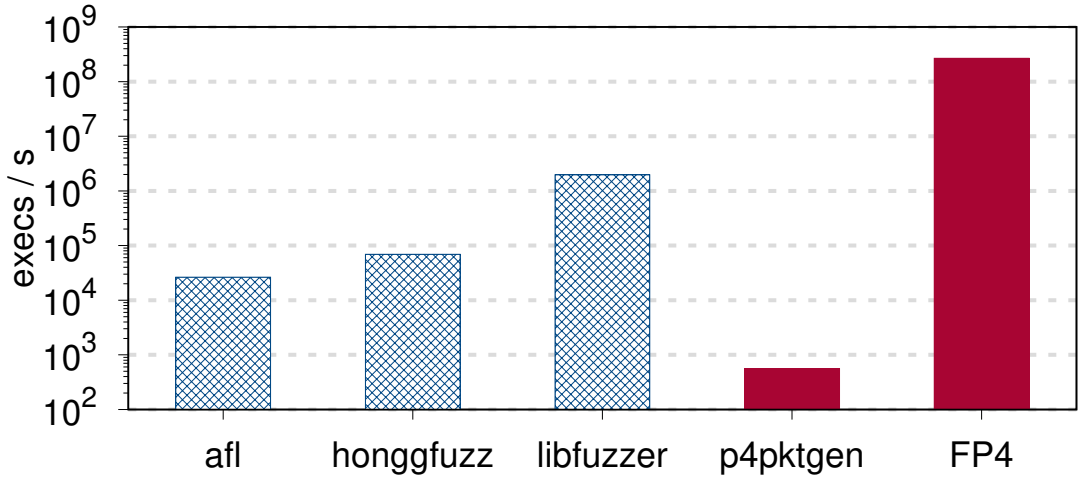


Figure 5.1: Maximum possible throughput of a single instance of modern fuzzing frameworks, both those for traditional programs (AFL, honggfuzz, and libfuzzer) and for P4 programs (p4pktgen and FP4). In each case, the fuzz target is an empty function or data plane program.

packets to real programmable switches to attempt to attempt to trigger violations of programmer-specified assertions. The switches are purposely kept as faithful as possible to their production deployments and run instrumented versions of their original P4 programs and control planes.

As a preview of the performance benefits of *FP4*'s approach, we measure executions per second for several traditional program fuzzers and p4pktgen, a software-emulated P4 fuzzer. The results are shown in Figure 5.1. All systems except *FP4* were on an Intel Xeon E5-2660v3 2.60 GHz CPU core with empty programs (empty parser and control block in the case of p4pktgen); thus, these numbers represent an upper bound for prior work. *FP4* has two orders of magnitude higher throughput than the fastest traditional software fuzzer and almost 6 orders of magnitude faster than p4pktgen.

Unfortunately, implementing fuzz testing in P4 requires addressing numerous challenges. First, to take advantage of the specialization of modern switch packet processing and achieve fast fuzzing speeds, we need methods to both generate and execute fuzzing at line rate. Second, line rate packet input generation, on its own, is insuffi-

	p4v	**vera**	**p4wn**	**p4pktgen**	**P6**	***FP4***
type	static	static	static	runtime	runtime	runtime
line rate	N/A	N/A	N/A	✗	✗	✓
DP logic	✓	✓	✓	✓	✓	✓
DP state	✓	✗	✓	✗	✓	✓
compiler	✗	✗	✗	✓	✓	✓
control plane	✗	✗	✗	✗	✗	✓
hardware	✗	✗	✗	✗	✓	✓

Table 5.1: Comparison of the features of a selection of P4 verification and testing frameworks, including whether they can catch bugs in data-plane logic, stateful behavior, the compiler, the control-plane, and hardware.

cient to find all bugs as the space of possible packets is large and often redundant. In both traditional and programmable-switch fuzzing, careful choice of inputs is critical to good fuzzing performance. Finally, typical methods to handle stateful behavior involve generating sequences of inputs and resetting the state between sequences (e.g., [148]). Unfortunately, resetting switch state is a fundamentally expensive operation that would severely limit fuzzing performance.

To address the first two challenges, *FP4* takes inspiration from two different fuzz testing approaches: generator-based and coverage-guided fuzz testing. Generator-based fuzzers [143] generate semi-random input such that inputs are passed through the input sanitation of the program. *FP4* knows the structure of the input from the headers and how they impact the processing of packet from the parser; and it uses it to generate valid packets. Coverage-guided fuzzing, on the other hand, leverages program instrumentation to trace the code coverage reached by each input and uses this information to make informed decisions about which inputs to mutate to maximize coverage. *FP4* tracks the actions visited by each packet in the dataplane by marking bits in the header. Both are implemented, tracked, and learned quickly with the help of a second programmable switch.

To address the third challenge, *FP4* splits switch state into a few categories. For

data-plane-only state, it leverages the fact that switches are meant to run continuously and, thus, most network tasks allow for intrinsic state resets (e.g., when a counter rolls over or a flow entry times out) [181]. Thus, continued fuzzing will eventually allow the switch to re-explore previous states. For everything else (i.e., control plane, table entry, or configuration state), *FP4* borrows another idea from traditional fuzz testing—context-sensitive branch coverage [46]—that seamlessly integrates state changes into greybox fuzzing approaches.

We implement and deploy *FP4* to a hardware testbed in order to instrument and debug real P4 programs. *FP4* works by modifying the input P4 program in a way such that it has no impact on normal packet processing. It also adds an extra header that stores information to track the actions visited and assertions failed by the packet. *FP4* uses this tracking information to generate new seed packets. Our work makes the following contributions:

- We leverage the observation that programmable switches can generate semi-random packets at line-rate to design, implement and evaluate fuzz testing framework for P4 programs that generates test packets 6 orders of magnitude faster than similar work for P4.
- We introduce a novel technique to instrument P4 programs to track their coverage and check for assertion failures at line rate.
- We implement an *FP4* prototype and evaluate it on a diverse set of P4 programs. Our results show *FP4* achieves 100% coverage quickly – in <1 min in most cases.

5.2. Background

In this section, we cover the challenges of discovering bugs in P4 switches and the possible role of fuzz testing.

5.2.1. Potential Bugs in P4 Programs

Bugs can occur in any point of the deployment and execution of a P4 program. They can include but are not limited to:

Bugs in the application logic The most straightforward class of bugs exists in the P4 code itself. In some cases, these issues are a result of ambiguities or subtleties in the language specification [56]. More generally, however, programmers are fundamentally fallible and just as capable of introducing bugs to P4 programs as they are to traditional code, especially when trying to reason about edge cases or complex interactions between features. For example, a P4 reference program previously contained a bug where ACL rules were incorrectly applied to control-plane traffic [142].

Issues in the compiler or hardware implementation The P4 program must be compiled to run on the hardware and optimized to adhere to resource limitations on pipeline stages, SRAM, TCAM, etc. As above, the programmers of these components are also fallible, creating instances where an otherwise correct P4 program produces unexpected behavior. While this class of bugs is typically rarer due to longer development cycles, lower-level specifications, and heavier testing, the above multicast issue and a glance at the errata of any processor or compiler documentation validates their presence. These are among the most difficult type of error to diagnose.

Bugs in the control plane Switch operation depends on the combination of the data and control planes. While the data plane is responsible for handling per-packet processing, the control plane—operating in parallel on a general-purpose CPU—is responsible for managing the data plane and handling all of the tasks that are too complex for line-rate processing. These include installing, updating, and removing data-plane table rules as well as executing routing protocols. All of these can evolve based on the sequence of incoming packets—real control planes are both dynamic and

stateful.

Switch misconfigurations Finally, network switch behavior is also affected by switch configuration options like knobs in the traffic manager, buffer slicing, and port speeds. These configuration options can be independent and set separate from either the traditional data plane or control plane programs. Errors can arise either from operator misconfiguration or through interactions with other issues, e.g., in the hardware implementation.

Bugs can occur within any of the above components, and some only manifest when issues in multiple layers combine.

5.2.2. Fuzz Testing

For traditional applications, programmers often augment their software engineering workflows with fuzz testing [72]. Fuzz testing feeds the program a set of random inputs and observes whether the program behaves correctly on each such input. This process is able to automatically discover bugs, even when those bugs result from complex runtime behavior and interactions between heterogenous systems.

As prior work has noted, however, the naïve approach of random inputs (i.e., pure blackbox fuzzing) can often lead to poor coverage as many inputs are simply invalid or fail to explore program paths with complex or hard-to-hit branch conditions [71]. On the other hand, approaches that try to reason precisely about the program's structure (i.e., pure whitebox fuzzers) come with their own set of issues ranging from being unable to model complex functions (e.g., hash functions) to exhibiting poor scaling that makes them not worth the extra overhead [42]. In the end, many of the most prolific fuzzers take a greybox approach that attempts to strike a balance.

FP4's input generation takes inspiration from three methods from the literature on greybox fuzzing of traditional programs:

(1) Coverage-guided fuzzing is exemplified by the widely used AFL fuzzer [73]. AFL begins with a set of seed inputs that it subsequently mutates to create random inputs that are then fed to the program. Based on feedback from the program (detailing paths and branches covered), AFL learns the quality of the inputs and selectively updates the set of seed inputs to explore new execution paths. This leads to significant improvement in the rate of coverage compared to completely random inputs.

(2) Generator-based fuzz testing allows users to write generator programs for producing inputs (see [143]). As an example, consider the structure of an Ethernet/IP protocol stack, which might accept IP-related EtherTypes and discard all other packets as corrupted or otherwise invalid. A generator-based fuzzer will generate only valid IP packet inputs. This ensures that the fuzzer does not waste time on inputs that are immediately discarded by input sanitation.

(3) Context-sensitive branch coverage is introduced in the Angora fuzzer (see [46], Section 3.2). Angora observes that not every execution of the same code block (containing a conditional branch) is equal. Instead, the current state of the program and its call stack can make an execution of the same code block materially different from prior executions. Including this context in coverage tracking therefore improves feedback and responsiveness.

5.3. Overview

Like most other coverage-guided greybox fuzzers, *FP4* is built around a single loop in which *FP4* generates a packet from a selected set of seeds, mutates the packet semi-randomly, passes it through the target (programmable switch), and computes the state and path coverage to determine whether the packet is a good candidate for a new seed. However, unlike other fuzzers, *FP4* is extremely fast. It achieves this

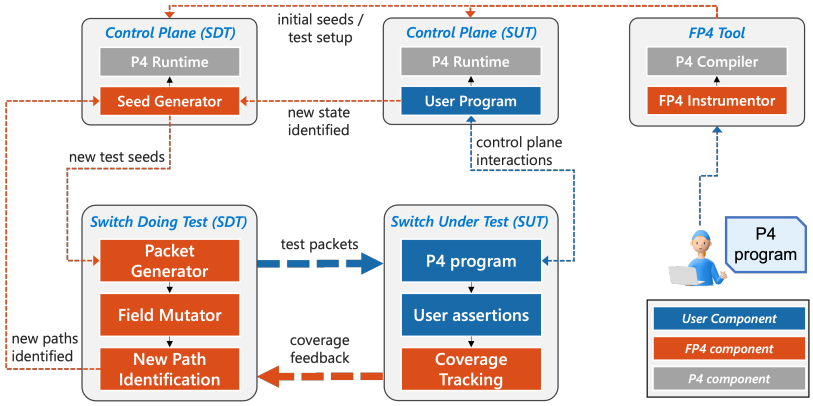


Figure 5.2: System design of *FP4*. An operator writes assertions in the P4 program. The program is an input to (1) instrumentation (Section 5.4) that adds statements to track packets and (2) program synthesizer (Section 5.5) that generates the P4 program and control plane to conduct the test. After installing respective programs on both switches, *FP4* runs fuzz testing. *FP4* generates valid packets (generator based fuzzing) and adds new seed packets based on coverage information (coverage guided fuzzing).

speed with two domain-specific insights:

1. Modern switches can execute packet processing orders of magnitude faster than commodity CPUs, and that speed is independent of the complexity of the program as long as it fits within a single pass through the switch. As mentioned, running the target system in vivo provides benefits to the speed, completeness, and accuracy of fuzz testing. A key contribution of *FP4* is to demonstrate that the fuzzer is *also* deployable to programmable switches, generating, modifying, and checking at line rate.

2. Switch programs are intended to be long-lived. This means that, in steady state, switch programs typically have intrinsic mechanisms that reset persistent state, e.g., when a counter overflows, when a ring buffer wraps, or when routes are torn down. This allows *FP4* to test most state transitions without needing an explicit reset of the switch. An exception are bugs that occur during initialization, but those are typically straightforward for operators to catch during development and canarying.

An *FP4* testbed consists of two switches: (1) the switch under test, which runs the target data plane, control plane, and switch configuration; and (2) the switch doing the test, which generates inputs, checks program coverage, and manages seed packet selection. See Figure 5.2 for a visual depiction.

Switch Under Test (SUT) The SUT executes the target switch system, including both its P4 data plane and control plane program. The system should operate identically to a real deployment with the exception of some additional annotations and instrumentation.

The annotations come in the form of a simple *operator-specified error conditions* on a given packet's contents or the state in the switch (i.e., registers, counters, and meters).

In the automatic *instrumentation step*, *FP4* then inserts code into both the data plane and (optionally) control plane programs to aid in checking for path coverage to trigger the above errors. This instrumentation takes the form of an additional packet header, an operation in every data-plane action, emulated output ports, and a couple of additional tables for bookkeeping and assertion checking. All of the above changes incur minimal overhead and, crucially, leave the original metadata, headers, and control path intact.

Switch Doing Test (SDT) Alongside the SUT, we run a second switch, the SDT. The SDT is responsible for all of the traditional tasks of a fuzzer: generating test packets, mutating them, sending them to the SUT, and checking for violations and coverage after packets return from the SUT.

It consists of a *data-plane test generator*, which is a synthesized companion P4 program that generates the test packets, mutates them, tracks coverage, and checks for assertion failures. The generator leverages a set of dynamically updated seed packets to generate billions of semi-random test packets per second. *FP4* mutates the packets

```
1  header_type fp4_header_t {
2    fields {
3      visited_action1 : 1;
4      visited_action2 : 1;
5      ...
6      assertion1 : 1;
7      ...
8      // 0 -> freshly generated packet
9      // 1 -> additional mutation needed
10     // 2 -> completed packet
11     // 3 -> state change from SUT control
12     pkt_typ : 2;
13   }
14 }
```

Figure 5.3: Structure of the *FP4* header.

at line rate using the programmability of the P4 switch. The mutations are such that they retain the validity of the packet but attempt to steadily increase the coverage of the fuzz testing.

Supporting the data-plane test generator is a *control-plane fuzzing manager* that is used to consider seed packets candidates, modify the data-plane generator accordingly, notify users of assertion failures, and perform other tasks that are beyond the capabilities of today's programmable data planes. Like in traditional networks, executing these tasks asynchronously in the control-plane CPU (while carefully maintaining correctness) allows the data plane to continue operating at line rate.

The remainder of this chapter describes the design of *FP4*'s SUT and SDT in more detail.

5.4. Switch Under Test (SUT) Instrumentation

We begin by outlining *FP4*'s modifications to the SUT.

5.4.1. Programmer Assertions

One type of instrumentation in *FP4* involves programmers adding assertions in their P4 programs that define error conditions in the processing of a packet. One example

of violation might be where the time to live field of a packet is zero, but the program fails to actually drop the packet:

```
assert(ipv4.ttl != 0 || std_metadata.drop == 1)
```

More generally, operators specify fields and their range of invalid values using basic comparison operators and boolean logic. These assertions serve as syntactic sugar that *FP4* uses to automatically generate a set of tables, actions, and table rules that will catch the assertion at runtime. Violations are marked in an *FP4* packet header that is appended to the packet (see Figure 5.3 for the header's format). Note that operators can use this syntax to detect issues that span multiple packets by manually tracking relevant information in stateful elements.

5.4.2. Coverage Instrumentation

The other type of instrumentation in *FP4* enables its greybox, coverage-guided fuzzing within the data plane. Traditional fuzzers typically track coverage at the granularity of basic blocks, adding instrumentation to each branch to record 'seed-worthy' inputs that trigger additional program coverage (i.e., that are not redundant with existing seeds). Programmable switches, with their concomitant control planes and frequent rearrangements of control flow (via control plane intervention), impose additional restrictions on what it means for an input to be worthy of use as a seed. *FP4* considers:

- *Actions:* In most P4 implementations, the most convenient single-entry, single-exit, straight-line (with the exception of ALU operations) block of code are the actions of the match-action pipeline. The goal of *FP4* is to fuzz test all possible actions, so coverage of novel actions is cause for addition to the input corpus.

- *Table entries:* The actions that are triggered and the conditions under which they are triggered are determined by the table entries of the match-action

pipeline. The overall path is defined by a sequence of table entry hits. Thus, table entries—and in particular, their union—have a massive influence on the reachability of bugs.

- *Control plane state:* Finally, while *FP4*, its design, and its assertions are primarily focused on bugs in the data plane, we note that packets sometimes pass through the control plane as part of their processing (e.g., routing updates that eventually add/remove table entries). *FP4* is not concerned with the code coverage of the control-plane program but does care about how it might eventually affect the data plane, e.g., through table entry updates.

Purposefully missing from the above set is data-plane state. While data-plane state (like table entries and control plane state) may also impact control flow and lead to additional program coverage, we found that properly tracking the uniqueness of data-plane state in the presence of per-packet register access limitations and packet reordering imposed too much overhead and too many limitations on the scope of P4 programs that *FP4* can test. We leave an exploration of more efficient methods of state tracking to future work.

In the remainder of this section, we describe the SUT instrumentation required to track changes to the above entities.

5.4.2.1 Actions Visited

To track the coverage of every action, *FP4* assigns a bit in the `fp4_header` for each action, and marks the respective bit in each packet as it passes through the switch pipeline. For example, consider the target program of Figure 5.4, there are four total actions and, thus, four reserved bits in the header. Laying it out in this way ensures that every unique path through the pipeline results corresponds with a unique 'visited' bitstring.

```
1  parser parse_ethernet {
2    extract(ethernet);
3    return select(latest.etherType) {
4      ETHERTYPE_IPV4 : parse_ipv4;
5      default: ingress;
6    }
7  }
8  parser parse_ipv4 {
9    extract(ipv4);
10   return ingress;
11 }
12 action on_l2_hit(vrf) {
13   modify_field(l3_metadata.vrf, vrf);
14 }
15 table ethernet_forward {
16   reads { ethernet.dstAddr : exact; }
17   actions { on_l2_hit; on_l2_miss; }
18 }
19 table ipv4_forward {
20   reads { l3_metadata.vrf : exact;
21           ipv4.dstAddr : lpm; }
22   actions { on_l3_hit; on_l3_miss; }
23 }
24 control ingress {
25   apply(ethernet_forward);
26   if valid(ipv4) { apply(ipv4_forward); }
27 }
```

Figure 5.4: A simple example target P4 program.

Note that if the same action is used in multiple tables, a naïve application of the above may leave ambiguity in the packet's path through the processing pipeline. *FP4* addresses this by duplicating the action and renaming it, so each action is unique to a table. The renaming has no impact on the switch hardware resources.

5.4.2.2 Control-plane State Changes

To account for the impact of control-plane changes, *FP4* augments the control plane to track its internal state. Note that this can include everything relevant to the processing of future packets, from object attributes that persist across packet events to the current state of the stack (which reflects function calls, parameter changes, and returns). This additional data can only improve coverage, but is not necessary

for functionality.

Naïvely, one could consider every packet that causes a state change as a candidate for inclusion in the seed corpus. Unfortunately, this fails to distinguish new states from previously seen ones. Instead, *FP4* leverages the CRC-32 algorithm to compute a 32-bit hash of the control plane state (all global variables followed by the sequence of function calls on the stack) that is both efficient to update and can distinguish between unique states. CRC-32 values can be updated bi-directionally (i.e., they support both pushing and popping bytes), so the hash is maintained on both function calls and returns in constant time. Our *FP4* prototype implements this approach with a semi-automatic annotation process. It currently assumes that the control plane is written in Python and all functionality is contained within a single class. Programmers annotate the class with a superclass and add a decorator to each of its methods and local variables, which wraps them to update the CRC value on each call, return, and modification (in principle, these can be automated). *FP4* also automatically stores the last-seen packet/digest from the data plane and wraps all table entry modifications.

Whenever a table entry is added or the control-plane state changes, *FP4* checks if the CRC value is novel and there is an active 'input packet/digest.' If both are true, the SUT control plane forwards the new state hash and the original headers of the input packet to the SDT for inclusion in the seed corpus.

5.4.2.3 Table Entry and Configuration Changes

Runtime updates to table entries and switch configurations can also affect the data-plane behavior, and *FP4* tracks them using a similar technique as above. Specifically, *FP4* automatically computes the CRC-32 of the string representation of all the configuration changes (the value is kept separate from the hash of internal control-plane state). For example, it interposes on the table write/update library calls to automat-

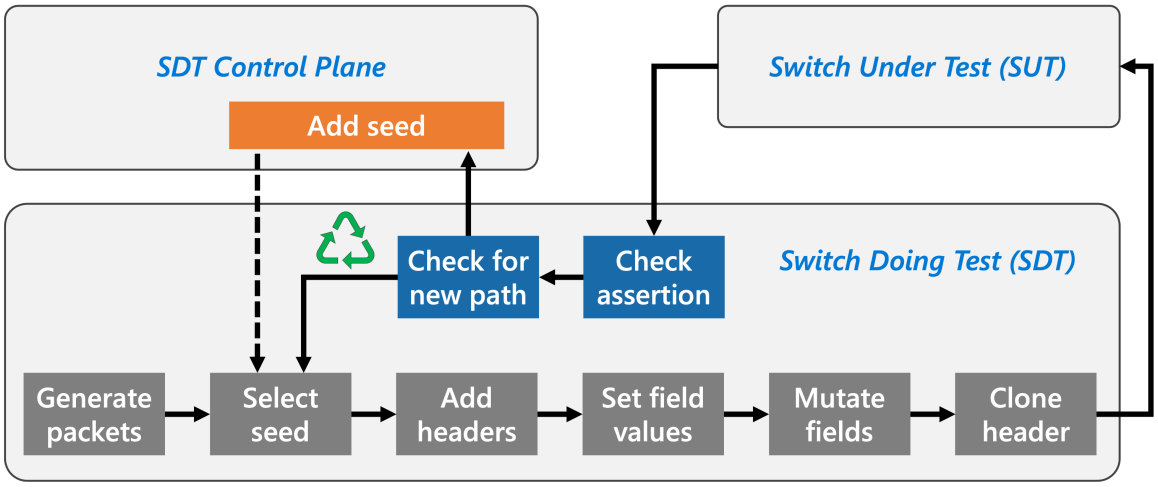


Figure 5.5: Lifecycle of a packet in *FP4*.

ically track the content of every table. It represents each entry in its runtime-CLI-command format (which provides a simple, unique representation of the entry), and computes the hash of a sorted list of such commands. As above, *FP4* automatically checks the uniqueness of the state and, if unique, forwards the input packet to the SDT.

5.5. Switch Doing Test (SDT) Design

To handle the fuzzer tasks, *FP4* leverages a second programmable switch: the SDT. For the tasks that must execute for *every* fuzzing input, *FP4* leverages the line-rate processing capabilities of the SDT's data plane. For other tasks, *FP4* leverages the SDT's control-plane CPU to implement more complex behaviors that improves its choice of inputs. In total, the SDT generates test packets, mutates them, sends them to the SUT, and checks for violations/coverage after packets return from the SUT. Figure 5.5 illustrates this lifecycle.

5.5.1. Packet Generation

To generate the input packets for the target at line rate, *FP4*'s SDT uses the built-in hardware packet generation capabilities of the Tofino and similar switches. Generated

```
1  action add_ethernet_ipv4_header() {
2    add_header(ethernet);
3    add_header(ipv4);
4    add_header(ethernet_original);
5    add_header(ipv4_original);
6    modify_field(ethernet.etherType, 0x0800);
7  }
8  action add_ethernet_ipv4_content(ethdstAddr, ...) {
9    modify_field(ethernet.dstAddr, ethdstAddr);
10   modify_field(ethernet.srcAddr, ethsrcAddr);
11   modify_field(ipv4.version, ipv4version);
12   ...
13 }
```

Figure 5.6: Actions that are used to create an Ethernet+IPv4 packet from an existing seed. These correspond to the 'Add headers' and 'Set field values' steps of Figure 5.5. For the example target of Figure 5.4, a separate set of actions would be synthesized for creating Ethernet-only packets.

packets contain an `fp4_visited` header (depicted in Figure 5.3) with all fields initialized to 0s. The generated packets are then passed through automatically synthesized tables that transform the zeroed packet into one of a limited set of seed packets.. The tables first select a random seed number. Based on that seed, the SDT will add a set of headers to the packet and fill them in with an initial value corresponding to one of the configured seed packets. Figure 5.6 provides a snippet of the synthesized actions that *FP4* uses to transform the generated packets.

When *FP4* is first executed, it only contains few seed packets based on the program in SUT; more are added during runtime (see Section 5.5.3).

Deriving expected packet formats The first step in any P4 program is the parser, which takes a sequence of bits from the MAC layer and parses it into its constituent headers. Packets at this stage must adhere to strict formats—all others are dropped before reaching the ingress pipeline of the switch. When generating packets, *FP4* ensures that all seeds (whether from the initial set or added later) pass this stage using a generator-based fuzzing approach.

More specifically, we note that P4 parsers are structured as state machines. The parser transitions between different states depending on the contents of the packet; a packet is only fed into the packet processing pipeline if it reaches an accept state in the parser. *FP4* analyzes the state machine to find all paths from the start to any terminal state; it records the headers extracted in each path along with any field and header contents that triggered the path.

FP4 synthesizes implementations of the above seed-packet generation tables so that it is able to configure seed packet contents from the control plane. Adding a seed is as simple as inserting a table rule to the above tables.

Computing an initial set of seeds *FP4* also uses information from the parser to compute the initial set of seeds. Specifically, it pre-computes one seed packet for each unique parser path, setting the content of the seed packet header to specific values that are part of transitions in the state machine[5]All other fields in the seed header that are *not* constrained by the parser state machine transitions are randomly populated.

As an example, consider Figure 5.4 and its synthesized actions in Figure 5.6. The parser for this program always extracts an Ethernet header, and then only extracts an IPv4 header if the contents of the EtherType are "0x0800." Its state machine, therefore, consists of three states: the start state (not shown), a state to parse the Ethernet header, and a state to parse the IPv4 header. Further, there are only two unique paths through this state machine that lead to accepting states: (1) an L2 frame with only an Ethernet header and (2) an L3 packet with both Ethernet and IPv4 headers. During initialization of the SDT, *FP4* discovers these two paths and randomly generates two initial seed packets that will trigger these parser paths.

[5]Note that, like other implementations of the P4 compiler, *FP4* limits parser recursion to a specified depth, ensuring finite seed packets.

5.5.2. Mutating the Generated Packets

With the seed packet in hand, *FP4*'s goal is then to use the packet to expose new switch behavior. A simple straw man approach to mutating packets would be to randomly select a subset of the header fields and set them to random values. While such an approach will *eventually* catch any bug, blindly mutating packets may only rarely result in inputs that traverse new control flow or trigger new table actions. Instead, *FP4* takes a more targeted approach using one of three techniques per packet (with configurable probability of each decision).

(1) Targeting a specific *table entry* or *conditional statement* Fundamentally, code coverage is determined by the actions triggered in the SUT. For a packet to trigger a given action, its headers and metadata must match the set of 'keys' in a table entry that is mapped to the target action. *FP4* takes advantage of the fact that the current set of table entries are known precisely at runtime to implement a 'magic value' approach to fuzzing [150, 116].

More specifically, the SDT contains a 'mutation' table with an action corresponding to each match-action table and conditional statement in the SUT. In the case of a SUT table, the action takes its match fields as parameters; thus, whenever an entry is added to a SUT table, *FP4* can attempt to add a corresponding entry to the mutation table with the match-key constants passed as parameters to the table-specific action. LPM keys are converted to their base value; 'do not care' bits of ternary matches are converted to zeros. In the case of a conditional, the action takes any referenced fields as parameters, and *FP4* tries to add a corresponding entry based on static analysis of the program.

When target values are sparse and directly dependent on the input packet header, this technique can greatly speed coverage. For example, consider an ingress MAC

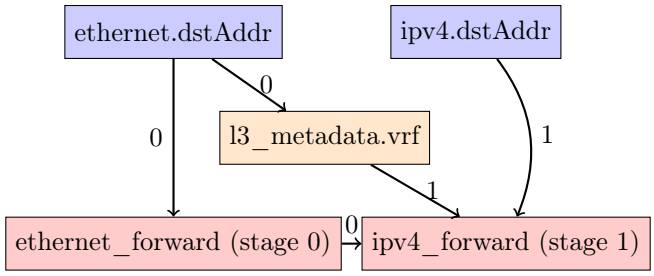


Figure 5.7: Example dependency graph for the example of Figure 5.4. Arrows indicate a "depends on" relation where the source node affects the computation of the target node. Tables are marked with their stage, and arrows are labelled with the earliest stage with the dependency. A table depends on a field if there is a path from the field to the table with only edges that have labels less than or equal to the table's stage.

filter that only matches the interface and broadcast MAC addresses, two values out of 2^{48}. Even at ∼2 billion packets per second, triggering the action would take an average of ∼20 hours.

(2) Targeting a *table* or *if statement* Note that not all matched fields are directly configurable. Examples include tables that match on metadata or on fields that are changed in previous stages of the pipeline. For these cases, *FP4* takes advantage of the P4 program's structure to preferentially mutate a set of fields together if it knows these fields are more likely to result in "hitting" a new table entry or conditional branch, thus dramatically shrinking the search space of mutations.

But how does *FP4* decide what groups of fields should be mutated together? To determine this, *FP4* makes use of a lightweight, stage-sensitive static analysis over the program control flow graph (CFG). The analysis extends traditional flow-insensitive static analysis techniques [132] to be sensitive to packet modifications occurring at different stages. In particular, *FP4* statically analyzes the input program to create a dependency graph that captures whether each packet field "could be" relevant for a given table lookup. The graph for Figure 5.4 is shown in Figure 5.7. It contains nodes for each of the packet's header and metadata fields as well as for each table appearing in the program.

In P4, each table is associated with a stage number. *FP4* labels the table nodes with that stage number and adds an arrow between two nodes when there is a dependency between these components. For example, if `on_l3_hit()` modifies the `egress_port` of the packet, the `egress_port` field would depend on the content of both the `ipv4.dstAddr` and a `vrf` metadata field. The `vrf` field may, in turn, depend on `ethernet.dstAddr` if it is modified in an action of the `ethernet_forward` table. *FP4* adds arrows for fields that are used as keys in tables as well as between tables whose actions may influence the future lookup of other tables. It also adds dependency edges when fields are used in conditionals or updates of stateful ALUs. In most cases, there is only one edge between tables; however, when an `if` statement immediately follows a table lookup, there can be more than one dependency edge as there are multiple possible next tables.

Each dependency edge is labelled with the earliest stage in which the dependency occurs. The mutation procedure is, thus, as follows: For each table in the graph, *FP4* precomputes the set of all packet fields that can impact that table. These fields are all those that can reach the table node in the dependency graph using only edges with stage labels less than or equal to the table stage. At runtime, for each seed packet, *FP4* stores the list of tables that this seed might be able to mutate given the fields present in the seed (which might be a subset of all possible fields). During mutation, after selecting seed packet, it picks one table from the list and preferentially mutate together fields that are keys of that table.

(3) Targeting *stateful counters* Finally, while *FP4* does not explicitly track data-plane state (for the reasons in Section 5.4.2), *FP4* can still discover bugs that depend on stateful behavior. The table-targeted mutation technique, for instance, will properly track the dependencies of stateful registers and modify them if the register outputs are useful for increasing coverage or testing assertions (it will not attempt

to mutate state that is write-only). There are, however, edge cases where *FP4*'s lack of visibility into data-plane state changes may impede its efficiency. For example, consider a target program that counts the number of packets for each 5-tuple and triggers an assertion violation if any counter exceeds a threshold. *FP4* will quickly cover the action with the stateful counter, and it will eventually trigger the violation (after sufficient random collisions), but may do so slowly. Noticing this tendency toward counters in network programs, *FP4* will occasionally repeat packets to ensure that the most common classes of stateful behaviors are captured quickly.

Chaining mutations Note that, particularly for (1) and (2), it may be advantageous to chain mutations to trigger matches that are impossible with only one mutation of existing seeds. *FP4* can pack a few such mutations within a single pipeline. It can also optionally recirculate the packet to apply even more rounds of mutations, albeit at the cost of throughput.

5.5.3. Evaluating Assertions and Coverage

At the end of pipeline, *FP4* makes a copy of all headers to reserved `*_original` headers. If the packet is later determined to be seed-worthy, the cloned headers serve as a record of the original input packet, prior to any SUT modifications.

Two types of packets will return from the SUT: test packets that have traversed the SUT data plane and state-change notifications from the SUT control plane. It may also receive packets generated at the SUT and destined for remote devices (e.g., a periodic control plane routing keepalive message), but these never result in an addition of seed packet (as they are not the result of an SDT input).

For packets from the SUT, the header will contain the visited action bitstring and assertion failure flags along with the original header. The packet has attained additional coverage iff its visited bitstring is novel, i.e., it visited a unique sequence of

actions or path. Because the string can potentially be large, *FP4* tracks uniqueness with the help of a bloom filter. On a filter miss or a set assertion flag, *FP4* sends the packet and its original header to the SDT control plane for further processing. Otherwise, *FP4* recycles the packet by removing all the headers; the recycled packets are treated as a freshly generated packet. Packets from the SUT control plane are always sent to the SDT control plane for addition to the seed packet set. The SUT control plane should have already verified its uniqueness.

5.6. Implementation

We implemented a prototype of *FP4*, including the SUT instrumentation and SDT data plane and control plane agent. Our hardware testbed consists of two Barefoot Wedge100BF-32X programmable switches with all ports on both switches connected by an array of 100 GbE DAC cables. Our implementation currently uses a single line card on each switch.

SUT Instrumentation The *FP4* instrumentation adds the required changes to the input P4 program stated in Section 5.4 to generate an instrumented P4 program. In total, instrumentation implementation comprises around 5500 lines of C++ code, with 4300 lines for a frontend to parse input P4 code using Flex/Bison and build an AST of the input program and 1200 lines to instrument the target SUT program.

The SUT control plane scaffolding currently requires that the programmer annotate their code as described in Section 5.4.2.2 and adhere to a general entrypoint signature.

SDT implementation Our prototype SDT data plane implements the pipeline and functionality detailed in Section 5.5. The code to synthesize the program for SDT incorporates an additional 4200 lines of code. The program synthesis code also outputs a json file to be used by the SDT control plane. This json file contains the structure of packets so the control-plane can parse the incoming packets.

Program	LoC	Actions	Stateful ALUs	Control Plane	Coverage (s)
Load Balancer	159	4	0	Static	0.79 (100%)
Basic Routing	165	6	0	Static	54.33 (100%)
Rate Limiter	197	7	3	Static	8.53 (100%)
Firewall	313	12	4	Static	1.66 (100%)
Netchain [93]	264	6	2	Static	1.11 (100%)
Mirroring	213	7	4	Static	0.24 (100%)
DV Router	284	11	0	Dynamic	39.14 (93%)

Table 5.2: Features and coverage of P4 programs we evaluated

Hardware limitations in the per-stage random number generator restrict the size of mutations to 32-bits, but we find that table-entry-targeted mutations are sufficient to fill this gap. Our prototype SDT control plane takes in the json file generated during instrumentation, adds initial seed packets, parses packets coming from the data plane, track coverage and installs new seed packets in the SDT data plane. The Python control plane is more than 1200 lines of code.

5.7. Evaluation

To evaluate the performance of *FP4*, we conducted experiments on a diverse collection of programs that vary in size and complexity. Rather than merely reaching a particular behavior (such as an assertion violation or invalid header access on a particular line), we focus on the more holistic problem of achieving full *coverage* of all actions and paths in P4 programs, which can be combined with assertions to catch specific bugs. As such, our evaluation aims to address the following questions: (1) How quickly does *FP4* achieve 100% code coverage, compared to existing tools for software-based fuzz testing? (2) Which factors of *FP4*s design have the biggest impact on coverage, and how does its performance compare to more naïve baselines? (3) What is the performance overhead of the added *FP4* instrumentation on the switch under test? (4) Finally, can *FP4* be used successfully to find bugs in existing P4 programs—with a particular eye to bugs that could not be caught with static verification techniques?

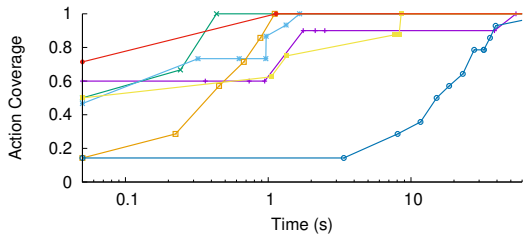
Programs tested Table 5.2 lists tested programs. Load Balancer, Basic Routing, and Rate Limiter represent programs designed primarily for packet forwarding. The Load Balancer makes use of hashing, while Rate Limiter tests Stateful ALUs. Firewall represents a more complex application using Bloom filtering. Netchain also uses concurrency control. In order to avoid false positives due to invalid table rules inserted by the control plane [59], for these examples, we employ a static controller which inserts a fixed number of table rules.

We also evaluate an additional two programs, Mirroring and DV Router. The first program contains the bug mentioned in the introduction: our goal is to determine whether we can catch the bug automatically with fuzzing. The second program is included to test the behavior of switch together with a dynamic, stateful control plane. This example involves additional instrumentation in the control plane, as overviewed in Section 5.4.2.2. Both examples include components that cannot be handled by static verifiers: a hardware-only bug in the first case, and a dynamic stateful control plane in the second case.

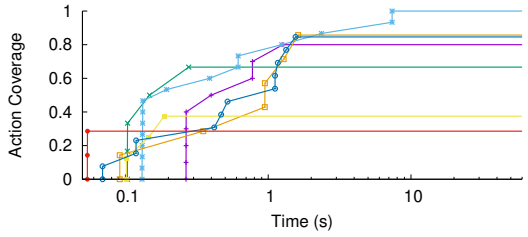
5.7.1. *FP4* Covers Code Quickly

We begin by evaluating the speed at which *FP4* can provide code coverage and test programmable switches. We ran *FP4* over all of the programs described in Table 5.2 on the setup described in Section 5.6, and we log every time a packet arrives at the SDT control plane with a newly covered path through the P4 program or state.

Figure 5.8a shows the speed at which *FP4* triggers all actions in the target programs. The y-axis is the fraction of unique actions triggered divided by the total number of unique table-action pairs in the programs. We note that, in all but one of our test programs, *FP4* provides complete coverage for all the programs within around 1 min. This is true even for programs with only a single entry in a table with a wide keyspace. The only exception was DV Router, where an action hit depends on (1) a properly

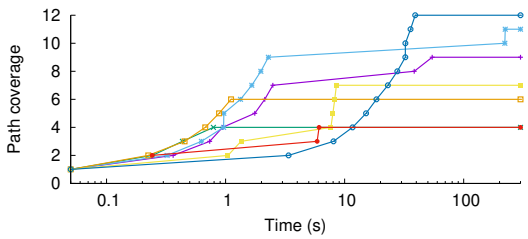


(a) *FP4*

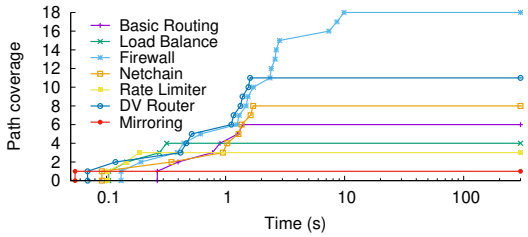


(b) p4pktgen (cherry-picked table rules)

Figure 5.8: Action coverage over time for *FP4* and p4pktgen. Coverage is normalized to the total number of actions in the program. When comparing (a) and (b), we caution readers to consider the differences laid out in Section 5.7.1



(a) *FP4*



(b) p4pktgen (cherry-picked table rules)

Figure 5.9: Path coverage over time for *FP4* and p4pktgen over up to a 5 min trace. When comparing ((a)) and ((b)), we caution readers to consider the differences laid out in Section 5.7.1.

formatted incoming routing update followed by (2) a matching ARP response for the next-hop router and (3) a packet that hits the target routing table entry. While *FP4* can eventually trigger this sequence of events, step (2) is currently improbable as it relies on a metadata field `nextHop` whose dependencies cross the DP/CP boundary. Figure 5.9a also shows results from the same run for paths through the program, defined as a either a unique sequence of triggered actions or a change in the control-plane state.

We also show results for another open-source P4 fuzzer, p4pktgen [140]. We note an important difference between the experiments: p4pktgen uses symbolic execution to solve for a small number of input packets and assumes it can freely configure the control plane rules when doing so. As such, some of the inputs/configurations are not actually achievable in real networks with real control planes. In fact, we needed to remove all `default_action` statements from the test programs (we modified p4pktgen

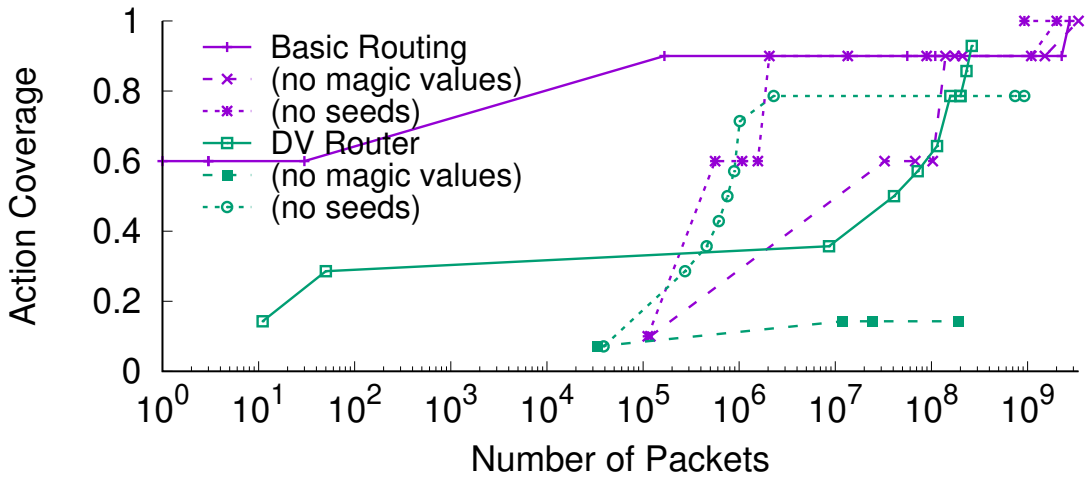


Figure 5.10: Packet efficiency (i.e., actions covered, relative to total, per test packet) with and without *FP4*'s optimizations.

so that it would stop considering impossible `NoAction` actions).

In contrast, *FP4* takes the deployed program and its existing table entries and configurations. Despite this advantage, Figures 5.8b and 5.9b demonstrate that p4pktgen still struggles to achieve full action coverage in real programs. While they can achieve some coverage very quickly, limitations in its current language support (e.g., statefulness and egress logic) mean that some paths are never solvable.

Note that the higher path coverage of p4pktgen in Figure 5.9b is due it taking impossible paths. For example, `Firewall` contains a IPv4 lookup followed by per-port map lookup. p4pktgen finds paths that include an IPv4 miss (where the packet is dropped) followed by a map hit on a non-existent port. The static table rules do not allow *FP4* to take this path.

5.7.2. Factor Analysis

To evaluate the benefits of different features of *FP4*, we evaluate the packet efficiency of *FP4* in covering the P4 program over a 5 min period. Figure 5.10 shows these results for (1) the full *FP4* implementation, (2) *FP4* without magic values (the mutations of

Program	Stages	Tables	SRAM (KB)	Metadata (b)
Load Balancer	2→3 (+1)	4→6 (+2)	144→160 (+16)	948→1056 (+108)
Basic Routing	6→8 (+2)	9→11 (+2)	2080→2096 (+16)	647 → 747 (+70)
Rate Limiter	5→6 (+1)	7→11 (+4)	128→144 (+16)	755→879 (+124)
Firewall	7→8 (+1)	12→18 (+6)	160→176 (+16)	1043→1178 (+135)
Netchain [93]	3→6 (+3)	6→10 (+4)	544→560 (+16)	1084→1203 (+119)
Mirroring	1→4 (+3)	7→13 (+6)	128→128 (+0)	651→781 (+130)
DV Router	3→4 (+1)	11→15 (+4)	352→400 (+48)	949→1104 (+155)

Table 5.3: Resource overhead of P4 programs we evaluated

Section 5.5.2 that are targeted for specific table entries or conditional statements), and (3) *FP4* without coverage-guided fuzzing (and only the parser-based seed generation). We show values for two programs that illustrate different effects. Basic Routing benefits from magic values to quickly cover most actions, but hitting the remaining actions relies solely on the long tail of randomness. DV Router, on the other hand, is aided by *FP4*'s optimizations in both its initial (first 100 packets) and final coverage.

5.7.3. Overhead of *FP4*

FP4's primary overheads stem from the instrumentation it adds to the SUT in order to gain enough visibility to implement its greybox fuzzing approach. We note that the resource consumption of the SDT is less important as the program is not co-resident with any other programs. Rather, Table 5.3 shows the key resources required on the SUT, which are generally low across all tested programs. The overhead does mean that not all programs are amenable to *FP4*'s greybox approach. An exploration of which of *FP4*'s features can be relaxed to address programs that are already close to exhausting all resources is out of the scope of this work.

5.7.4. Case Studies

We now present our experiences testing and finding issues in real programs using *FP4*.

```
1 table tiMirror {
2   reads { ethernet.etherType : exact; }
3   actions { aiMirror; }
4   default_action: aiMirror();
5 }
```

Figure 5.11: A snippet of a table that triggers the mirroring bug.

5.7.4.1 Mirroring Bug

As previously mentioned, during the development of *FP4*, we stumbled on a bug in the interaction between mirroring and default actions. Specifically, given a table such as Figure 5.11, if the action `aiMirror()` configures the packet to be mirrored, the P4 specification would suggest that the packet should be mirrored even if the user does not add any entries beyond the default entry. In reality, while other primitive actions in `aiMirror()` execute properly, we find that packet is not multicasted. Only after adding a non-default entry does the multicast properly apply. After adding an assertion to the input program (Mirroring), *FP4* finds the path that violates the assertion in under 1 seconds.

While this bug was identified and addressed in more recent versions of the switch SDE, static verification tools that only consider the P4 program itself would not have found any faults in this program. Further, we note that the simulators provided as part of the behavioral model and the switch SDE also do not catch this bug. Only when this program is deployed to a hardware switch will this issue manifest.

5.7.4.2 Testing a Distance Vector Router

We also use *FP4* to test for issues in a device in which the P4 data plane and its control plane interact to implement a distance-vector powered IPv4 router. The router provides the basic functionality required for it to be placed in an arbitrary network and learn its surroundings. Thus, the router's functionality includes the ability to handle ARP requests and responses, to understand Ethernet forwarding,

and to execute a distance vector protocol to determine the correct set of LPM routing table entries. Like a real router, the data plane is designed to be general to the topology, port counts, and address assignments of the network; instead, the control plane will configure all tables based on a provided interface configuration and data learned from neighbors or passing packets.

The original purpose of this router is as a teaching tool: students are given some skeleton code and are expected to fill in the P4 and control-plane logic for the above protocols. Testing and debugging students' implementations is a critical task. While testing the end-to-end correctness of the implementation is straightforward (e.g., by deploying a set of their routers and connecting two Linux hosts to either side), it is often also useful to test for violations of basic invariants, which can serve as both sanity checks and a method to localize errors. Because of the tight integration of the control and data plane, static verification tools are not sufficient and prone to false positives. In particular, there are many invariants where one set of table rules may result in correct behavior and another that result in errors. For example:

- Testing that the router only responds to L2 frames destined for the local interface's MAC address or the broadcast address.
- Testing that outgoing packets are all filled with a sender MAC address corresponding to the egress port.
- Testing that outgoing ARP responses match the incoming requests (e.g., the correct operation code and sender/target hardware and protocol addresses.

Further, the target includes routing packets that are generated by the onboard control plane, and while the data plane *could* produce errors depending on the contents of the generated packets, they *will not* because of the correctness of the control plane. Example of such properties include:

- Testing that outgoing distance-vector updates include the correct source IP.
- Testing that the outgoing distance-vector updates are properly formatted, e.g., the advertised cost is less than some preconfigured 'small infinity' so as to mitigate the count-to-infinity problem of distance vector protocols.

In all cases, *FP4* enables expressive testing of the complete programmable switch that only finds bugs that are feasible.

5.8. Limitations and Future Work

In principle, by virtue of randomness, *FP4* is able to eventually able to trigger any possible data plane bug except those that involve unused fields or that only manifest when incoming packets are rare. One could also cover those types of bugs, e.g., if *FP4* had support for dropping a random number of seed packets, but doing so would greatly sacrifice throughput.

Looking forward, we note that there are significant opportunities for extending *FP4* to incorporate more recent advancements in the field of fuzzing, and/or to more efficiently cover several classes of bugs that are currently inefficient for *FP4* to catch. The space of possible optimizations is infinite, but promising directions include: coordinated time stamp emulation in the data plane and control plane to better handle timing-triggered bugs, pruning of old or useless seeds to improve the efficiency of exploration, prioritization of existing seeds to target known gaps in the program coverage, gating seed packets based on the current system state to guarantee uniqueness of seed coverage, and tracking control plane code coverage to more efficiently explore bugs that originate there.

5.9. Summary

In this chapter, we present *FP4*, the first line-rate greybox fuzzing framework for P4 programmable switches. *FP4* adapts several carefully selected, time-tested ideas from

the realm of traditional application fuzz testing and demonstrates how to adapt the ideas to programmable switches—with the switches serving as both the target and the fuzzer. Our evaluation demonstrates that *FP4* can quickly find bugs in programs, even if the bugs are not in the P4 program itself (e.g., in the case of compiler and control-plane bugs).

CHAPTER 6

Related Work

This chapter summarizes areas of research that are related to the problems discussed and systems introduced by this thesis.

6.1. Network Measurements

Network measurement is a well-studied field, with many proposals for better and more expressive measurement tools [134, 196, 77, 104]. As networks grow, it becomes even more important to have good monitoring and debugging tools. Our work is, to the best of our knowledge, the first to demonstrate practical, synchronous, and consistent network-wide measurement with analysis. A large body of prior work has tackled related goals and solutions. We discuss that work below.

Hardware-assisted measurement. With the recent rise of programmable data and control planes, there has been increased interest in novel measurement applications [134, 149, 184, 123, 117, 169]. Thus far, these approaches have concentrated on exploring the limits of what can be feasibly collected. Together, they are a testament to the expressiveness and utility of programmable switches. Our work is complementary—network snapshots can be of *any* local state, including the statistics generated by these systems.

Multi-device measurement. One method to move beyond single-component measurement is to leverage traffic to capture relevant state as it traverses the network [16, 77, 20, 104]. For example, packets could record the minimum queue depth at any intermediate switch. These techniques have the advantage of enforcing causal consistency at the level of an single sample; however, like single component measurement, it is

still difficult to compare across samples and paths.

Distributed snapshots. The literature on distributed snapshot algorithms is similarly rich. The original paper on the topic [45] inspired a wide variety of improvements and refinements. Of particular note are piggybacking-based protocols like [108, 114]. Originally designed to allow for non-FIFO channels, Speedlight borrow their techniques for handling packet drops, but prohibit out-of-order delivery for efficiency. Finally, we note that others have discussed the practicality of distributed snapshots in networks [96, 160], but in the control plane rather than the data plane.

6.2. Network Analysis

Measurement aggregation. An approach for trying to understand network-wide behavior is to take measurements of individual devices or paths and build larger insights on top of their aggregates. There are too many such approaches to cover here, but these largely rely on statistics, thresholds, and similar techniques. Network tomography [113, 133, 100, 47] is a common example that uses statistics to tease out interesting behavior from long-term traces of multiple devices. While this class of approaches can assist in a variety of use cases, they lack the granularity to answer the questions related to network-wide behavior.

Traffic pattern inference. We note that the concept of a network traffic pattern is not novel. Many prior works have both identified and used traffic patterns to great benefit [48, 78, 112, 157, 193, 154]. Unfortunately, these insights have typically been limited to situations where the pattern can be measured at a single link/device [194, 112, 157, 193] or have been a result of property-specific analyses, often with a large dose of manual effort [78, 48, 154, 36]. The goal of `tpprof` is instead the automatic extraction and ranking of common patterns from running networks.

Network visualization tools. We also acknowledge the vast array of existing

network monitoring and visualization tools, both commercial [149, 138, 70, 98, 1, 2, 3, 4, 5] and academic [194, 131, 123, 179, 162, 79]. We lack sufficient space to discuss them all, but one worth mentioning is Cisco's recent Tetration platform [149]. Among other features, Tetration can extract the control flow of a distributed application by clustering hosts based on the partners with which they communicate. Other work has attacked similar problems [95]. To the best of our knowledge, `tpprof` is the first tool that extracts common *network-wide traffic patterns*, rather than application-level communication patterns or packet/flow-level behavior. Broadly speaking, `tpprof` operates at a higher-level of abstraction than these existing systems.

Wee note, however, that `tpprof` is compatible with some infrastructure monitoring frameworks like Nagios [98] that collect monitoring data from across the network. By default, none of these provide the same abstraction as `tpprof`, but many allow custom measurement configurations and plugins, of which `tpprof` could be one.

Application performance profilers. `tpprof` draws inspiration from a long history of work in application performance profiling [75, 171, 153, 43, 50, 186, 130]. Some of which are even able to profile distributed applications [86, 118, 141, 85]. While `tpprof` borrows its approach from the subset of these that profile stochastically, it does this for traffic patterns, which have their own unique set of challenges.

Anomaly detectors. `tpprof` alerting mechanisms are related to prior work in anomaly detection. Compared to unsupervised anomaly detection [41, 191], however, `tpprof` provides a much more accurate and fine-grained detection method. Compared to traditional profiling-based anomaly detection in which a user provides a 'correct' trace and the system determines whether the current system diverges [109, 167], `tpprof` can distinguish between different anomalies and does not require the user to obtain a correct trace. More generally, `tpprof`'s scoring engine presents a natural, declarative interface for the user tell the detector, via traffic pattern signatures, the

approximate characteristics of relevant traffic patterns.

Clustering and compression. Finally, we note that our techniques for compressing network states borrow from or are related to the rich literature on clustering and compression [66, 53, 34, 91]. Our network state extraction techniques, in particular, leverage existing algorithms. The contribution of this work is instead the choice and tuning of these clustering algorithms to the domain of network traffic pattern analysis.

6.3. Network Verification

Network verification is an extensively studied field. Verification tools such a HSA [101] and Anteater [128] verify the correctness of a static snapshot of network forwarding tables. Later tools such as Veriflow [102] perform runtime verification by constantly re-verifying the network state as changes occur. Each of these tools reasons about all packet behaviors—a challenging task—however, their reasoning is limited to verification of *stateless* network forwarding. This dissertation is focused on complex temporal, stateful properties of general-purpose distributed NFs and full stack testing for programmable switches.

Runtime verification Researchers have studied runtime verification extensively, with many papers dedicated to improving its expressiveness and performance. We find that, unfortunately, these existing systems are a poor fit for our setting. For example, D^3S [122] is a runtime verifier. Like *Aragog*, it focuses on identifying bugs in distributed systems at runtime, and its usage of C++ implementations to specify general-purpose properties means that it can check a wider range of properties than *Aragog*. On the other hand, *Aragog* is able to leverage its domain-specific IV specification language (based on regular expressions) to reduce overhead (e.g., with event suppression). Similarly, while CrystalBall [177] can proactively steer a distributed system away from bad states, it imposes restrictions on the target system's architec-

ture that make sense for a distributed system, but not necessarily for a large-scale NF. A third system, Pivot Tracing [127] tracks only causal relationships and not unrelated events at different machines—a property required by some of NATGW's uniqueness invariants. We emphasize that none of the above implies strict superiority. In particular, as *Aragog* is domain-customized for NFs, it should not be used for more general cases (e.g., it may not be able to verify systems like Chord or Paxos efficiently).

We also note that *Aragog* borrows ideas from two areas within runtime verification. The first is verification of distributed systems, which is broadly separated into two categories based on whether the system assumes a synchronized global clock [64]. In this respect, *Aragog* would be considered a *decentralized* [64, 58] runtime verification system. The second is parametric verification, which focuses on checking universally or existentially quantified expressions [80, 151, 54, 126]. The location variables in *Aragog* are examples of parametric variables. The main distinction of *Aragog* from these systems is its combination of parametric and decentralized runtime verification through its support for location variables. Moreover, *Aragog*'s efficient implementation of this combination of features through its use of sharding and local symbolic state machine partitioning is new in this context.

Static verification of NFs and distributed systems Static verification has as equally rich history, including in the domain of NFs and distributed systems [192, 144, 22, 188]. Static verification approaches may provide exhaustive guarantees of correctness, but often suffer from issues of scalability. For this reason, many static verifiers (e.g., [192, 188]) assume single-machine middleboxes, while others (e.g., [125, 103, 178]) may require checking an exponential number of states/paths. Leveraging hand-written NF models can improve scalability compared to verifying source code, but requires tedious and error-prone manual translation of NF models and divorces the verifier from the behavior of the actual deployed system [144, 22].

Aragog makes a different set of tradeoffs, opting to sacrifice principled exploration for improved scalability and giving up the ability to catch bugs early for the ability to test real implementations running over live data. We argue that these tradeoffs are a better fit for our operators' requirements.

Related to the above approaches is the use of semi-automated theorem provers such as Dafny [111]. Users can apply these tools to build systems that are provably correct. A good example of this approach is IronFleet [81], which was used to build a verified, Paxos-based replicated-state-machine library. On the other hand, a drawback of this approach is that it requires significant development effort. IronFleet verification, for example, involved tens of thousands of lines of proof. In contrast, *Aragog* aims to be a lightweight (but sans proof-of-correctness) alternative, requiring little to no developer effort by catching bugs at run time.

Static P4 verification Static verification offers an alternative approach to finding bugs in P4 programs. There is a long line of prior work on applying static verification techniques such as symbolic execution [60, 139, 172, 119, 124] and model checking [173] to P4. Symbolic execution techniques can be categorized based on the correctness specification; for example, netdiff [60] uses differential testing [129, 76] to define correctness, and Assert-P4 [65] uses an expressive assertion language. Systems like bf4 [59] combine static verification with additional runtime checks to ensure properties that it can not verify statically.

Static verification offers strong guarantees regarding completeness—all possible input packets are checked for correctness—but it is not a panacea. Static verifiers can not yet (1) prove properties about *stateful* programs that would involve reasoning about potentially arbitrarily large *sequences* of packets, and they can not (2) find bugs in the switch hardware, (3) P4 compiler or (4) the control plane. Finally, because these tools lack visibility into the control plane, they may (5) report false positives [59] by

conservatively overapproximating the control plane's actual behavior.

We view *FP4* as being complementary to static verification tools. It can test a switch program *in vivo* but provides no coverage guarantees. To improve coverage in practice, *FP4* leverages a combination of (1) a design based on leveraging high-throughput programmable switches, (2) P4 instrumentation for coverage guided feedback, and (3) a bevy of other optimizations (See §5.4 and §5.5).

6.4. Fuzz Testing Programmable Switches

Prior to *FP4*, p4pktgen [140], P4RL [164], and P6 [163] were the first to propose fuzzing for P4 dataplanes. These tools demonstrated that P4 fuzz testing could be effective at identifying a wide variety of bugs. Going off the past observation from software fuzzing that more tests equals more bugs [42], *FP4* leverages programmable switches to generate test packets up to 6 orders of magnitude faster than the prior P4 fuzzing work that is based on software emulation. Moreover, emulating the P4 switch in software means that most prior works cannot detect bugs that only show up in hardware (an observation also made in [82]) such as the mirroring bug from 5.7.4.1.

Work on P4 fuzzing builds on prior research in automatic test packet generation (ATPG [190]), including tools such as PAZZ [165] and Pronto [195]. *FP4* shares a similar goal to these works (i.e., automatically identifying bugs), but specifically takes advantage of the programmability of P4 to enable efficient greybox (rather than blackbox) testing by tracking the test coverage of packets in the dataplane itself.

P4Fuzz [17] and Gauntlet [156] propose fuzz testing for P4 compilers by generating structured test P4 programs more in the style of traditional compiler testing [180]. These works can identify a variety compiler bugs across a diverse set of P4 programs. In contrast *FP4* focuses more narrowly on rigorously testing a *particular* program in its entirety, including the compiler, hardware, control plane, and data plane.

Stateful fuzz testing Finally, *FP4* falls into a broad category of research in bringing fuzz-testing to complex stateful systems. Popular general-purpose stateful fuzzers include RESTler [31], Ijon [30], Peach [55], BeSTORM [38], and Sulley [23]. Outside of programmable dataplanes, stateful fuzzing has also been successfully applied to stress-test the security of network protocols [147, 148, 67, 135, 32, 15, 174]). Compared to these works, *FP4* can not efficiently restart the switch, and thus opts not to generate sequences of test inputs. Instead it leverages the observation that most P4 programs are meant to be run in a "continuous" mode and naturally reset switch state. *FP4* also collects control plane state change information and incorporates this information to derive new seed inputs.

CHAPTER 7

Conclusion

The size, number and complexity of datacenters are growing. Managing such an extensive network with a diverse set of network policies has become more complicated with the introduction of programmable hardware and distributed network functions. Unfortunately, existing debugging tools are not sufficient to monitor, analyze, or debug modern networks; either they lack visibility in the network, require manual analyses, or cannot check for some properties. These limitations arise from the outdated view of the networks, i.e., that we can look at a single packet or path in isolation.

This thesis calls for a different approach for network debugging: To look at the network as a whole and design tools that look at measuring, understanding, and debugging the network across "space" (multiple devices), and "time". Our key idea is to use both (1) in-network packet processing including programmable data-planes and distributed middleboxes to collect precise measurements and (2) out-of-network processing to coordinate measurement and scale analytics. Looking into specific systems, we show how we can use Speedlight to capture network wide measurements that give us in-network visibility and `tpprof` to automatically analyze the network behavior using the measurements. While Speedlight and `tpprof` expand on monitoring space to analyze network-wide behavior, *Aragog* and *FP4* focus on testing the full stack of network functions and programmable switches. *Aragog* collects measurements from stateful distributed network functions and does runtime verification using symbolic finite automata to alert for violations at the data center scale in real-time. *FP4* is a fuzz-testing framework for P4 switches that achieves high expressiveness, coverage, and scalability.

Overall, this thesis design tools that can reduce bugs present before deployment and builds framework to simplify the debugging process. I hope this thesis would guide admins towards self driving networks. In theory, self-driving networks would be able to detect, locate and find root cause of network problems without human inference without human interference. It would also predict application resource usage and network performance trends to avoid problems that could occur in the future, schedule configuration, and request optimal capacity upgrades

Bibliography

[1] https://www.nutanix.com/products/epoch.

[2] https://www.pluribusnetworks.com/.

[3] https://www.logicmonitor.com/.

[4] https://endace.com.

[5] https://www.bigswitch.com/products/big-monitoring-fabric/.

[6] https://aws.amazon.com/blogs/security/tag/network-monitoring-tools/.

[7] http://docs.libmemcached.org/bin/memaslap.html.

[8] Antlr. https://www.antlr.org/.

[9] Apache Flink: Stateful computations over data streams. https://flink.apache.org/.

[10] Apache Kafka. https://kafka.apache.org/.

[11] Maglev outage. https://status.cloud.google.com/incident/cloud-networking/18013.

[12] NetFilter. http://conntrack-tools.netfilter.org/.

[13] A symbolic automata library. https://github.com/lorisdanto/symbolicautomata.

[14] Z3. https://github.com/Z3Prover/z3.

[15] Humberto J Abdelnur, Radu State, and Olivier Festor. Kif: a stateful sip fuzzer. In *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 47–56, 2007.

[16] Aijay Adams, Petr Lapukhov, and Hongyi Zeng. NetNORAD: Troubleshooting networks via end-to-end probing, 2016. https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/.

[17] Andrei-Alexandru Agape, Madalin Claudiu Danceanu, Rene Rydhof Hansen, and Stefan Schmid. P4fuzz: Compiler fuzzer fordependable programmable dataplanes. In *International Conference on Distributed Computing and Networking 2021*, pages 16–25, 2021.

[18] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[19] Ehab Al-Shaer, Hazem Hamed, Raouf Boutaba, and Masum Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE journal on selected areas in communications*, 23(10):2069–2084, 2005.

[20] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 503–514, New York, NY, USA, 2014. ACM.

[21] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[22] Kalev Alpernas, Roman Manevich, Aurojit Panda, Mooly Sagiv, Scott Shenker, Sharon Shoham, and Yaron Velner. Abstract interpretation of stateful networks, 2017.

[23] Pedram Amini, Aaron Portnoy, and Ryan Sears. Sulley. https://github.com/OpenRCE/sulley, 2014.

[24] Dormando Anatoly Vorobey, Brad Fitzpatrick. Memcached, 2009.

[25] Apache Software Foundation. Giraph, 2012.

[26] Apache Software Foundation. Hadoop, terasort, 2012.

[27] Apache Software Foundation. Hadoop, yarn, 2012.

[28] Apache Software Foundation. Pagerank, graphx, 2014.

[29] Apache Software Foundation. Spark, 2016.

[30] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

[31] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

[32] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: toward a stateful network protocol fuzzer. In *International conference on information security*, pages 343–358. Springer, 2006.

[33] Barefoot. Barefoot tofino. https://www.barefootnetworks.com/technology/, 2017.

[34] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 476–489, New York, NY, USA, 2018. ACM.

[35] Richard E Bellman. *Adaptive control processes: a guided tour*. Princeton university press, 2015.

[36] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, January 2010.

[37] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In Catriel Beeri and Peter Buneman, editors, *Database Theory — ICDT'99*, pages 217–235, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[38] HelpSystems beyondsecurity. Bestorm. https://beyondsecurity.com/solutions/bestorm-dynamic-application-security-testing.html, 2022.

[39] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[40] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[41] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: Identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International*

Conference on Management of Data, SIGMOD '00, page 93–104, New York, NY, USA, 2000. Association for Computing Machinery.

[42] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019.

[43] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[44] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[45] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[46] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[47] Yan Chen, David Bindel, Hanhee Song, and Randy H. Katz. An algebraic approach to practical and scalable overlay network monitoring. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 55–66, New York, NY, USA, 2004. ACM.

[48] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 73–82, New York, NY, USA, 2009. ACM.

[49] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 31–36, New York, NY, USA, 2012. ACM.

[50] W.E. Cohen. Tuning programs with oprofile. *Wide Open Magazine*, 1:53–62, 01 2004.

[51] Christophe Croux and Catherine Dehon. Influence functions of the spearman and kendall correlation measures. *Statistical methods & applications*, 19(4):497–515, 2010.

[52] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification, 29th International Conference (CAV '17)*, July 2017.

[53] William HE Day and Herbert Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of classification*, 1(1):7–24, 1984.

[54] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 341–356. Springer Berlin Heidelberg, 2014.

[55] Christoph Diehl, Michael Eddington, Jesse Schwarzentruber, Tyson Smith, Christian Holler, Mark Goodwin, and Aditya Murray. Peach (by mozilla security). https://github.com/MozillaSecurity/peach, 2016.

[56] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. Petr4: Formal foundations for p4 data planes. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[57] Dormando. mc-crusher, 2016.

[58] M. Ali Dorosty, Fathiyeh Faghih, and Ehsan Khamespanah. Decentralized runtime verification for LTL properties using global clock, 2019.

[59] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. Bf4: Towards bug-free p4 programs. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 571–585, New York, NY, USA, 2020. Association for Computing Machinery.

[60] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Dataplane equivalence and its applications. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 683–698, 2019.

[61] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[62] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, pages 523–535, 2016.

[63] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 339–350, New York, NY, USA, 2010. ACM.

[64] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. *Runtime Verification for Decentralised and Distributed Systems*, pages 176–210. 2018.

[65] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[66] Karl Pearson F.R.S. LIII. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[67] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*, pages 330–347. Springer, 2015.

[68] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, pages 81–94, 2018.

[69] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. Design principles for packet parsers. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 13–24, Washington, D.C., USA, 2013. IEEE.

[70] Gigamon. Security and networking solutions | gigamon, 2018.

[71] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

[72] Google. Oss-fuzz, Jun 2021.

[73] Michal Zalewski (Google). american fuzzy lop (afl), Jul 2020.

[74] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72, 2016.

[75] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[76] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *29th International Conference on Software Engineering (ICSE'07)*, pages 621–631. IEEE, 2007.

[77] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 139–152, New York, NY, USA, 2015. ACM.

[78] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 38–49, New York, NY, USA, 2011. ACM.

[79] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation NSDI 14*, pages 71–85, Seattle, WA, 2014. USENIX Association.

[80] Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zălinescu. *Monitoring Events that Carry Data*, pages 61–102. 2018.

[81] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob Lorch, Bryan Parno, Michael Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving safety and liveness of practical distributed systems. *Communications of the ACM*, 60:83–92, 06 2017.

[82] Stefan Heule, Konstantin Weitz, Waqar Mohsin, Lorenzo Vicisano, , and Amin Vahdat. Leveraging p4 to automatically validate networking switches. https://www.opennetworking.org/wpcontent/uploads/2019/09/2.30pm-Stefan-Heule-P4-Presentation.pdf, 2022.

[83] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[84] Mukesh Hira and LJ Wobker. Improving network monitoring and management with programmable data planes, Sep 2015.

[85] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 95–110, Renton, WA, 2018. USENIX Association.

[86] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed performance monitoring: methods, tools, and applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598, June 1994.

[87] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, RFC Editor, November 2000.

[88] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical PFC deadlock prevention in data center networks. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 451–463, New York, NY, USA, 2017. ACM.

[89] Information Sciences Institute. Transmission Control Protocol. RFC 793, RFC Editor, September 1981.

[90] Intel. https://www.hyperscan.io/.

[91] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651 – 666, 2010. Award winning papers from the 19th International Conference on Pattern Recognition (ICPR).

[92] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mos: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 113–129, Boston, MA, March 2017. USENIX Association.

[93] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain:scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, 2018.

[94] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[95] Yu Jin, Esam Sharafuddin, and Zhi-Li Zhang. Unveiling core network-wide communication patterns through application traffic activity graph decomposition. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, page 49–60, New York, NY, USA, 2009. Association for Computing Machinery.

[96] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus routing: The internet as a distributed system. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, pages 351–364, Berkeley, CA, USA, 2008. USENIX Association.

[97] Prem Jonnalagadda. Disaggregation and programmable forwarding planes. https://barefootnetworks.com/blog/disaggregation-and-programmable-forwarding-planes/, 2017.

[98] David Josephsen. *Building a Monitoring Infrastructure with Nagios*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[99] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur W. Berger. Dynamic load balancing without packet reordering. *Computer Communication Review*, 37(2):51–62, 2007.

[100] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. *ACM SIGCOMM Computer Communication Review*, 39(4):243–254, 2009.

[101] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*, pages 9–9, Berkeley, CA, USA, 2012.

[102] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. *SIGCOMM Comput. Commun. Rev.*, 42(4):467–472, September 2012.

[103] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI '07)*, Cambridge, MA, April 2007.

[104] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *Demo paper at SIGCOMM '15*, 2015.

[105] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[106] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[107] Ajay D Kshemkalyani, Michel Raynal, and Mukesh Singhal. An introduction to snapshot algorithms in distributed computing. *Distributed systems engineering*, 2(4):224, 1995.

[108] Ten H Lai and Tao H Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[109] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, page 201–206, New York, NY, USA, 2004. Association for Computing Machinery.

[110] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 454–467, New York, NY, USA, 2016. ACM.

[111] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, 2010.

[112] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw.*, 2(1):1–15, February 1994.

[113] Ma łgorzata Steinder and Adarshpal S Sethi. A survey of fault localization techniques in computer networks. *Science of computer programming*, 53(2):165–194, 2004.

[114] Hon Fung Li, Thiruvengadam Radhakrishnan, and K. Venkatesh. Global state detection in non-FIFO networks. In *International Conference on Distributed Computing Systems*, ICDCS, pages 364–370, Washington, D.C., USA, 1987. IEEE Computer Society.

[115] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[116] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

[117] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI '16, pages 311–324, Berkeley, CA, USA, 2016. USENIX Association.

[118] Inc. Lightstep. Lightstep [x]pm, 2019.

[119] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 490–503, New York, NY, USA, 2018. Association for Computing Machinery.

[120] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 399–412, Berkeley, CA, USA, 2013. USENIX Association.

[121] Vincent Liu, Danyang Zhuo, Simon Peter, Arvind Krishnamurthy, and Thomas Anderson. Subways: A case for redundant, inexpensive data center edge links. In *Proceedings of the*

11th ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '15, pages 27:1–27:13, New York, NY, USA, 2015. ACM.

[122] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D^{3}S: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*, page 423–437, USA, 2008.

[123] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 101–114, New York, NY, USA, 2016. ACM.

[124] Nuno Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. Automatically verifying reachability and well-formedness in p4 networks. *Technical Report, Tech. Rep*, 2016.

[125] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*, New York, NY, USA, 2019.

[126] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 285–300, 2014.

[127] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic causal monitoring for distributed systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016.

[128] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 290–301, New York, NY, USA, 2011.

[129] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[130] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 13–23, New York, NY, USA, 2005. ACM.

[131] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 129–143, New York, NY, USA, 2016. ACM.

[132] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[133] Radhika Niranjan Mysore, Ratul Mahajan, Amin Vahdat, and George Varghese. Gestalt: Fast, unified fault localization for networked systems. In *USENIX ATC*, pages 255–267, Philadelphia, PA, June 2014. USENIX Association.

[134] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 85–98, New York, NY, USA, 2017. ACM.

[135] Roberto Natella and Van-Thuan Pham. Profuzzbench: a benchmark for stateful protocol fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 662–665, 2021.

[136] Tim Nelson, Nicholas DeMarinis, Timothy Adam Hoff, Rodrigo Fonseca, and Shriram Krishnamurthi. Switches are monitors too! stateful property monitoring as a switch design criterion. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*, page 99–105, New York, NY, USA, 2016.

[137] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[138] Barefoot Networks. Barefoot deep insight – product brief, 2018.

[139] Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. Verification of p4 programs in feasible time using assertions. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 73–85, 2018.

[140] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. P4pktgen: Automated test case generation for p4 programs. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[141] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.

[142] p4lang. ipv6 nd fix and prevent cpu bound packets hitting egress acls (issue #82), Dec 2016.

[143] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA, 2019. Association for Computing Machinery.

[144] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, pages 699–718, Boston, MA, March 2017.

[145] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 207–218, 2013.

[146] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[147] Joshua Pereyda et al. Boofuzz. https://github.com/jtpereyda/boofuzz, 2022.

[148] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465, 2020.

[149] Remi Philippe. Next generation data center flow telemetry. Technical report, Cisco, 2016.

[150] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[151] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. MarQ: Monitoring at runtime with QEA. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 596–610, 2015.

[152] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *;login:, the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[153] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, NT'97, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[154] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM.

[155] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. Passive realtime datacenter fault detection and localization. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI '17, pages 595–612, Berkeley, CA, USA, 2017. USENIX Association.

[156] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. Gauntlet: Finding bugs in compilers for programmable packet processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 683–699, 2020.

[157] Bong K. Ryu and Anwar Elwalid. The importance of long-range dependence of vbr video traffic in atm traffic engineering: Myths and realities. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, pages 3–14, New York, NY, USA, 1996. ACM.

[158] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation, 2020.

[159] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a" kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st international conference on distributed computing systems workshops*, pages 166–171. IEEE, 2011.

[160] Liron Schiff, Michael Borokhovich, and Stefan Schmid. Reclaiming the brain: Useful openflow functions in the data plane. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 7:1–7:7, New York, NY, USA, 2014. ACM.

[161] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Changhoon Kim, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI '17, pages 67–82, Berkeley, CA, USA, 2017. USENIX Association.

[162] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[163] Apoorv Shukla, Kevin Hudemann, Zsolt Vági, Lily Hügerich, Georgios Smaragdakis, Artur Hecker, Stefan Schmid, and Anja Feldmann. Fix with p6: Verifying programmable switches at runtime. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[164] Apoorv Shukla, Kevin Nico Hudemann, Artur Hecker, and Stefan Schmid. Runtime verification of p4 switches with reinforcement learning. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pages 1–7, 2019.

[165] Apoorv Shukla, S Jawad Saidi, Stefan Schmid, Marco Canini, Thomas Zinner, and Anja Feldmann. Toward consistent sdns: A case for network state fuzzing. *IEEE Transactions on Network and Service Management*, 17(2):668–681, 2019.

[166] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In

Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM.

[167] Ningombam Anandshree Singh, Khundrakpam Johnson Singh, and Tanmay De. Distributed denial of service attack detection using naive bayes classifier through info gain feature selection. In *Proceedings of the International Conference on Informatics and Analytics*, ICIA-16, New York, NY, USA, 2016. Association for Computing Machinery.

[168] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 15–28, New York, NY, USA, 2016. ACM.

[169] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 11:1–11:16, New York, NY, USA, 2018. ACM.

[170] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *International Conference on Distributed Computing Systems*, ICDCS, pages 382–388, Washington, D.C., USA, 1986. IEEE Computer Society.

[171] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 196–205, New York, NY, USA, 1994. ACM.

[172] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 518–532, 2018.

[173] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, et al. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 17–32, 2021.

[174] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. Secfuzz: Fuzz-testing security protocols. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 1–7. IEEE, 2012.

[175] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. Opennetmon: Network monitoring in openflow software-defined networks. In *Network Operations and Management Symposium*, NOMS, pages 1–8, Washington, D.C., USA, 2014. IEEE.

[176] Guangming Xing. Minimized thompson NFA. *International Journal of Computer Mathematics*, 81:1097 – 1106, 2004.

[177] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, page 229–244, USA, 2009.

[178] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, page 213–228, USA, 2009.

[179] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 561–575, New York, NY, USA, 2018. ACM.

[180] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[181] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 701–718. USENIX Association, November 2020.

[182] Nofel Yaseen, John Sonchack, and Vincent Liu. Speedlight bmv2, 2018.

[183] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 402–416, New York, NY, USA, 2018. ACM.

[184] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with OpenSketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 29–42, Berkeley, CA, USA, 2013. USENIX Association.

[185] Yifei Yuan, Soo-Jin Moon, Sahil Uppal, Limin Jia, and Vyas Sekar. NetSMC: A custom symbolic model checker for stateful network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 181–200, February 2020.

[186] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the mips r10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[187] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[188] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, page 275–290, New York, NY, USA, 2019.

[189] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified NAT. In *Proceedings of the ACM SIGCOMM 2017 Conference*, page 141–154, 2017.

[190] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252, 2012.

[191] Jiong Zhang and Mohammad Zulkernine. Anomaly based network intrusion detection with unsupervised outlier detection. *2006 IEEE International Conference on Communications*, 5:2388–2393, 2006.

[192] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 221–239, Santa Clara, CA, February 2020.

[193] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 78–85, New York, NY, USA, 2017. ACM.

[194] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pages 101–114, New York, NY, USA, 2004. ACM.

[195] Yu Zhao, Huazhe Wang, Xin Lin, Tingting Yu, and Chen Qian. Pronto: Efficient test packet generation for dynamic network data planes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 13–22. IEEE, 2017.

[196] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 479–491, New York, NY, USA, 2015. ACM.