Novel Abstractions for Data Center Network Management

By

Aaron Gember-Jacobson

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2016

Date of final oral examination: 04/29/2016

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

Aditya Akella, Associate Professor, Computer Sciences
Paul Barford, Professor, Computer Sciences
Parameswaran Ramanathan, Professor, Electrical and Computer Engineering
Jennifer Rexford, Professor, Computer Science, Princeton University
Michael Swift, Associate Professor, Computer Sciences

To Emily and Henry.

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Aditya Akella, for his dedication and support. Many of my achievements as a graduate student would not have been possible without his mentoring.

I would also to thank my peers and collaborators who have contributed their time and insights to the work contained in this thesis: Robert Grandl, Shan-Hsiang Shen, Junaid Khalid, Xiujun Li, Ratul Mahajan, Chaithan Prakash, Raajay Viswanathan, and Wenfei Wu. Similar thanks goes to those I have worked with on other projects throughout my PhD, including: Archie Abhashkumar, Ashok Anand, Theo Benson, Ramon Caceres, Shoban Chandrabose, Mark Coatsworth, Sourav Das, Chris Dragga, Alexis Fisher, Xiaoyang Gao, Keqiang He, Shachar Itzhaky, Aleksandr Karbyshev, Anand Krishnamurthy, Roney Michael, Jeff Pang, Tom Ristenpart, Mooly Sagiv, Vyas Sekar, Saul St. John, Asaf Valadarsky, Alex Varshavsky, and Liang Wang.

I appreciate the valuable feedback provided by my prelim and thesis committee members: Paul Barford, Somesh Jha, Parmesh Ramanathan, Jen Rexford, and Mike Swift.

Finally, I am extremely grateful for my wife Emily, who has provided continual encouragement and support, and my son Henry, who brings joy to my life. Emily and Henry, along with my parents (Lynn and Jim), grandparents (Eloise, Bob, Carol, and Ken), and other family, have all been an important part of my journey.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Data center failures have become increasingly problematic due to the plethora of critical web and storage services hosted in today's data centers. Frequently, the problem lies in the data center network, which is prone to both functional and performance failures caused by hardware or software faults, misconfiguration, overload, or other issues with links and devices.

Preventing such failures is challenging, because data center network operators lack a formal understanding of how their design and operational decisions impact the frequency of network problems. Furthermore, current frameworks for verifying and maintaining the functionality and performance of data center networks are incomplete and/or inefficient. Consequently, this thesis explores how to analyze an organization's network management practices and efficiently guarantee that a data center network functions correctly and offers reasonable performance amidst changes in infrastructure, configuration, and workload.

We first present the design of a management plane analytics (MPA) framework which uncovers the relationships between network management practices and the frequency of network problems. By applying MPA to over 850 data center networks operated by a large online service provider, we identify several practices that strongly impact the frequency of problems in these networks, including: the number of control plane configuration changes and the number of device types (i.e., the presence of middleboxes).

Armed with this information, we explore how to design abstractions that aid in ensuring the correct and performant operation of a data center's control plane and middleboxes. We introduce an abstract representation for control planes that efficiently models a data center network's forwarding behavior under all possible link/device failure scenarios. This allows us to verify important functional invariants—e.g., traffic between subnets S_1 and S_2 *always* traverses a middlebox—three to five orders of magnitude faster than current verification tools. Additionally, we introduce a middlebox state management framework that allows network operators to realize a "one-big-middlebox" abstraction and avoid middlebox-induced functional and performance failures in the presence of hardware/software faults or overload.

Our framework guarantees the safety and consistency of transferred/replicated middlebox state with minimal latency and resource overhead.

1 INTRODUCTION

Tremendous growth in web services, big data analytics, storage-as-a-service, and other resource-intensive workloads has fueled a steady increase in the prevalence, scale, and importance of data centers. Today's data centers host a plethora of critical services that organizations and individuals rely on daily for commerce, transportation, communication, social interaction, and much more.

As our dependence on data centers has increased, so has the impact of data center failures. A 2016 survey of 49 US-based organizations found that the average cost of a data center outage is $740K [29]. Failures in large cloud data centers, such as Amazon EC2, are even more problematic due to the large number of popular web services hosted in these data centers: e.g., over 4% of the Alexa top one million websites are, at least partially, hosted in Amazon EC2 [78].

One of the frequent elements that fails is the data center network [50, 126, 145, 146]. In particular, two types of network failures commonly occur:

- *Functional failures* occur when the network does not forward, filter, monitor, or modify network traffic as expected. Such failures impact reachability, security, and efficiency. For example, when a pair of servers cannot communicate—despite a network operator's intent that such communication be allowed—we say a functional failure has occurred. The inverse (i.e., two servers can communicate despite an operator's intent that such communication be blocked) is also a functional failure. Other examples include security alarms not being raised for malicious traffic that matches known signatures and traffic volumes (used for billing purposes) being over- or underestimated beyond standard tolerances.
- *Performance failures* occur when throughput, latency, loss, or other quality of service measures fall outside acceptable ranges. For example, when the latency between a pair of servers exceeds some threshold (e.g., 100ms), we say a performance failure has

occurred.

These two types of failures may be caused by a variety of low-level issues, including hardware or software faults, configuration errors, overload, or other problems with network links and devices. Several recent studies [75, 113] have examined the frequency of such issues in data center networks. However, little has been done to relate these failures back to the high-level design and operational decisions made by network operators—e.g., the diversity of devices used in the network or the extent to which configuration changes are automated. Consequently, it remains challenging for operators to determine the best ways to manage their data center networks.

Furthermore, many of the tools currently available to network operators for verifying network functionality (e.g., reachability) and maintaining the performance and availability of specialized packet processing elements (i.e., middleboxes) operate at a low-level of abstraction. This leads to inefficiencies that force data center network operators to make undesirable trade-offs. For example, operators must either limit network correctness checks to a small set of possible infrastructure faults (e.g., all single link failures) or wait an extended period of time (days or weeks) while the network's configuration is fully verified. In the case of middleboxes, operators must compromise on performance and availability, resource efficiency, or correctness (e.g., in-progress connections passing through a load balancer may be abruptly terminated); all three cannot be achieved simultaneously using currently available frameworks.

To address these gaps, we focus on answering two important questions in this thesis:

1. How do network operator's design and operational decisions impact the frequency of functional and performance failures in data center networks?
2. How can we design abstractions to efficiently guarantee that a data center network functions correctly and offers reasonable performance amidst changes in infrastructure (e.g., link or device failures), configuration, and workload?

The next two sections (Sections 1.1 and 1.2) describe our goals and the associated challenges in more depth. At the end of the chapter (Section 1.3), we summarize the key

contributions of this thesis.

1.1 Identifying Problematic Network Management Practices

Careful design and operation of a data center network is essential to avoiding functional and performance failures. For example, a network operator's choice of devices—such as the vendor and model of switches, routers, and middleboxes—can impact the frequency of hardware or software faults [75, 113]. Similarly, the design of the network control plane—such as which protocols are used—can impact the complexity of switch and router configurations [40], thus increasing (or decreasing) the likelihood of a configuration error. Both issues may lead to functional or performance failures that negatively impact users and applications.

Unfortunately, network operators and researchers alike currently *lack a systematic understanding of the best practices* for managing data center networks. The design and operational practices network operators employ today are primarily based on operators' experience and vendors' recommendations. Consequently, there is a great diversity of opinion with regards to the impact of specific management practices on the frequency of network problems. Our survey of 51 network operators[1] shows clear consensus for just one practice (Figure 1.1): the number of configuration changes has a high impact on the frequency of network problems. Otherwise, operator perceptions are mixed: e.g., the fraction of operators who said the number of device models and inter-device configuration complexity have low impact is roughly the

[1]We recruited operators from the North American Network Operators Group (NANOG) mailing list (45 operators), the University of Wisconsin-Madison network engineering team (4 operators), and the network operations team of a large online service provider (2 operators).

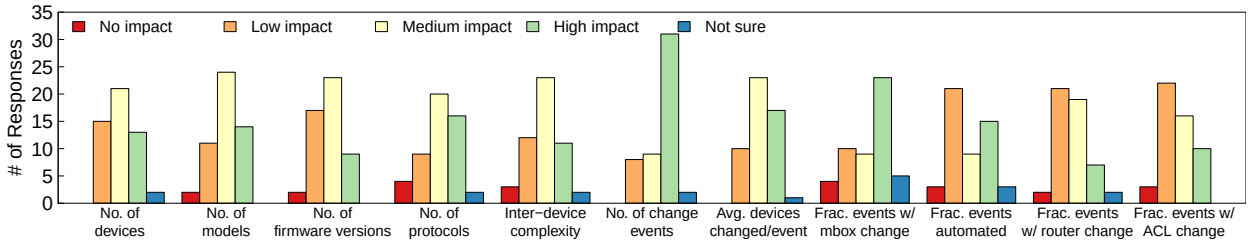


Figure 1.1: Results of network operator survey

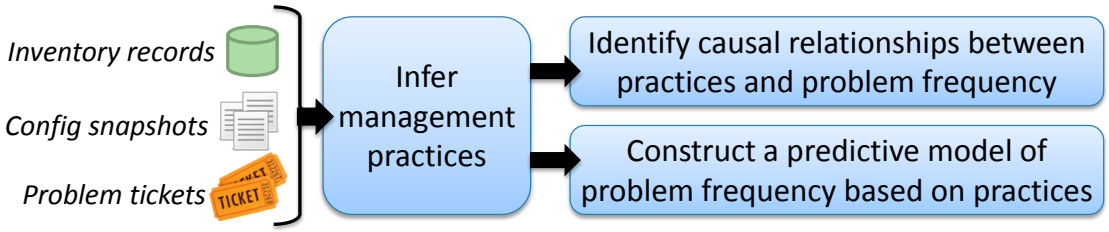


Figure 1.2: Steps in management practice analytics

same as the fraction who said these practices have high impact. Even operators from the same organization have differing opinions.

Although the adage "there's no substitute for experience" may be true when it comes to managing data center networks,[2] we argue that *formally characterizing the impact of network management practices on the frequency of problems in data center networks can help network operators reduce the frequency and severity of functional and performance failures.* However, there are several questions we must answer to enable such an analysis:

- How do we determine what management practices are used in data center networks? Management practices are rarely directly recorded. Even if expected practices are documented, there is no guarantee operators actually follow these practices.
- How do we draw meaningful conclusions from limited data? The number of available snapshots of a data center network's design and operation is often limited, and some snapshots may be missing due to incomplete or inconsistent logging.
- How do we account for skew and overlap in management practices? Skew makes it challenging to accurately identify the impact of less common practices, while overlap makes it challenging to determine the impact of individual practices.

In Chapter 2, we introduce a *management plane analytics* (MPA) framework (Figure 1.2) that addresses these questions and uncovers the relationships between management practices and the frequency of network problems. An organization can apply MPA to its data center networks to: (1) determine which practices cause an increase, or decrease, in the frequency

[2]In our survey, operators wrote "skill levels of operators" and "documentation and training provided" as other practices they believe to have a high impact on the frequency of network problems.

of network problems; and (2) develop a predictive model of problem rate, based on management practices, to aid what-if analysis. The former is achieved using a quasi-experimental design technique known as propensity score matching [135, 136] (Section 2.4), while the latter is achieved using decision tree learning algorithms augmented with boosting [67] and oversampling techniques (Section 2.5). The knowledge gained from MPA can help network operators select and refine the management practices they use in current and future data center networks.

By applying MPA to over 850 data center networks operated by a large online service provider (OSP), we uncover several management practices that strongly influence the number of problems these networks experience, including: network size, the number and type of configuration changes, and the number of device types (e.g., router, switch, firewall, load balancer, etc.). We also find several instances where network operators' perceptions (Figure 1.1) conflict with reality: e.g., the fraction of changes where an access control list (ACL) is modified has a non-trivial impact on the frequency of problems despite a majority opinion that the impact is low.

1.2 Abstractions for Guaranteeing Network Functionality and Performance

The ability to change practices that contribute to data center network failures varies based on the class of practice and the needs of the organization. Adjusting design practices (e.g., the number and type of devices) requires deploying new data centers or significantly overhauling existing data center networks. In some cases, adjustments in design are limited by workload demands. For example, the number of end hosts the network must support places a lower bound on the number of routers and switches it must contain. Similarly, the security and efficiency requirements of applications may dictate that certain types of middleboxes be present in the network. Operational practices can be adjusted more easily: e.g., configuration

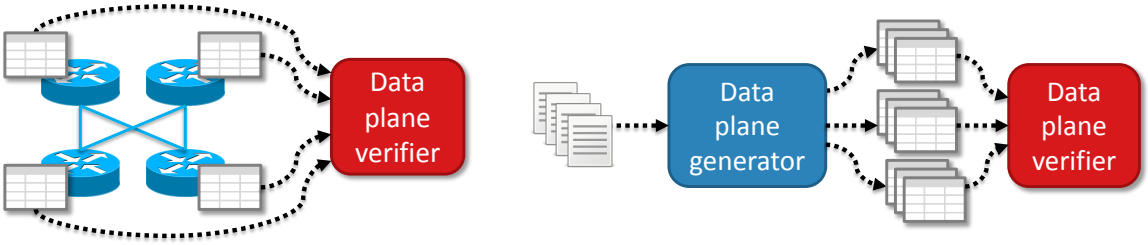
changes can be aggregated or reduced by more carefully planning network reconfigurations. However, such adjustments may have other repercussions: e.g., configuration errors may be more likely when the size and scope of a configuration change is larger. Moreover, even if best practices are followed, problems will inevitably arise: hardware will fail, operators will make mistakes, and operating conditions will change.

Consequently, we believe that adjustments in management practices (informed by MPA) must be accompanied by the introduction of *new frameworks designed to guarantee that a data center network functions correctly and performs well, even amidst changes in infrastructure, configuration, or workload.* In particular, we focus on designing abstractions that aid in ensuring the correct and performant operation of two crucial components of data center networks: (1) the control plane and (2) middleboxes.

We focus on these components for two reasons. First, our application of MPA to the data center networks of a large OSP (Chapter 2), as well as other research [75, 113, 147], shows that control plane configuration changes and middleboxes both have a strong impact on the frequency of problems in data center networks. Second, both routers (which run the control plane) and middleboxes play an important role in meeting the needs of applications and end hosts, so it is critical to ensure both are robust. For example, we must ensure that: (1) routers *always* forward traffic to the desired middleboxes, and (2) middleboxes *always* process this traffic correctly and efficiently. However, routers and middleboxes operate very differently, so we need separate but complementary abstractions to ensure these devices do not cause functional or performance failures in data center networks.

Checking Control Plane Correctness. Today's data center networks generally rely on a traditional distributed control plane, with each router running one or more distributed routing protocols—e.g., Open Shortest Path First (OSPF) and Border Gateway Protocol (BGP)—to compute forwarding paths and generate the network's data plane.[3] Additional

[3]Software-defined networking (SDN) has not been widely adopted in data centers' physical infrastructure, because SDN's use of a logically centralized controller eliminates the inherent scalability and fault tolerance of a distributed control plane.



(a) Current data plane verifier

(b) Data plane generator and verifier

Figure 1.3: State-of-the-art methods for verifying data center networks

mechanisms such as access control lists, route filters, and route redistribution may also be used to satisfy various functional requirements.

Unfortunately, control plane configurations are notoriously buggy [64, 150], due in part to the complexity of configuring routing protocols and their interactions [40, 99, 106]. While functional failures sometimes occur as soon as a buggy configuration is applied (and routing has re-converged), other errors manifest *only* during infrastructure faults. These "hidden" errors can have catastrophic consequences for data centers [138]. Another large class of functional failures arise when refactoring a network's control plane to eliminate problematic design practices: e.g., consolidating routing domains to reduce complexity [40] or replacing devices to reduce the number of vendors and models in the network (Section 2.4).

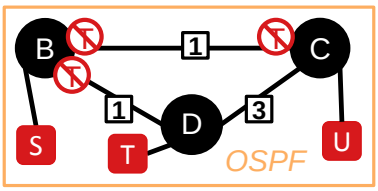
Proactively detecting errors that only manifest during infrastructure faults, as well as identifying differences between an original and re-factored control plane, is impossible, or at least impractical, using existing verification tools [66, 87, 88, 91, 105]. In particular, existing tools (Figure 1.3) are either: (1) limited to checking the data center network's current data plane, or (2) must generate the data plane for every possible infrastructure fault by simulating the low-level message exchanges of individual routing protocols. Ideally, a network operator would proactively check each new control plane configuration before applying it to the network; given that thousands of configuration changes may occur each month (Section 2.2), such checks must happen in a matter of minutes in order to be practical.

To efficiently check the correctness of data center network control planes, we must address the following questions:

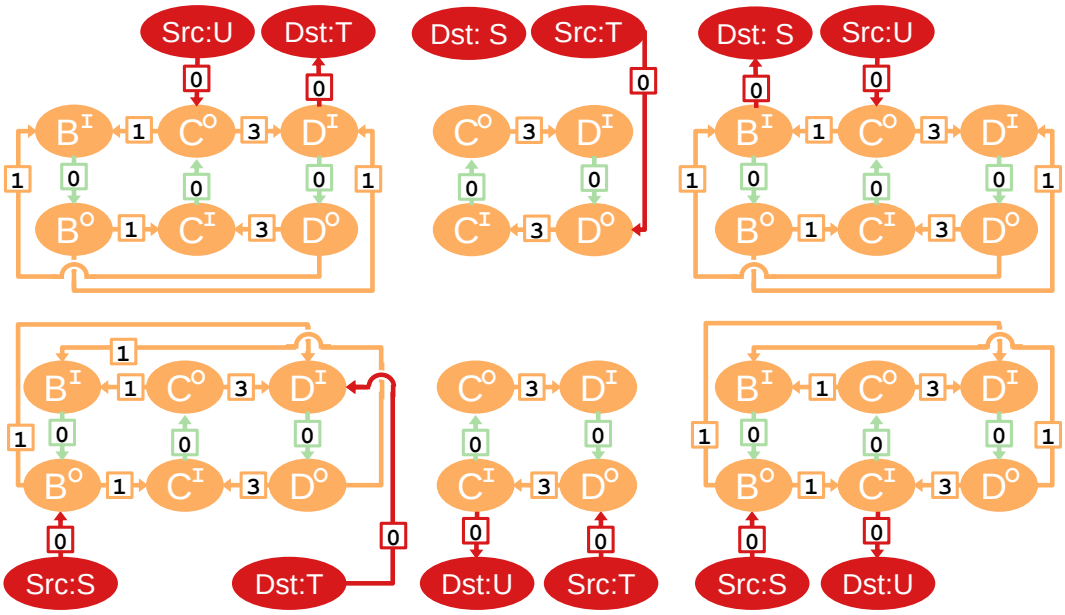
- How do we concisely model all possible data planes that may result from a given control plane configuration? We must capture the behavior of multiple routing protocols—e.g., we show in Section 2.2 that many data centers use both BGP and OSPF—and their complex interactions [99, 106] without individually modeling each data plane.
- How do we encode the invariants we want to check such that they can be quickly verified under arbitrary infrastructure faults using our model? We must be able to handle a variety of security and availability invariants—e.g., traffic between subnets S_1 and S_2 is *always* blocked or *always* traverses a middlebox—as well as control plane equivalence—i.e., verify that two control planes *generate the same data plane* under arbitrary infrastructure faults.

In Chapter 3, we introduce an *abstract representation for control planes* (ARC) that addresses these issues. ARC abstracts the mechanics of individual routing protocols and uses a series of weighted digraphs to model the protocols' collective impact on the network's data plane. Figure 1.4 shows an example network control plane and its corresponding ARC. Such modeling—which we describe in detail in Sections 3.3 and 3.4—is made possible by our observation that data center networks tend to use a limited set of routing protocols which interact in very specific ways (Section 3.6). With ARC, verifying key invariants boils down to computing simple graph characteristics, such as connected components and max-flow (Section 3.5.1). Checking the equivalence of two control planes is simply a matter of comparing the edges and weights of the graphs in each control plane's ARC (Section 3.5.2). By applying ARC to a subset of the data center networks studied in Chapter 2, we show that such proactive analyses generally run in a matter of seconds, which is three to five orders of magnitude faster than state-of-the-art tools.

Maintaining Middlebox Functionality and Performance. While ARC ensures that routers correctly forward and filter traffic, even in the presence of infrastructure faults, ARC does not ensure that middleboxes behave correctly. Middleboxes, also known as network functions, perform rich packet processing to offer security, performance, and monitoring



(a) Control plane for a network with three subnets (squares) and three routers (circles) participating in a single OSPF instance (rectangle); no-entry symbols indicate inbound ACLs on traffic from T



(b) Abstract representation for the control plane (ARC): it contains one digraph for every pair of source and destination subnets; vertices correspond to routing processes (two per process for reasons described in Section 3.3); edges represent the possible flow of data traffic enabled by the exchange of routing information between the connected processes

Figure 1.4: Example network with a single OSPF instance and its ARC

capabilities not available on routers and switches. Unlike routers and switches, middleboxes' operations are typically: (1) *complex*—e.g., deep packet inspection and elaborate packet modifications are common; and (2) *stateful*—i.e., the processing of one packet influences the processing of a later packet from the same connection or end host. Consequently, it is challenging to maintain suitable middlebox performance and transparently recover from middlebox-related problems (e.g., connectivity errors, hardware failures, and software issues [75, 113]).

One step towards addressing these issues is to replace individual hardware appliances—the

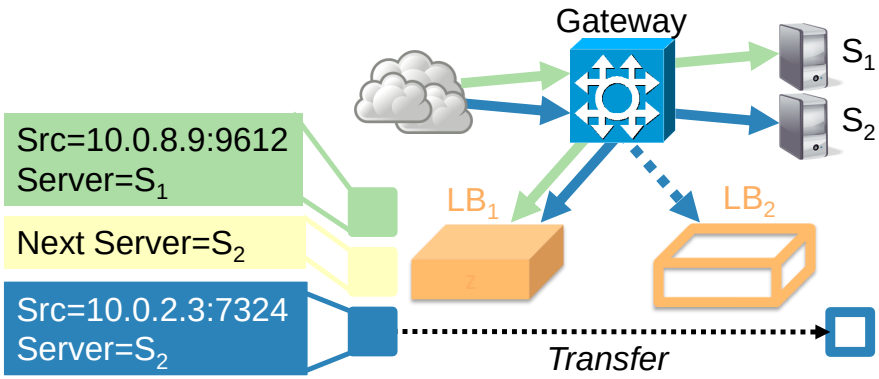


Figure 1.5: A scenario requiring scale-out and special handling of middlebox state to avoid performance and functional failures: The load balancer (e.g., HAProxy [10]) assigns incoming connections to application servers (e.g., S_1 and S_2) in a round robin fashion. For each active flow, the load balancer maintains a connection object with source IP and port and the name of the selected server. It also maintains a record of which server should receive the next connection. If a second load balancer instance is launched and the blue (darker) flow is reassigned to the second instance to avoid performance issues, then the flow-specific state must be moved to ensure active connections are not broken.

norm in today's data center networks [75, 113]—with a collection of software instances that expose a "one-big-middlebox" abstraction—i.e., the illusion of a *monolithic, always available, predictably performing middlebox*. With such a deployment model, new middlebox instances can be quickly launched on compute nodes anywhere in the data center, eliminating the long provisioning and repair times associated with hardware appliances [113]. Furthermore, by centralizing the middlebox configuration interface, such an abstraction reduces the likelihood of configuration incompatibilities between middlebox instances (e.g., mismatched cryptographic keys [113]).

However, the fluidity of software middleboxes must not be taken for granted. Great care must be taken in managing the lifecycle of middlebox instances and the distribution of traffic among them, lest a new class of problems arise due to missing or inconsistent middlebox state. As an example, consider a scenario where a layer-4 load balancer (LB_1) is distributing incoming connections among a pool of application servers (Figure 1.5). If the load balancer's performance approaches a critical threshold, we must launch a new instance (LB_2) and reroute some traffic to the new instance to preserve the abstraction of a monolithic,

predictably performing middlebox. However, rerouting in-progress connections to the new instance may cause those connections to be terminated, because the new instance lacks the necessary state indicating which server a connection has been assigned to. Thus, in an effort to avoid a performance failure we have caused a functional failure. As an alternative, we could route only new connections to the new load balancer instance [144], but the load on the original instance would not be reduced until some in-progress connections complete and a performance failure may still occur.

As the above example illustrates, providing a one-big-middlebox abstraction without compromising on performance or functionality requires *special handling of middlebox state*. In particular, we must ensure that the middlebox state associated with a specific set of traffic is available at the middlebox instance responsible for processing that traffic. To achieve this, we must answer several important questions:

- How do we efficiently make a middlebox instance's state available at another instance? Middlebox state may be quite complex, and traffic may be arbitrarily divided among middlebox instances.
- How do we avoid race conditions? Packets may continue to arrive at a middlebox instance while we are making its state available at another instance; unless care is taken, the state available at each middlebox instance may be incomplete or incorrect.
- How do we accommodate a variety of middleboxes with minimal changes? Data center networks contain a variety of middlebox types, vendors, and models (Section 2.2), so accommodating a wide range of middleboxes in a largely non-intrusive fashion is key to making middlebox state management practical.

In Chapter 4, we introduce a novel *middlebox state management framework*, called OpenNF, that addresses these issues. OpenNF allows middleboxes' internal state to be replicated, transferred, or shared at fine granularity to facilitate the realization of a one-big-middlebox abstraction. The complexities of distributed state control are handled by a logically centralized OpenNF controller that, when requested, guarantees loss-freedom,

order-preservation, and various levels of consistency for middlebox state and packets. Using traces of traffic exchanged with a cloud data center [78], we show that OpenNF can prevent both functional and performance failures by safely transferring state for hundreds of flows in just a few hundred milliseconds. Furthermore, the packet processing times at middleboxes increase by less than 6% during such transfers.

1.3 Contributions

In summary, this thesis makes the following contributions:

- We have developed a *management plane analytics* (MPA) framework [18, 74] that uncovers the relationships between network operators' management practices and the frequency of problems in data center networks. By applying MPA to over 850 data center networks operated by a large online service provider, we have uncovered several practices—e.g., network size, number and type of configuration changes, and the presence of middleboxes—that strongly influence the number of problems these networks experience; we have also found several instances where network operators' perceptions conflict with reality.
- We have designed an *abstract representation for control planes* (ARC) [1, 72] that models data center networks' forwarding and filtering behaviors at a higher level than today's network verifiers, thus enabling more direct proactive analysis. ARC enables network operators to verify that key functional invariants always hold, even in the presence of arbitrary infrastructure faults, in just a few seconds.
- Lastly, we have built a *middlebox state management framework* [28, 71, 73], called OpenNF, that provides safe and efficient control of middlebox state. This allows quick, safe, and fine-grained reallocation of flows across middlebox instances in order to prevent infrastructure fault or overload-induced failures. OpenNF was awarded the Internet Research Task Force (IRTF) Applied Network Research Prize for its relevance

to ongoing standardization efforts.

We have made the code for all of these frameworks publicly available [1, 18, 28] to allow network operators to apply our solutions to their own data center networks.

2 IDENTIFYING PROBLEMATIC PRACTICES USING MPA

Our first step toward reducing failures in data center networks is to *understand how network operator's design and operational decisions impact the frequency of network problems.* Such an analysis can help operators improve their network management practices, as well as identify network management challenges that require new solutions.

We begin this chapter (Sections 2.1 and 2.2) with a systematic characterization of the management practices employed in over 850 data center networks managed by a large online service provider (OSP). We present detailed definitions of practices in Section 2.1.2. This characterization provides the first in-depth look into the management practices used in modern data center networks. We find significant diversity in management practices—even within this single organization. This provides further evidence of the networking community's limited understanding of which practices are "best" (an issue we raised in Section 1.1) and highlights the importance of analyzing how network management practices impact the likelihood of network failures.

The rest of this chapter (Sections 2.3 through 2.5) presents a *management practice analytics* (MPA) framework that an organization can use to identify how its management practices impact the health of its networks (i.e., the number of problems its networks experience). In particular, MPA helps network operators derive the *top* k *management practices that impact network health.* Armed with this information, operators can develop suitable best practices to improve organization-wide design and operational procedures. MPA also helps operators *predict*, in an ongoing fashion, what impact a specific set of management practices will have on the *health of individual* networks. This goes beyond focusing on the top practices; it incorporates the effects of one-off deviations from established procedures, as well as the effects of management practices whose impact on network health manifests only in a narrow set of situations. Armed with such metrics, operators can focus on improving their management practices in networks that are predicted to have more problems.

2.1 Inferring Management Practices

One of the primary challenges in analyzing management practices is that they are not explicitly logged. While the control and date planes can be queried to quantify their behavior [27, 58, 80, 104, 134], no such capability exists for management practices. This gap stems from humans being the primary entities responsible for network management. Operators translate high-level intents into a suitable setup of devices, protocols, and configurations to create a functional, healthy network. Even when recommended procedures are documented, there is no guarantee that operators adhere to these practices.

Fortunately, we are able to infer management practices from other readily available data sources. In this section, we describe these sources and the management practice *metrics* we can infer.

2.1.1 Data Sources

We can infer management practices and network health from three data sources that are commonly available. Such data sources have already been used in prior work, albeit to study a limited set of management practices [41, 92, 115, 137]. We build upon these efforts to provide a more thorough view of management practices and their relationship to network health. The data sources are:

1) Inventory records. Most organizations directly track the set of networks they manage, and the role the networks play. Organizations that manage a large number of devices typically do not view all devices as belonging to one network, even if the devices are housed within the same building; instead they view the devices as partitioned across multiple networks. A *network* in this context is a collection of devices that either connects compute equipment that hosts specific workloads (e.g., a Web service) or connects other networks to each other or the external world. Organizations record the vendor, model, location, and type (switch, router, load balancer, etc.) of every device in their deployment, and the network it belongs to. This

data can be used to infer a network's basic composition and purpose.

2) Device configuration snapshots. Network management systems (NMS) track changes in device configurations to aid network operators in a variety tasks, such as debugging configuration errors or rolling back changes when problems emerge. NMSes such as RANCID [24] and HPNA [11] subscribe to syslog feeds from network devices and snapshot a device's configuration whenever the device generates a syslog alert that its configuration has changed. Each snapshot includes the configuration text, as well as metadata about the change, e.g., when it occurred and the login information of the entity (i.e., user or script) that made the change. The snapshots are archived in a database or version control system.

3) Trouble ticket logs. When users report network problems, or monitoring systems raise alarms, a trouble ticket is created in an incident management system. The ticket is used to track the duration, symptoms, and diagnosis of the problem. Each ticket has a mix of structured and unstructured information. The former includes the time the problem was discovered and resolved, the name(s) of device(s) causing or effected by the problem, and symptoms or resolutions selected from pre-defined lists; the latter includes syslog details and communication (e.g., emails and IMs) between operators that occurred to diagnose the issue.

2.1.2 Metrics

Using these data sources, we can infer management practices and network health, and model them using *metrics*. We broadly classify management practices into two classes (Table 2.1): *design practices* are long-term decisions concerning the network's structure and provisioning (e.g., selecting how many switches and from which vendors); *operational practices* are day-to-day activities that change the network in response to emerging needs (e.g., adding subnets).

Design practices. Design practices influence four sets of network artifacts: the network's purpose, its physical composition, and the logical structure and composition of its data and

Design practices
D1. Number of services, users, or networks connected
D2. Number of devices, vendors, models, types (e.g., switch, router, firewall), and firmware versions
D3. Hardware and firmware heterogeneity
D4. Number of data plane constructs used (e.g., VLAN spanning tree, link aggregation), and instance counts
D5. Number and size of BGP & OSPF routing instances
D6. Intra- and inter-device config reference counts
Operational practices
O1. Number of config changes and devices changed
O2. Number of automated changes
O3. Number and modality of changes of specific types (e.g., interface, ACL, router, VLAN)
O4. Number of devices changed together

Table 2.1: Management practice metrics

control planes. The metrics we use to quantitatively describe a network's purpose and its physical composition are rather straightforward to compute, and are listed in lines D1 and D2 in Table 2.1. We synthesize these metrics to measure a network's *hardware heterogeneity* using a normalized entropy metric (line D3): $\frac{-\sum_{i,j} p_{ij} \log_2 p_{ij}}{\log_2 N}$, where p_{ij} is the fraction of devices of model i of type j (e.g., switch, router, firewall, load balancer) in the network, and N is the size of the network. This metric captures the extent to which multiple models of the same type of device are used; a value close to 1 indicates significant heterogeneity. We compute a similar firmware heterogeneity metric.

Computing metrics that capture the logical composition and structure of the data and control planes is more intricate as it involves parsing configuration files. To conduct our study, we extended Batfish [66] to parse the configuration languages of various device vendors (e.g., Cisco IoS). Given parsed configurations, we determine the logical composition of the data plane by enumerating the number of data plane constructs used (e.g., spanning tree, VLAN, link aggregation), as well as the number of instances of each (e.g., number of VLANs configured); Table 2.1, line D4.

To model control plane structure, we leverage prior work on configuration models [40].

In particular, we extract routing instances from device configurations, where each instance is a collection of routing processes of the same type (e.g., OSPF processes) on different devices that are in the transitive closure of the "adjacent-to" relationship. A network's routing instances collectively implement its control plane. We enumerate the number of such instances, as well as the average size of each instance (Table 2.1, line D5) using the same methodology as Benson et al. [40].

Finally, we enumerate the average number of inter- and intra-device configuration references in a network [40]. These metrics (Table 2.1, line D6) capture the configuration complexity imposed in aggregate by all aspects of a network's design, as well as the impact of specific configuration practices followed by operators.

Operational practices. We infer operational practices by comparing two successive configuration snapshots from the same device. If at least one stanza differs, we count this as a configuration change.

We compute basic statistics about the configuration changes observed over a certain time window (Table 2.1, line O1). In addition, we study the modality of changes (line O2). We infer modality (automated vs. manual) using the login metadata stored with configuration snapshots: we mark a change as *automated* if the login is classified as a special account in the organization's user management system. Otherwise we assume the change was *manual*. This conservative approach will misclassify changes made by scripts executing under a regular user account, thereby under-estimating the extent of automated changes.

To model change type, we leverage the fact that configuration information is arranged as *stanzas*, each containing a set of options and values pertaining to a particular construct—e.g., a specific interface, VLAN, routing instance, or ACL. A stanza is identified by a type (e.g., interface) and a name (e.g., TenGigabit0/1). When part (or all) of a stanza is added, removed, or updated, we say a change of type T occurred, where T is the stanza type. We count the number of changes of each type over a certain time window (Table 2.1, line O3).

There are a few challenges and limitations with this approach. First, type names differ

between vendors: e.g., an ACL is defined in Cisco IoS using an `ip access-list` stanza, while a `firewall filter` stanza is used in Juniper JunOS. We address this by manually identifying stanza types on different vendors that serve the same purpose, and we convert these to a vendor-agnostic type identifier. Second, even after generalizing types, a change with the same effect may be typified differently on different vendors: e.g., an interface is assigned to a VLAN in Cisco IoS using the `switchport access vlan` option within an `interface` stanza, while in Juniper JunOS the `interface` option is used within a `vlan` stanza; even though the effect of the change is the same, it will be typified as an interface change on a Cisco device and a VLAN change on a Juniper device. Operators using MPA should be aware of this limitation and interpret prediction results according to the mix of vendors in their networks.

In addition to computing change metrics over changes on individual devices, we compute change metrics over *change events* (Table 2.1, line O4). Change events account for the fact that multiple devices' configurations may need to be changed to realize a desired outcome. For example, establishing a new layer-2 network segment (e.g., a VLAN) requires configuration changes to all devices participating in the segment.

To identify change events, we group changes using a simple heuristic: if a configuration change on a device occurs within δ time units of a change on another device in the same network, then we assume the changes on both devices are part of the same change event. Figure 2.1 shows how different values of δ influence the number of change events. The rest of our analysis uses $\delta = 5$ minutes, because operators indicated they complete most related changes within such a time window.

Network Health. The health of a network can be analyzed from many perspectives, including performance (e.g., latency or throughput), quality of experience (e.g., application responsiveness), and failure rate (e.g., packet loss or link/device downtime). Networks are often equipped with monitoring systems that track these metrics and raise alarms when critical thresholds are crossed. In the networks we study, trouble tickets are automatically

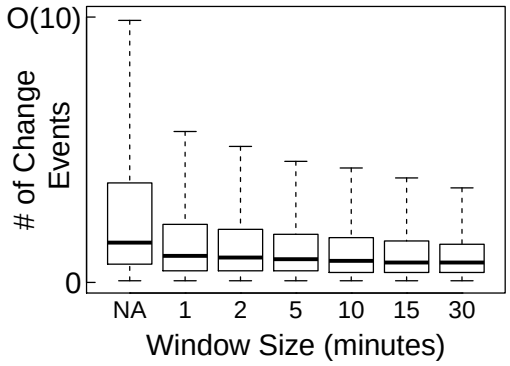


Figure 2.1: Impact of change grouping threshold (δ) on the number of change events: each box shows the 25th, 50th, and 75th percentile number of change events per-network per-month using various values of δ; whiskers indicate the most extreme data points within twice the interquartile range

created when such alarms are raised. Tickets are also created when users report problems or operators conduct planned maintenance. We exclude the latter from our analysis, because maintenance tickets are unlikely to be triggered by performance or availability problems.

Given that ticket logs capture a wide-range of network issues, operators view tickets as a valuable measure of network health. In particular, operators from the OSP whose networks we study indicated that number of tickets is a useful metric. Other metrics computed from network tickets (e.g., number of problems marked as high severity, mean time to resolution, etc.) are less useful because of inconsistencies in ticketing practices: e.g., severity levels are often subjective, and tickets are sometimes not marked as resolved until well after the problem has been fixed.

2.2 Characterization of Management Practices

We now provide a detailed characterization of the network management practices used by a large online service provider (OSP). This offers a unique and rich view into the practices used in modern data center networks. The management practices used in data center networks operated by other organizations may differ, but our characterization is nonetheless useful for identifying possible contributors to data center network failures. For brevity, we quantify a

Property	Value
Months	17, Aug 2013 – Dec 2014
Networks	850+
Services	O(100)
Devices	O(10K)
Config snapshots	O(100K), ≈450GB
Tickets	O(10K), ≈80MB

Table 2.2: Size of datasets

subset of the practice metrics in Table 2.1. Overall, we find significant diversity in the design and operational practices employed across the OSP's networks.

Dataset. The OSP owns 850+ data center networks that are managed based on documented "best practices." Each network hosts one or more Web services or interconnects other networks. Our datasets cover a 17 month period from August 2013 through December 2014. Table 2.2 shows its key aspects. For confidentiality we do not list exact numbers.

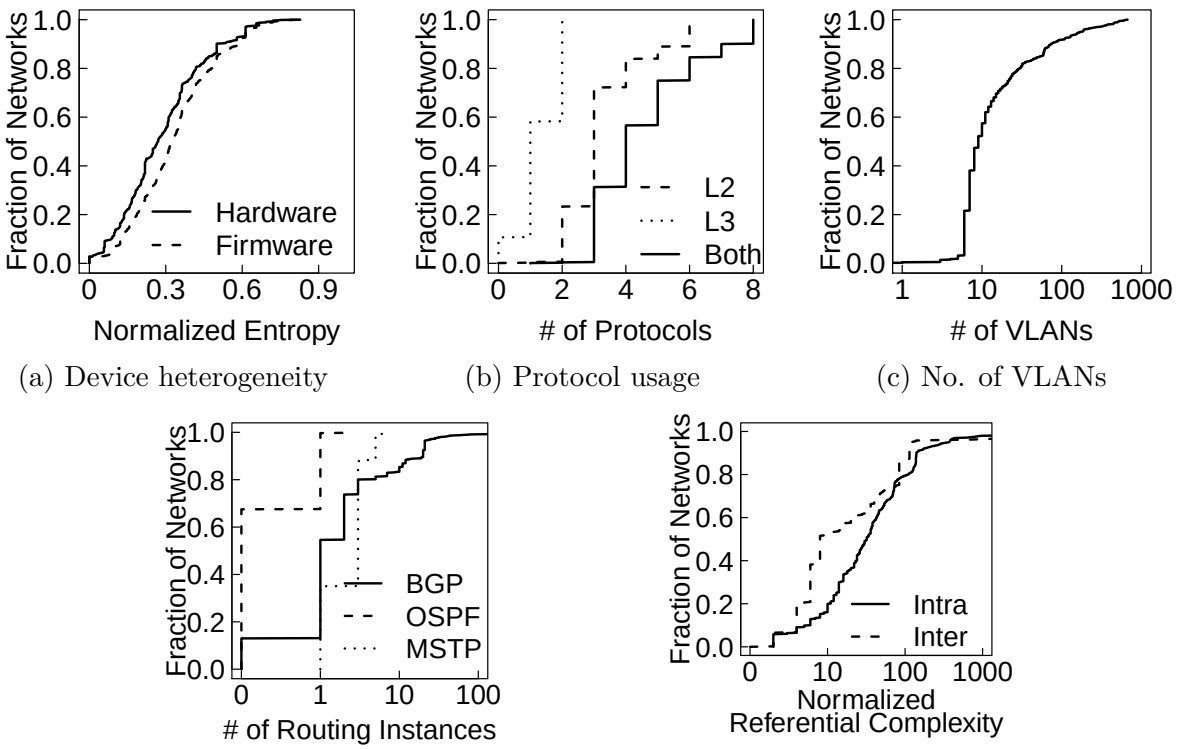
Design Practices. We start by examining the OSP's networks in terms of their purpose, physical composition, and control plane design.

The majority (81%) of networks host *only one workload*—networks are quite homogeneous in this respect. A handful of networks do not host any workloads; they only connect networks to each other or the external world.

The networks contain a mix of device types, including routers, switches, firewalls, application delivery controllers (ADCs)[1], and load balancers. Most networks (86%) have multiple types of devices and 71% of networks contain at least one middlebox (firewall, ADC, or load balancer). In half of the networks, at least 10% of the devices are middleboxes, and in one quarter of the networks at least 25% of the devices are middleboxes.

We also find that over 81% of networks contain devices from more than one vendor, with a maximum of six, and over 96% of networks contain more than one device model, with a maximum of 25. Thus, some networks must use more than one device model for the same

[1]ADCs perform TCP and SSL offload, HTTP compression and caching, content-aware load balancing, etc.



(a) Device heterogeneity (b) Protocol usage (c) No. of VLANs

(d) No. of routing Instances (e) Referential complexity

Figure 2.2: Characterization of design practices

type of device. Indeed, a closer look at the hardware entropy of the networks (solid line in Figure 2.2a) shows that only 4% of networks have just one model and one type of device; *the remaining (96% of) networks have varying degrees of heterogeneity*, up to a maximum entropy metric value of 0.82. The extent of firmware heterogeneity is similar (dashed line in Figure 2.2a).

Next, we look at the logical composition and structure of the data and control planes. As shown in Figure 2.2b, all networks use at least two layer-2 protocols (VLAN, spanning tree, link aggregation, unidirectional link detection (UDLD), DHCP relay, etc.), and 89% of networks use at least one routing protocol (BGP and/or OSPF). In 10% of networks, eight different protocols are used. Overall *there is significant diversity in the combination of protocols used*.

We find the same diversity in the number of instances of each protocol. Less than 5

VLANs are configured in 5% of networks, but over 100 VLANs are configured in 9% of networks (Figure 2.2c). Similarly, 86% of networks use BGP for layer-3 routing, with just one BGP instance in 39% of networks and more than 20 instances in 8% of networks (Figure 2.2d). In contrast, only 31% of networks use OSPF for layer-3 routing, with just one or two OSPF instances used in these networks.

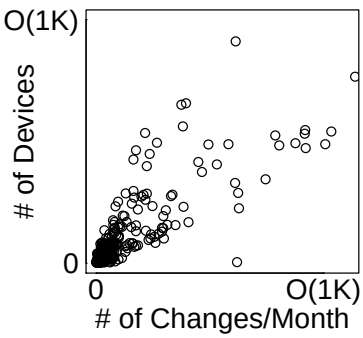
Finally, to characterize configuration complexity, Figure 2.2e shows a CDF of intra- and inter-device referential complexity. We find that some networks' configuration is extremely complex (based on Benson et al.'s metrics [40]): in 20% of networks, the mean intra- and inter-device reference counts are above 100. However, it is worth noting that: (1) *the range in complexity is rather large*—it varies by 1–2 orders of magnitude across networks—and (2) most networks have significantly lower configuration complexity metrics than the worst 10%.

Operational Practices. We now characterize the frequency, type, and modality of configuration changes, as well as those of change events.

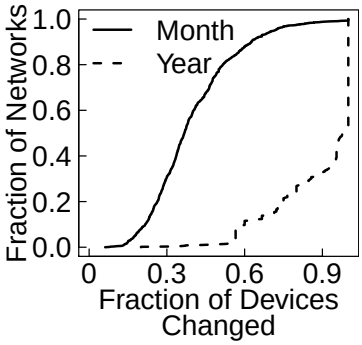
In general, the average number of configuration changes per month is correlated with network size (Figure 2.3a; Pearson correlation coefficient of 0.64). However, *several large networks have relatively fewer changes per month*: e.g., one network has over 300 devices but less than 150 changes per month. Likewise, *there are several small networks with a disproportionately high change rate.* Furthermore, not every device is changed every month—in 77% of networks less than half of a network's devices are changed in a given month—but most devices are changed at least once per year—in 80% of networks more than three-quarters of the devices are changed in a year (Figure 2.3b). Thus, *changes occur frequently, and to different sets of devices in different months.*

We now analyze different types of changes. Across our entire dataset there are ≈480 different types of changes. Figure 2.3c shows CDFs of the fraction of changes in which at least one stanza of a given type is changed. On a per-network basis, interface changes are the most common, followed by pool (used on load balancers), ACL, user, and router.[2]

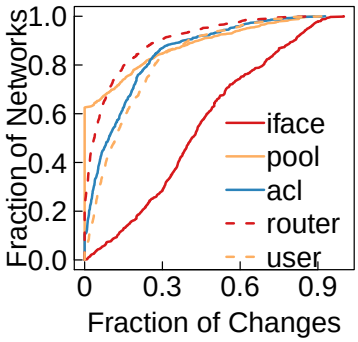
[2]There are no pool changes in 63% of networks because these networks do not contain load balancers.



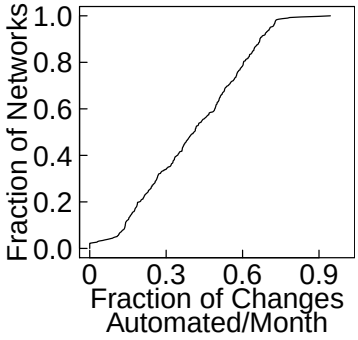
(a) Avg. number of changes per month



(b) Avg. fraction of devices changed



(c) Most frequent config changes

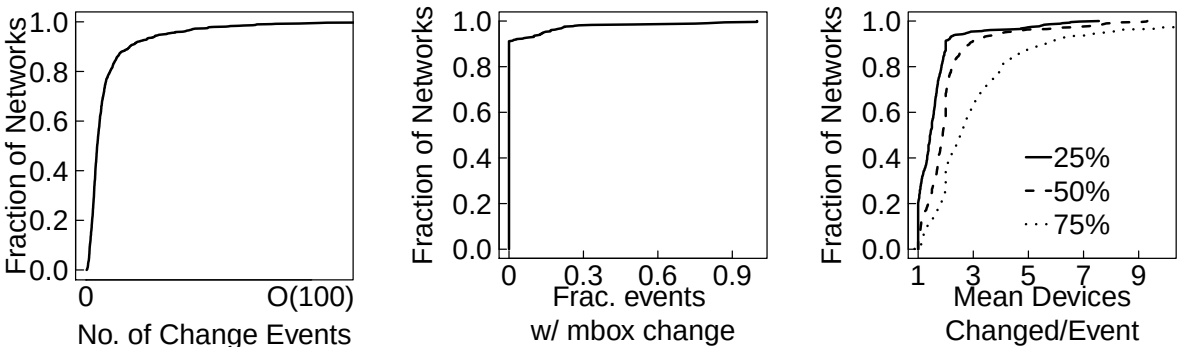


(d) Frac. of changes automated

Figure 2.3: Characterization of configuration changes

Among the above most-frequently changed types, pool changes are also the most frequently automated—more than half of all pool changes are automated in 77% of networks—followed by ACL and interface changes. We also look at the extent of automation over *all* types of changes. As shown in Figure 2.3d, more than half (quarter) of the changes each month are automated in 41% (81%) of networks. In general, *we note a significant diversity in the extent of automation*: it ranges between 10% and 70%. Equally interestingly, the fraction of automated changes is not strongly correlated with the number changes (Pearson correlation coefficient is 0.23). Furthermore, the types of changes that are automated most frequently—sflow and QoS—are not the most frequent types of changes.

Lastly, we look at change events, both in terms of how many there are in a network as well as the composition of an event (in terms of number of devices changed). Figure 2.4a shows a distribution of the number of change events. They are few in number ($\mathcal{O}(10)$) in



(a) Avg. number of change events per month

(b) Frac. of events that change a middlebox

(c) Avg. devices changed per event

Figure 2.4: Characterization of configuration change events

most networks (80%); however about 5% of the networks experience tens if not hundreds of change events in a month. We see a similar *diversity in the number of change events* involving middleboxes (Figure 2.4b). Figure 2.4c shows a CDF of the average number of devices changed per change event. Most change events we see across networks are small: in about half of the networks, a change event affects only one or two devices (on average). Further, in almost all networks, the average change event affects only one type of device and one device model. Limiting changes to just a few, similar devices is intuitively a good practice to simplify debugging and rollback.

Implications. The diversity of management practices used in the data center networks of a single organization corroborates the takeaway from our network operator survey (Section 1.1): *operators have little agreement on which management practices are "best".* Consequently, network operators need a framework to systematically understand practices' impact on the health of (i.e., the frequency of problems in) their data center networks.

In the rest of this chapter we present such a *management practice analytics* (MPA) framework. MPA derives the top k management practices that impact network health by analyzing statistical dependencies (Section 2.3) and causal relationships (Section 2.4) between an organization's practices and the health of its networks. MPA also builds models that accurately predict the health of individual networks, based on a set of management practices,

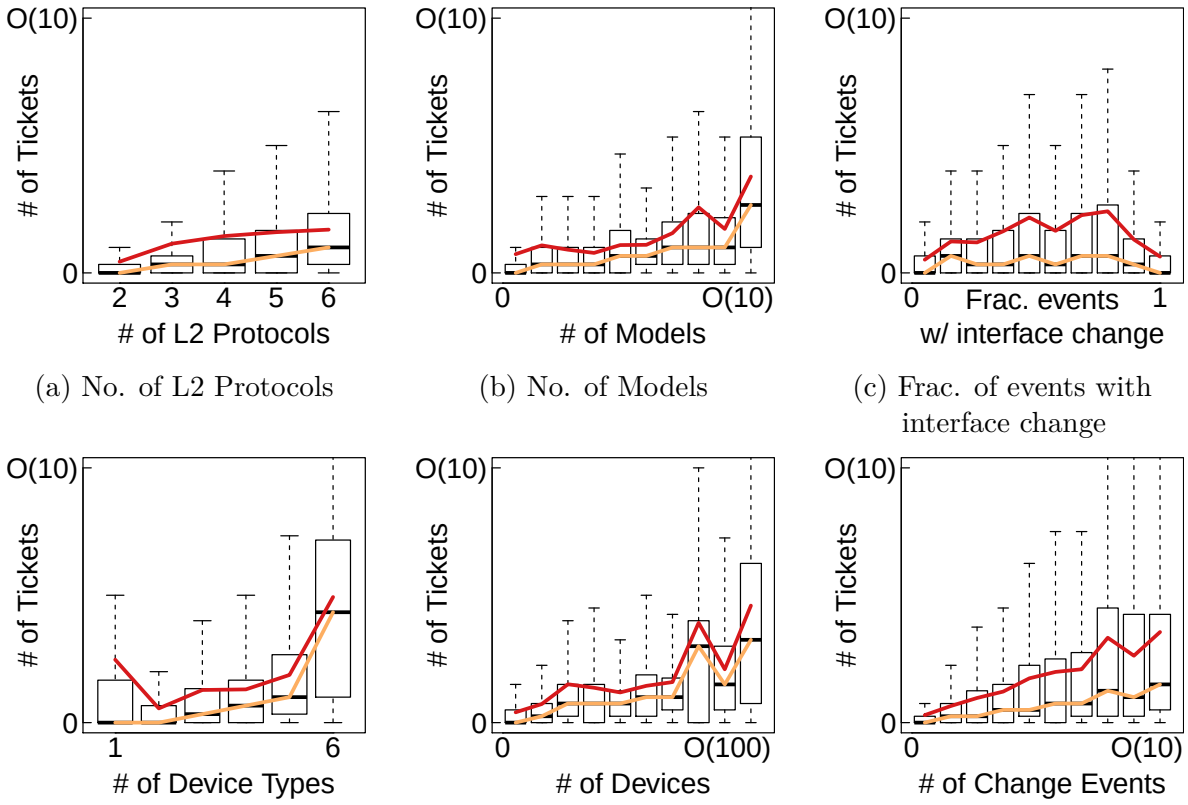
using machine learning (Section 2.5). We describe MPA's mechanisms in more detail below, and we use data from the OSP to illustrate how they work.

2.3 Identifying Statistical Dependencies

Common approaches for decomposing the impact of different factors include analysis of variance (ANOVA) [21] and principal/independent component analyses (PCA/ICA) [51]. However, these techniques make key assumptions that do not always hold for network management practices. In particular, ANOVA assumes factors are linearly related, but management practices may not have a linear, or even monotonic, relationship with network health: e.g., Figure 2.5 shows three different management practices—*number of L2 protocols*, *number of models*, and *fraction of events with an interface change*—that have a linear, monotonic, and non-monotonic relationship, respectively, with number of tickets. ICA attempts to express the outcome (health metric) as a linear or non-linear combination of independent components; applying PCA first helps identify the components to feed to ICA. However, the components output by PCA are linear combinations of a subset of management practices. Thus, similar to ANOVA, this approach makes the implicit assumption that linear combinations of practice metrics can explain network health. Furthermore, the outcome of ICA may be hard to interpret (especially if it relies on a non-linear model).

To overcome this challenge, we identify statistical dependencies using a more general approach: *mutual information* (MI). When computed between a management practice metric and network health, MI measures how much knowing the practice reduces uncertainty about health. Crucially, MI does not make assumptions about the nature of the relationship.

Mutual information. The MI between variables X and Y (a management practice and network health) is defined as the difference between the entropy of Y, $H(Y)$, and the conditional entropy of Y given X, $H(Y|X)$. Entropy is defined as $H(Y) = -\sum_i P(y_i)\log P(y_i)$, where $P(y_i)$ is the probability that $Y = y_i$. Conditional entropy is defined as $H(Y|X) =$



(a) No. of L2 Protocols (b) No. of Models (c) Frac. of events with interface change

(d) No. of Device Types (e) No. of Devices (f) No. of Change Events

Figure 2.5: Ticket counts based on management practices: boxes show 25th and 75th percentile, while whiskers show twice the interquartile range; red (dark) lines show the average number of tickets, while orange (light) lines show the median

$\sum_{i,j} P(y_i, x_j) \log \frac{P(x_j)}{P(y_i, x_j)}$, where $P(y_i, x_j)$ is the probability that $Y = y_i$ and $X = x_j$. MI is symmetric.

We also examine statistical dependencies between management practices using *conditional mutual information* (CMI). The CMI between a pair of management practices and network health measures the expected value of the practices' MI, given health.[3] The CMI for two variables X_1 and X_2 relative to variable Y is defined as $H(X_1|Y) - H(X_1|X_2, Y)$. Like MI, CMI is also symmetric (with respect to X_1 and X_2).

Binning. Prior to computing MI or CMI, we compute the value of each management practice and health metric on a monthly basis for each network, giving us ≈11K data points. We bin the data for each metric using 10-equal width bins, with the 5th percentile value as the lower

[3]In a sense, the pair's joint probability distribution.

Management Practices	Avg. Monthly MI
No. of devices (D)	0.388
No. of change events (O)	0.353
Intra-device complexity (D)	0.329
No. of change types (O)	0.328
No. of VLANs (D)	0.313
No. of models (D)	0.273
No. of device types (D)	0.221
Avg. devices changed per event (O)	0.215
Frac. events w/ interface change (O)	0.201
Frac. events w/ ACL change (O)	0.198

Table 2.3: Top 10 management practices related to network health according to average monthly MI: parenthetical annotation indicates practice category (D=design, O=operational)

bound for the first bin, and the 95th percentile value as the upper bound for the last bin. Networks whose metric value is below the 5th (above the 95th) percentile are put in the first (last) bin.

Our motivation for this binning strategy is twofold. First, in our characterization of management practices (Section 2.2), we observed that many management practices have a long tail: e.g., number of VLANs (Figure 2.2c). Using the 5th and 95th percentile bounds for the first and last bins significantly reduces the range of values covered by each bin, thereby reducing the likelihood that the majority of networks will fall into just one or two bins. Second, minor deviations in a management practice or health metric—e.g., one more device or one more ticket—are unlikely to be significant. Our binning helps reduce noise from such minor variations.

Results for the OSP. We now present statistical dependence results for the OSP. Table 2.3 shows the 10 management practices that have the strongest statistical dependence with network health. It includes five *design practices* and five *operational practices*, thus highlighting the potential importance of both types of practices to a healthy network.

We visually confirmed the assessment that the practices in Table 2.3 have a statistical dependence with network health. For example, Figure 2.5 illustrates the strong statistical dependence with network health for *number of devices*, *number of change events*, *number*

Management Practice Pair		CMI
Frac. events w/ pool change (O)	Frac. events w/ mbox change (O)	1.107
Firmware entropy (D)	Hardware entropy (D)	0.978
No. of OSPF instances (D)	No. of L3 protocols (D)	0.923
No. of models (D)	No. of change types (O)	0.735
No. of BGP instances (D)	Inter-device complexity (D)	0.732
No. of device types (D)	No. of models (D)	0.713
No. of BGP instances (D)	No. of L2 protocols (D)	0.601
Avg. size of an OSPF instance (D)	No. of change types (O)	0.576
Intra-device complexity (D)	Inter-device complexity (D)	0.574
No. of devices (D)	No. of VLANs (D)	0.569

Table 2.4: Top 10 pairs of statistically dependent management practices according to CMI: highlighted practices are in the top 10 according to MI; parenthetical annotation indicates practice category (D=design, O=operational)

of models, *number of device types*, and *fraction of change events with an interface change* (ranked 1st, 2nd, 6th, 7th, and 9th, respectively, in Table 2.3).

Interestingly, one of the practices which has high impact according to our operator survey (Figure 1.1)—*fraction of events with a middlebox change*—does not appear in the top 10 practices (Table 2.3); this practice is ranked 23 out of 28. This may be due to the fact that the majority of changes to the configurations of the OSP's middleboxes are simple adjustments to the server pools configured on load balancers.

Table 2.4 shows the 10 management practices that have the strongest statistical dependence with *each other*. We observe that six of the top 10 practices related to network health (Table 2.3) are statistically dependent with other practices (Table 2.4); this includes all five of the top design practices and one operational practice. In general, more design practices (as opposed to operational practices) are statistically dependent with each other. This trend stems from the natural connections between many design decisions: e.g., configuring more BGP instances results in more references between devices and increases inter-device complexity.

We also observe from Table 2.4 that several practices are dependent with multiple other practices: e.g., *number of models* is dependent with *number of device types* and *number of*

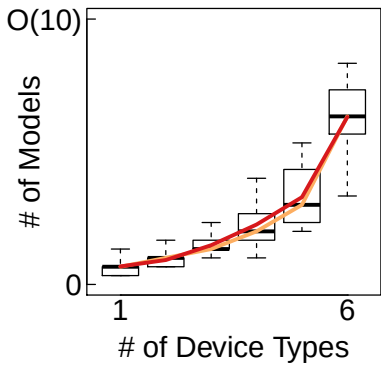


Figure 2.6: Relationship between *number of models* and *number of device types*: boxes show 25th and 75th percentile, while whiskers show twice the interquartile range; the red (dark) line shows the average number of models, while the orange (light) line shows the median

change types, and *number of change types* is also dependent with *average size of an OSPF instance*. Thus, evaluating the impact of a management practice on network health requires accounting for many other practices; we next discuss how we achieve this.

2.4 Identifying Causal Relationships

Although we can select the k practices with the highest MI as the top k management practices associated with network health, there is no guarantee these practices actually impact health. To establish a *causal relationship* between a management practice and network health, we must eliminate the effects of *confounding factors* (i.e., other practices) that impact this practice and network health [82]. For example, Figures 2.5b and 2.5d (and rows 6 and 7 in Table 2.3) show that *number of models* and *number of device types*, respectively, are related to network health, and Figure 2.6 (and row 6 in Table 2.4) shows that the two practices are also related to each other.

Ideally, we would eliminate confounding factors and establish causality using a *true randomized experiment*. In particular, we would ask operators to employ a specific practice (e.g., decrease the number of device models) in a randomly selected subset of networks. We would then compare the network health (outcome) across the selected (treated) and remaining (untreated) networks. Unfortunately, conducting such experiments takes time (on the order

of months), and requires operator compliance to obtain meaningful results. Moreover, true experiments ignore already available historical network data.

To overcome these issues, we use *quasi-experimental design* (QED) [125]. QED uses existing network data to affirm that an independent (or treatment) variable X has a causal impact on a dependent (or outcome) variable Y.

Matched design. We use a specific type of QED called the *matched design* [136]. The basic idea is to pair cases—each case represents a network in a specific month—that have equal (or similar) values for all confounding variables $\mathsf{Z}_1 ... \mathsf{Z}_n$, but different values for the treatment variable X. Keeping the confounding variables equal negates the effects of other practices on the outcome (network health), and increases our confidence that any difference in outcomes between the paired cases must be due to the treatment (practice under study).

Using a matched design to identify a causal relationship between a management practice and network health entails four key steps: (1) determine the practice metric values that represent treated and untreated; (2) match pairs of treated and untreated cases based on a set of confounding factors, a distance measure, and a pairing method; (3) verify the quality of the matches to ensure the effect of confounding practices is adequately accounted for; and (4) analyze the statistical significance of differences in outcomes between the treated and untreated cases to determine if there is enough support for a causal relationship.

A key challenge we face in using a matched design is obtaining a sufficient number of quality matches to provide an adequate foundation for comparing the outcomes between treated and untreated cases. As shown in Section 2.2, practices tend to vary significantly across networks. Furthermore, many management practices are statistically dependent with network health and each other (Section 2.3). We use nearest neighbor matching based on propensity scores [136] to partially address this challenge, but there are also fundamental limitations imposed by the size of our datasets.

We now describe the analysis steps in more detail, using *number of change events* as an example management practice for which we want to establish a causal relationship with

network health. At the end of the section, we present results for the 10 management practices that have the highest statistical dependence with network health for the OSP (Table 2.3).

1) Determining the treatment. While most other studies that use QEDs (e.g., those in the medical and social sciences) have a clear definition of what constitutes "treatment," there is no obvious, definitive choice for most management practices. The majority of our management practice metrics have an (unbounded) range of values, with no standard for what constitutes a "normal range": e.g., for the OSP's networks, the average number of change events per month ranges from 0 to hundreds (Figure 2.4a). Hence, we must decide what values constitute *treated* and *untreated*.

One option is to define untreated as the practice metric value that represents the absence of operational actions (e.g., no change events), or the minimum possible number of entities (e.g., one device model or one VLAN). However, we find it is often the case that: (1) several confounding practices will also have the value 0 or 1 (or be undefined) when the treatment practice has the value 0 or 1—e.g., when *number of change events* is 0, *number of change types*, *average devices changed per event*, and *fraction of events with a change of type T* are undefined; and (2) several confounding practices will be non-zero (or >1) when the treatment practice is non-zero. This observation makes sense, given that our CMI results showed a strong statistical dependence between many management practices (Table 2.4). Unfortunately, it makes it difficult to find treated cases with similar confounding practices that can be paired with the untreated cases.

Given the absence of a "normal range," and the strong statistical dependence between practices, we choose to use multiple definitions of treated and untreated and conduct multiple causal analyses. In particular, we use the same binning strategy discussed in Section 2.3 to divide cases into 5 bins based on the value of the treatment practice. Then we select one bin ($\mathfrak{b}$) to represent untreated, and a neighboring bin ($\mathfrak{b}+1$) to represent treated. This gives us four points of comparison: bin 1 (untreated) vs. bin 2 (treated), 2 vs. 3, 3 vs. 4, and 4 vs. 5; we denote these experimental setups as *1:2*, *2:3*, *3:4*, and *4:5*, respectively. More (or

fewer) bins can be used if we have an (in)sufficient number of cases in each bin. Later in this section, we discuss how to evaluate the quality of matches, which can help determine whether more (fewer) bins can be used.

2) Matching pairs of cases. Matching each treated case with an untreated case is the next step in the causal analysis process. For our causal conclusions to be valid, we must carefully select the confounding factors, distance measure, and pairing method used in the matching process.

During the matching process, it is important to consider all practices (except the treatment practice) that may be related to the treatment or outcome. Excluding a potentially important confounding practice can significantly compromise the validity of the causal conclusion, while including practices that are actually unassociated with the outcome imposes no harm—assuming a sufficiently large sample size and/or a suitable measure of closeness [135]. Therefore, we include all 28 of the practice metrics we infer, minus the treatment practice, as confounding factors.

One caveat of including many confounding practices is that it becomes difficult to obtain many *exact matches*—pairs of cases where both cases have the exact same values for all confounding practices. For example, exact matching produces at most 17 pairs (out of ≈11K cases) when *number of change events* is the treatment practice. The same issue exists when matching based on Mahalanobis distance [122].

We overcome this challenge using *propensity scores*. A propensity score measures the probability of a case receiving treatment (e.g., having a specific *number of models*) given the observed confounding practices (e.g., *number of device types*) for that case [136]. By comparing cases that have the same propensity scores—i.e., an equally likely probability of being treated based on the observed confounding practices—we can be confident that the actual presence or absence of treatment is not determined by the confounding practices. In other words, a treated case and an untreated case with the same propensity score have the same probability of having a given value for a confounding practice (e.g., *number of device*

Comp. Point	Untreated Cases	Treated Cases	Pairs	Untreated Matched	Abs. Std. Diff. of Means	Ratio of Var.
1:2	8259	1745	1742	1109	0.0000	1.0091
2:3	1745	616	614	431	-0.0002	1.0314
3:4	616	296	295	200	0.0052	1.0744
4:5	296	783	673	174	-0.0002	1.0411

Table 2.5: Matching based on propensity scores

types); thus propensity score matching mimics a randomized experiment.

Given propensity scores for all treated and untreated cases, we use the most common, and simplest, pairing method: *k=1 nearest neighbor* [135]. Each treated case is paired with an untreated case that results in the smallest absolute difference in their propensity scores. To obtain the best possible pairings, we match *with replacement*—i.e., allow multiple treated cases to be paired with the same untreated case. We also follow the common practice of discarding treated (untreated) cases whose propensity score falls outside the range of propensity scores for untreated (treated) cases.

Table 2.5 shows the matching results for each of the four comparison points for *number of change events*. There are significantly more matched pairs using propensity scores: up to 99.8% of treated cases are matched, versus <1% with exact matching. Furthermore, the number of untreated cases that are matched with treated cases is less than the number of matched pairs, implying that matching with replacement is beneficial.

3) Verifying the quality of matches. When matching based on propensity scores, rather than the raw values of confounding practices, it is important to verify that the distribution of values for each confounding practice is similar for both the matched treated cases and the matched untreated cases. Otherwise, the effects of confounding practices have not been successfully mitigated, and any causal conclusions drawn from the matched pairs may not be valid.

Figure 2.7 visually confirms the distribution equivalence for two of the confounding practices. However, to facilitate bulk comparison, we use two common numeric measures of balance: standardized difference of means and ratio of variances [135]. The former is

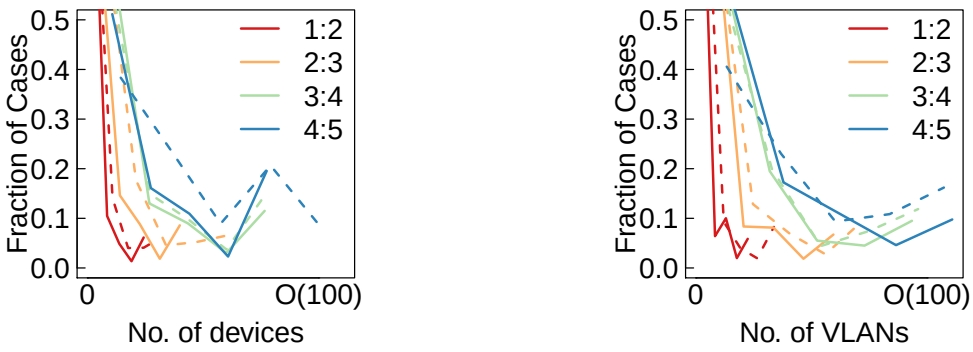


Figure 2.7: Visual equivalence of confounding practice distributions: lines of the same color are for the same comparison point; solid lines are for matched untreated cases and dashed lines are for matched treated cases

computed as $\frac{\bar{Z}_T - \bar{Z}_U}{\sigma_T}$, where $\bar{Z}_T$ and $\bar{Z}_U$ are the means of a confounding practice Z for the matched treated and matched untreated cases, respectively, and σ_T and σ_U are the standard deviations. The ratio of variances is computed as σ_T^2/σ_U^2. For each confounding practice, the absolute standardized difference of means should be less than 0.25 and the variance ratio should be between 0.5 and 2 [135]. These equations and thresholds also apply to the propensity scores for the matched cases.

As shown in Table 2.5, the absolute standard difference of means and the ratio of variances of the propensity scores satisfy the quality thresholds for all comparison points. The same also holds for all confounding factors (not shown).

Although we consider a large set of management practices in our causal analysis, it is possible that other practices or factors also contribute to the observed outcomes. We can easily incorporate new practices into our propensity scores as we learn about them. Additionally, our matching based on propensity scores introduces some randomness that can help mitigate the effects of any unaccounted for factors. However, we can never definitely prove causality with QEDs [95]; any causal relationships identified by MPA should thus be viewed as "highly-likely" rather than "guaranteed".

4) Analyzing the statistical significance. The final step is to analyze the statistical significance of the difference in outcomes between the matched treated and untreated cases.

Comparison Point	Fewer Tickets	No Effect	More Tickets	p-value
1:2	562	350	830	1.05×10^{-12}
2:3	251	61	302	3.34×10^{-2}
3:4	110	25	160	2.80×10^{-3}
4:5	282	38	343	1.63×10^{-2}

Table 2.6: Statistical significance of outcomes: causality is deemed to exist for higlighted comparison points

For each matched pair, we compute the difference in outcome (*number of tickets*) between the treated and untreated case: $y_t - y_u$. If the result is positive (negative), then the treatment practice has led to worse (better) network health; if the result is zero, then the practice has not impacted health. We use the outcome calculations from all pairs to produce a binomial distribution of outcomes: more tickets (+1) or fewer tickets (-1). Table 2.6 shows the distribution for the matched pairs at each comparison point.

If the treatment practice impacts network health, we expect the median of the distribution to be non-zero. Thus, to establish a causal relationship, we must reject the null hypothesis H_0 that the median outcome is zero. We use the *sign test* to compute a *p-value*—the probability that H_0 is consistent with the observed results. Crucially, the sign test makes few assumptions about the nature of the distribution, and it has been shown to be well-suited for evaluating matched design experiments [79]. We choose a moderately conservative threshold of 0.001 for rejecting H_0.

Table 2.6 shows the p-value produced by the sign test for each of the comparison points for *number of change events*. We observe that the p-value is less than our threshold for the *1:2* comparison point. Hence, the difference in the *number of change events* between bins 1 and 2 is statistically significant, and a causal impact on network health exists at these values. In contrast, the results for the other comparison points (*2:3*, *3:4*, and *4:5*) are not statistically significant. This is due to either the absence of a causal relationship—i.e., increasing the *number of change events* beyond a certain level does not cause an increase in the *number of tickets*—or an insufficient number of samples. We believe the latter applies for our data,

because there is at least some evidence of a non-zero median: the number of cases with more tickets is at least 20% higher than the number of cases with fewer tickets for the *2:3*, *3:4*, and *4:5* comparison points.

Results for the OSP. We now conduct a causal analysis for the 10 management practices with the highest statistical dependence with network health (Table 2.3). Due to skew in our data, we can only draw meaningful conclusions for low values of our practice metrics (bins 1 and 2).

Table 2.7 shows the p-value for the comparison between the first and second bin for each practice. We observe that 8 of the 10 practices have a causal relationship according to our p-value threshold. In fact, the p-values for these practices are well below our chosen threshold (0.001). The strongest evidence of a causal relationship exists for *number of change types*, *number of change events*, and *number of device types*.

Several of the practices with a causal relationship, including *number of devices* and *average devices changed per event*, are practices for which operators who responded to our survey had mixed opinions regarding their impact (Figure 1.1). Our analysis also matches the prevailing opinion that *number of change events* has a high impact on health, and, to some extent, discredits the belief that the *fraction of events with ACL changes* has low impact.

For the remaining two metrics, *intra-device complexity* and *fraction of events with an interface change*, there is not enough evidence to support a causal relationship. The high statistical dependence but lack of a causal relationship is likely due to these practices being affected by other practices which do have a causal relationship with network health. For example, *number of VLANs* has a causal relationship with network health and may influence *intra-device complexity*. Hence, a change in *number of VLANs* may change both network health and *intra-device complexity* in a way that makes *intra-device complexity* statistically similar to health.

Table 2.8 shows the p-value for the comparison between the upper bins for the same 10 practices. We observe that over one-third of the matchings have poor quality (i.e.,

Treatment Practice	p-value for 1:2
No. of devices	1.92×10^{-8}
No. of change events	1.05×10^{-12}
Intra-device complexity	1.53×10^{-2}
No. of change types	5.75×10^{-12}
No. of VLANs	6.46×10^{-6}
No. of models	1.31×10^{-7}
No. of device types	2.99×10^{-10}
Avg. devices changed per event	3.56×10^{-8}
Frac. events w/ interface change	5.27×10^{-3}
Frac. events w/ ACL change	9.10×10^{-9}

Table 2.7: Causal analysis results for the first and second bin for the top 10 statistically dependent management practices: highlighted p-values satisfy our significance threshold

Treatment Practice	Comparison Point 2:3	3:4	4:5
No. of devices	Imbal.	Imbal.	Imbal.
No. of change events	3.34×10^{-2}	2.80×10^{-3}	1.63×10^{-2}
Intra-device complexity	Imbal.	1.71×10^{-1}	1.47×10^{-1}
No. of change types	9.02×10^{-1}	1.42×10^{-5}	Imbal.
No. of VLANs	Imbal.	1.94×10^{-3}	Imbal.
No. of models	Imbal.	Imbal.	Imbal.
No. of device types	Imbal.	6.63×10^{-1}	Imbal.
Avg. devices changed per event	4.53×10^{-3}	2.25×10^{-1}	Imbal.
Frac. events w/ interface change	4.51×10^{-2}	4.58×10^{-1}	2.89×10^{-12}
Frac. events w/ ACL change	4.88×10^{-2}	2.78×10^{-1}	6.48×10^{-2}

Table 2.8: Causal analysis results for upper bins for the top 10 statistically dependent management practices: highlighted p-values satisfy our significance threshold

strong imbalance), and most of the others have large p-values. This primarily stems from management practice metrics following a heavy-tailed distribution. For example, when the treatment practice is *number of devices*, 81% of cases fall into the first bin and 8% fall into the second bin; this means there are few cases from which to select matched pairs for the *2:3*, *3:4*, and *4:5* comparison points. The only way to address this issue is to obtain (more diverse) data from more networks.

2.5 Predicting Network Health

We now move on to the second goal of MPA: building models that take a set of management practices as input and predict the expected network health. Such models are useful for network operators to explore how adjustments in management practices will likely impact network health: e.g., will combining configuration changes into fewer, larger changes improve network health?

We find that basic learning algorithms (e.g., C4.5 [117]) produce mediocre models because of the skewed nature of management practices and health outcomes. In particular, they over-fit for the majority healthy network case. Thus, we develop schemes to learn more robust models despite this limitation. We show that we can predict network health at coarse granularity (i.e., healthy vs. unhealthy) with 91% accuracy; finer-grained predictions (i.e., a scale of 1 to 5) are less accurate due to a lack of sufficient samples.

2.5.1 Building an Organization's Model

We start with the following question: given all data from an organization, what is the best model we can construct?

An intuitive place to start is support vector machines (SVMs). SVMs construct a set of hyperplanes in high-dimensional space, similar to using logistic regression to construct propensity score formulas during causal analysis. However, we found the SVMs performed worse than a simple majority classifier. This is due to unhealthy cases being concentrated in a small part of the management practice space.

To better learn these unhealthy cases, we turn to decision tree classifiers (the C4.5 algorithm [117]). Decision trees are better equipped to capture the limited set of unhealthy cases, because they can model arbitrary boundaries between cases. Furthermore, they are intuitive for operators to understand.

Methodology. Prior to learning, we bin data as described in Section 2.3. However, we

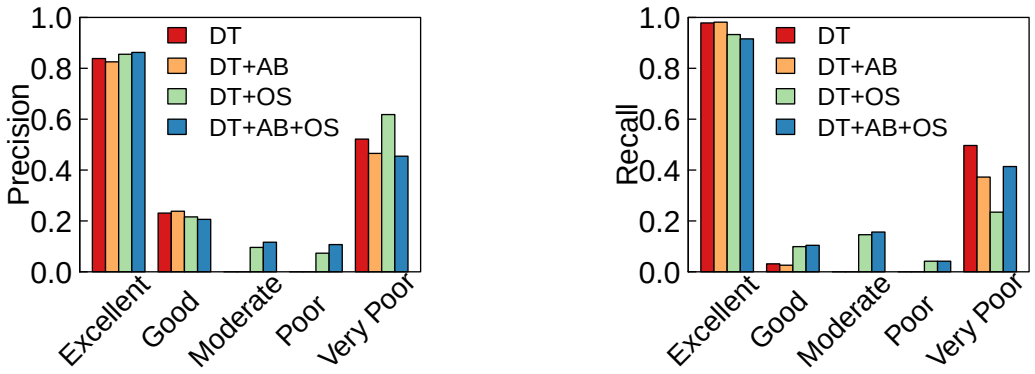
use only 5 bins for each management practice (instead of 10), because the amount of data we have is insufficient to accurately learn fine-grained models. For network health, we use either 2 bins or 5 bins; two bins (classes) enables us to differentiate coarsely between *healthy* ($\leqslant$1 tickets) and *unhealthy* networks, while five bins captures more fine-grained classes of health—excellent, good, moderate, poor, and very poor ($\leqslant$2, 3–5, 6–8, 9–11, and $\geqslant$12 tickets, respectively). As is standard practice, we prune a decision tree to avoid over-fitting: each branch where the number of data points reaching this branch is below a threshold α is replaced with a leaf whose label is the majority class among the data points reaching that leaf. We set α=1% of all data.

Model Validation. We measure the accuracy, precision, and recall of the decision trees using 5-fold cross validation. *Accuracy* is the mean fraction of test examples whose class is predicted correctly. For a given class C, *precision* measures what fraction of the data points that were predicted as class C actually belong to class C, while *recall* measures what fraction of the data points that belong to class C are correctly predicted as class C.

We find that a 2-class model performs very well. The accuracy of the pruned decision tree is 91.6%. In comparison, a majority class predictor has a significantly worse accuracy: 64.8%. Furthermore, the decision tree has very high precision and recall for the healthy class (0.92 and 0.98, respectively), and moderate precision and recall for the unhealthy class (0.62 and 0.31, respectively). A majority class predictor has only moderate precision (0.64) for the healthy class and no precision or recall for the unhealthy class.

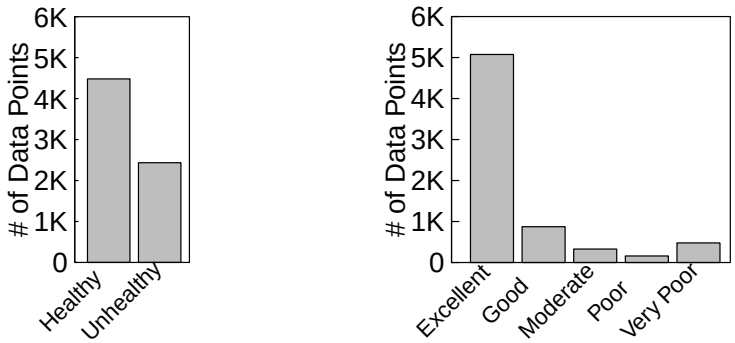
The accuracy for a 5-class model is 81.1%, but the precision and recall for the intermediate classes (good, moderate, and poor) are very low (DT bars in Figure 2.8). The root cause here is skew in the data: as shown in Figure 2.9b, a majority of the samples represent the "excellent health" case (73%), with far fewer samples in other health classes (e.g., the poor class has just 2.3% of the samples). Our 5-class decision tree ends up overfitting for the majority class.

Addressing Skew. Because networks are generally healthy, such skew in data is a funda-



(a) Precision (b) Recall

Figure 2.8: Accuracy of 5-class models (DT=standard decision tree learning algorithm, AB=AdaBoost, OS=oversampling)



(a) 2 classes (b) 5 classes

Figure 2.9: Health class distribution

mental challenge that predictive models in MPA need to address, especially when attempting to predict fine-grained health classes. To address skew and improve the accuracy of our models for minority classes, we borrow two techniques from the machine learning community: boosting (specifically, AdaBoost [67]) and oversampling.[4]

AdaBoost helps improve the accuracy of "weak" learners. Over many iterations (we use 15) AdaBoost increases (decreases) the weight of examples that were classified incorrectly (correctly) by the learner; the final learner (i.e., decision tree) is built from the last iteration's weighted examples. Oversampling directly addresses skew as it repeats the minority class

[4]We also experimented with random forests [46, 90]; neither balanced [46] nor weighted random forests [90] improve the accuracy for the minority classes beyond the improvements we are already able to achieve with boosting and oversampling.

examples during training. When building a 2-class model we replicate samples from the unhealthy class twice, and when building a 5-class model we replicate samples from the poor class twice and the moderate and good classes thrice.

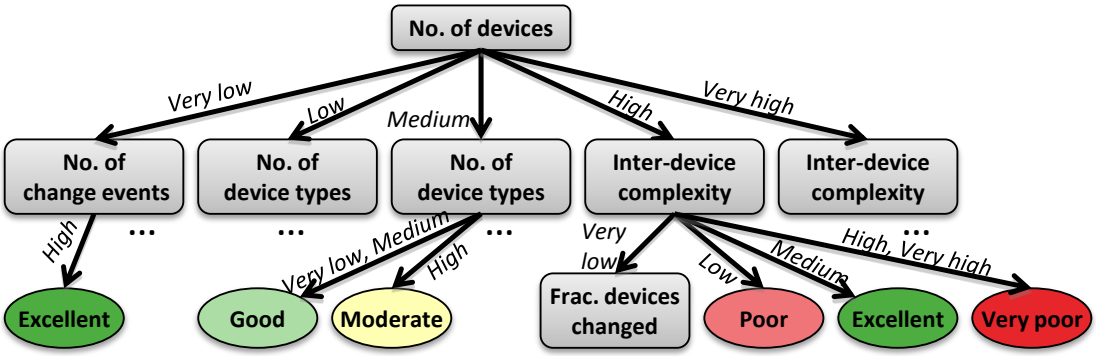
The results from applying these enhancements are shown in Figure 2.8. We observe that AdaBoost results in minor improvement for all classes. In contrast, using oversampling significantly improves the precision and recall for the three intermediate health classes, and causes a slight drop in the recall for the two extreme classes (excellent and very poor). Using oversampling and AdaBoost in combination offers the best overall performance across all classes.

The final 5-class model is *substantially better than using a traditional decision tree*. However, it is still sub-optimal due to the significant skew in the underlying dataset. Separating apart a pair of nearby classes whose class boundaries are very close—e.g., excellent and good—requires many more *real* data points from either class; oversampling can only help so much. Thus, lack of data may pose a key barrier to MPA's ability to model network health at *fine granularity*. Nonetheless, we have shown that *good models can be constructed for coarse grained prediction*.

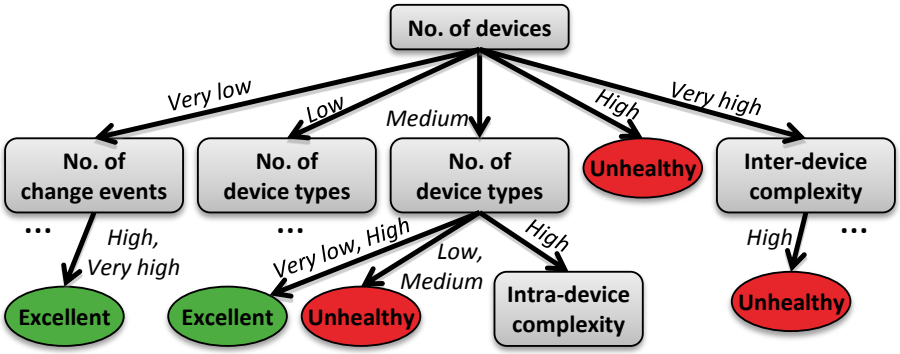
2.5.2 Using an Organization's Model

Operators can use an organization's model to determine which combinations of management practices lead to an (un)healthy network, and to evaluate how healthy a network will be in the future when a specific set of management practices are applied.

Tree Structure. Figure 2.10a shows a portion of the best 5-class tree. Since decision trees are built by recursively selecting the node with the highest mutual information, the management practice with the strongest statistical dependence (identified in Section 2.3)—*number of devices*—is the root of the tree. In the second level, however, two of the three practices are not present in our list of the 10 most statistically dependent practices (Table 2.3). This shows that the importance of some management practices depends on the nature of



(a) 5-class



(b) 2-class

Figure 2.10: Decision trees (only a portion is shown)

other practices: e.g., when the *number of devices* is medium or low, the *number of device types* (i.e., the presence of middleboxes) is a stronger determinant of health than the *number of change events*. Thus, examining the paths from the decision tree's root to its leaves provides valuable insights into which combinations of management practices lead to an (un)healthy network. The same observations apply to the 2-class tree (Figure 2.10b).

Predicting Future Health. We now show that an organization's model can accurately predict the future health of an organization's networks. In particular, we build decision trees using data points from M months ($t - M$ to $t - 1$). Then, we use management practice metrics from month t to predict the health of each network in month t. The accuracy for month t is the fraction of networks whose health was correctly predicted.

Table 2.9 shows the average accuracy for M=1, 3, 6, and 9 for values of t between February and October 2014. We observe that *a 2-class model has consistently high prediction*

M (months)	5 classes	2 classes
1	0.734	0.881
3	0.756	0.893
6	0.779	0.901
9	0.779	0.903

Table 2.9: Accuracy of future health predictions

accuracy of 89% irrespective of the amount of prior data used to train the model. This trend primarily stems from having less severe skew between the majority and minority classes when using two classes (Figure 2.9a).

The prediction accuracy of a 5-class model reaches 78% for $\mathsf{M} = 9$. Also, accuracy improves with a longer history of training data: using 9 months, rather than 1 month, of training data results in a 5% increase in accuracy. However, as the amount of training data increases (i.e., increasing M) the relative improvement in accuracy diminishes. Thus, a reasonably accurate prediction of network health can be made with less than a year's worth of data.

2.6 Limitations

Generality. While the observations we make for the OSP's networks provide a valuable perspective on the relationship between management practices and network health, the statistical dependencies and causal relationships we uncover may not apply to all organizations. Differences in network types (e.g., data center vs. wide area networks), workloads, and other organization-specific factors may affect these relationships. Nonetheless, our techniques are likely generally applicable, and any organization can run our tool [18] to discover these relationships for its networks.

Intent of Management Practices. The metrics we infer (Section 2.1.2) quantify management practices in terms of their direct influence on networks' physical infrastructure and data and control planes: e.g., how heterogeneous is network hardware, and which configuration stanzas are changed. However, we could also quantify management actions in terms of

their *intent*, or the goal an operator is trying to achieve: e.g., an operator may want to reduce firmware licensing costs, so they design a network to use RIP rather than OSPF [40]. By analyzing the relationships between intent and network health, we can gain a richer understanding of what practices are the most problematic. Unfortunately, intent is much more difficult to infer from network data sources (Section 2.1.1), and doing so is part of our future work (Section 5.2).

2.7 Related Work

Analyzing network failures. Prior work has examined network failures in great detail. For example, Turner et al. use device logs, network probes, and incident reports to understand the causes, frequency, and impact of failures in enterprise campus networks [139]. Turner et al. also examine how to combine various data sources to obtain a better view of failures and their root causes in regional backbone networks [140, 141].

In the context of data centers, Gill et al. [75] study the frequency, causes, and impact of link and device failures in data center networks; Benson et al. [42] and Navendu et al. [114] study such failures specifically in cloud data centers. Navendu et al. [113] also provide a detailed characterization of middlebox failures in data centers. The latter two studies use output from NetSieve, which uses natural language processing to analyze the free-form text of network trouble tickets and generate a synopsis of the problem, troubleshooting steps, and resolution actions [115].

While these studies provide valuable insights into network failures, they do not link failures back to the high-level design and operational practices employed by network operators, nor do not provide solutions to mitigate failures. However, some of the data sources and techniques considered in these studies could be useful for deriving better network health metrics that could then improve the usefulness of MPA.

General best practices. Establishing, following, and refining management practices is

an important part of information technology (IT) service management. ITIL [14] provides guidance on: service design, which focuses on health-related concerns such as availability and service levels; service transition, which focuses on change, configuration, and deployment management; and continual service improvement. Some of the general metrics used in ITIL to asses the health of an IT organization (e.g., *number of changes*) are also used in MPA (Section 2.1.2), but MPA considers many networking-specific metrics as well. The major steps in MPA—defining metrics, characterizing practices, and uncovering relationships between practices and health—are similar to the steps employed in security management [81]. However, MPA's analyses are focused more on causality and prediction (Sections 2.4 and 2.5).

Configuration management practices. Several studies have examined network management practices from the perspective of device configurations. Kim et al. study several configuration-related design and operational issues in two university networks: e.g., how network-wide configuration size grows over time, what causes this growth, how configurations of different device types (e.g., router, firewall, etc.) change and why, and the qualitative differences among the campuses in these aspects [92]. Others have looked at more narrow aspects of configuration practices: e.g., Benson et al. examine configuration complexity in seven enterprise networks [40] and study the design and change patterns of various network-based services of a large ISP [41]; Garimella et al. and Krothapalli et al. study VLAN usage in a single university network [69, 97]. In contrast to these prior works, MPA considers a much more comprehensive set of design and operational practices. Also, by virtue of studying hundreds of data center networks operated by a large online service provider, we are able to provide a unique view into the *variation* in data center network management practices. Finally, none of these prior studies tie their the observations to network failures.

Analysis techniques. Our use of quasi-experimental designs (QEDs) in MPA (Section 2.4) is inspired by recent network measurement studies focused on video streaming quality [95] and video ad placement [96]. While these works use exact matching in their QEDs, we use nearest neighbor matching of propensity scores, because exact matching cannot accommodate

the large number of confounding factors in the management plane.

We use standard machine learning algorithms, e.g., C4.5 [117] and AdaBoost [67], for producing a predictive model of network problems given a specific set of management practices (Section 2.5). Machine learning has previously been used in networking for task such as predicting the quality of experience for streaming video [38, 133] and classifying traffic (as malicious) [101, 132]. However, MPA is one of the first contributions to the networking community that leverages machine learning for improving network management.

Relation to other subfields of computer science. MPA as a whole is inspired by research into software engineering practices, also called "empirical software engineering", which has helped improve the quality of software and reduced the number of bugs [43]. We expect similar positive impact from MPA.

2.8 Summary

This chapter introduced the *first framework for systematically analyzing how an organization's network management practices impact the likelihood of network failures.* By studying the management practices employed in over 850 data center networks operated by a large online service provider (OSP), we showed that such a framework is both: (1) necessary—the diversity we found in the management practices used in the OSP's networks, in combination with our operator survey (Section 1.1), illustrates that the networking community has a limited understanding of which management practices are "best"; and (2) feasible—through the use of carefully selected analysis and learning techniques, including mutual information, propensity score matching, boosting, and oversampling, we are able to overcome the challenges introduced by the sometimes non-linear, overlapping, and skewed relationships between management practices and network health.

More importantly, the application of MPA to the OSP's data center networks revealed several management practices that strongly influence the number of problems these networks experience. In particular, there is strong statistical and causal evidence that the number of

devices, the number and type of configuration changes, and the number of device types (i.e., the presence of middleboxes) impacts network health. The first and last of these practices are difficult to change: the number of devices is largely dictated by the number of end hosts the network must support, and various types of middleboxes are essential for meeting the security and efficiency needs of applications. The number and diversity of configuration changes may be easier to reduce, but some configuration changes are inevitable, so the risk of misconfiguration-induced failures is still non-negligible. Thus, it is important for data center network operators to have well-designed network verification and middlebox management frameworks at their disposal.

Unfortunately, many tools for verifying network functionality (e.g., reachability) [66, 87, 88, 91, 105] and managing the performance and availability of middleboxes [70, 111, 116] are inefficient, because they operate at a low-level of abstraction. Therefore, we design and enable new abstractions that help network operators ensure routers and middleboxes—the two crucial components of data center networks—function correctly and perform well. We motivate and discuss these abstractions in detail in the next two chapters.

3 CONTROL PLANE CHECKING USING ARC

In the previous chapter, we showed that the frequency of configuration changes in data center networks has a strong impact on the number of problems these networks experience. This corroborates the widely held belief that control plane configurations are often buggy [64, 150]. In some cases, the problems arising from configuration bugs may occur immediately after a new control plane configuration is applied (and routing has re-converged). In other cases, bugs may only become apparent when links or devices fail: e.g., in 2012, failure of a router in a Microsoft Azure data center triggered previously unknown configuration errors on other devices, degrading service in the West Europe region for several hours [138]. Thus, it is desirable to detect control plane configuration bugs *before* they impact network security and availability, lest end hosts and applications be susceptible to attacks or experience failures due to an inability to communicate.

Unfortunately, many network verification tools [87, 88, 91, 105] analyze a network's current data plane. This limits the scope of their analyses to the current live network and prevents them from being used for proactive analysis. To overcome this limitation, more recent tools [66] simulate the control plane and generate the network's expected data plane under specific failure scenarios, e.g, a single link failing. However, these tools operate at a low level of abstraction, modeling individual protocol message exchanges to generate the data plane. Furthermore, these tools must generate the complete data plane for every possible failure scenario of interest. Consequently, they tend to be slow and impractical for proactively verifying important security and availability invariants under *arbitrary failures*. Ideally, a network operator should be able to verify every configuration change—up to thousands of changes per month (Section 2.2)—before they are applied to the network. This requires a method of verification that takes at most a few minutes to check important functional invariants, such as those in Table 3.1, for an entire data center network.

In this chapter, we present an *abstract representation for control planes* (ARC) that

Invariant	Example
I1: Always blocked	External hosts can never communicate with subnet S
I2: Always reachable with $< k$ infrastructure faults	Up to 5 links can fail without breaking connectivity between subnets S_1 and S_2
I3: Always isolated	Traffic between subnets S_1 & S_2 and S_3 & S_4 never traverses the same link simultaneously
I4: Always traverse a middlebox	Traffic between external hosts and internal hosts must always traverse a firewall
I5: Equivalent(C_1, C_2)	Traffic between hosts must always traverse the same paths if control plane C_2 were to replace C_1

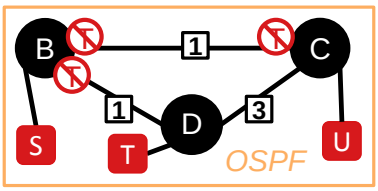
Table 3.1: Invariants of common interest

models a data center network's forwarding behavior at a higher level than today's network verifiers, thus enabling such efficient analysis. This is made possible by two key factors: (1) proactive control plane analysis tasks often require computing *properties* of paths, not the paths themselves—e.g., invariants I1–I4 in Table 3.1 focus on the existence (or absence) of paths; and (2) data center networks tend to use only a handful of routing protocols (Section 2.2) which interact in very specific ways (Section 3.6).

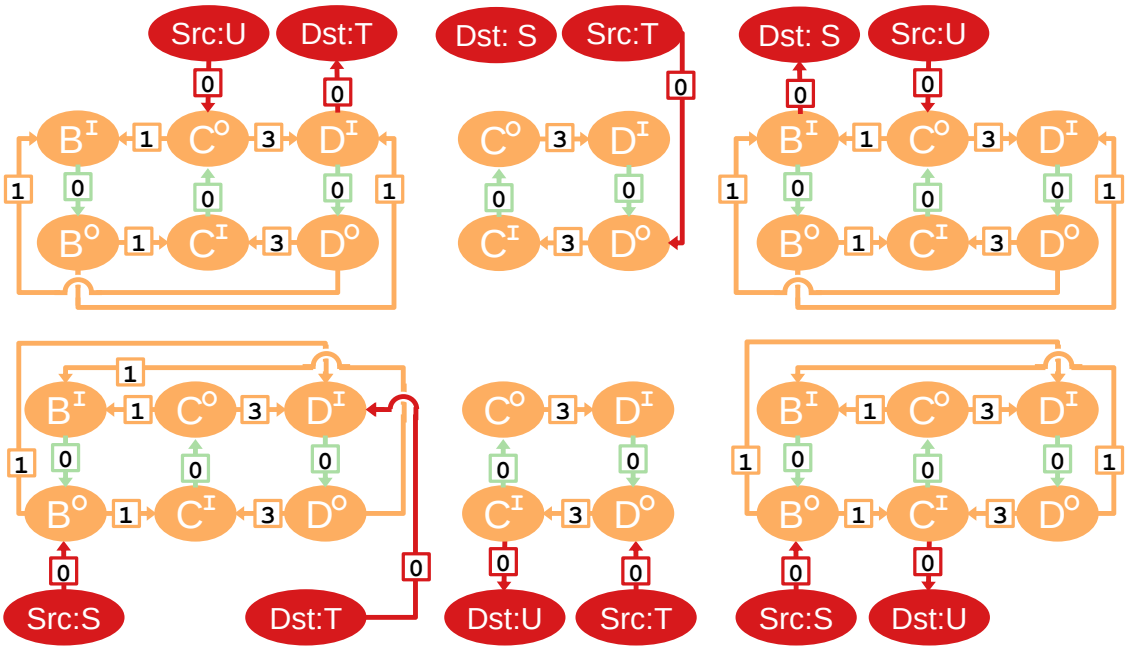
We begin this chapter with an overview of the key requirements (Section 3.1) and challenges (Section 3.2) in designing ARC. We then present algorithms for constructing a network's ARC (Sections 3.3 and 3.4); in Appendix A, we formally prove the resulting ARC satisfies our requirements. We then discuss how ARC can be used to efficiently check important invariants (Section 3.5), and we conclude (Section 3.6) by evaluating ARC's efficiency using control plane configuration snapshots from a subset of the OSP's networks we analyzed in Chapter 2.

3.1 Important Attributes of ARC

A network's ARC consists of a collection of *weighted digraphs*, with one digraph for each "traffic class" (i.e., source-destination subnet pair). As an example, Figure 3.1b shows the ARC for the simple control plane in Figure 3.1a; the ARC has six graphs: one for each possible combination of source and destination subnets. Each digraph models the behavior



(a) Control plane for a network with three subnets (squares) and three routers (circles) participating in a single OSPF instance (rectangle); no-entry symbols indicate inbound ACLs on traffic from T



(b) Abstract representation for the control plane (ARC): it contains one digraph for every pair of source and destination subnets; vertices correspond to routing processes (two per process for reasons described in Section 3.3); edges represent the possible flow of data traffic enabled by the exchange of routing information between the connected processes

Figure 3.1: Example network with a single OSPF instance and its ARC

of the routing instances/protocols in the control plane, and the interactions among them, with respect to the corresponding traffic class.

In order to check important invariants, such as those listed in Table 3.1, using ARC, the constituent digraphs must posses two key attributes: *comprehensiveness* and *precision*.

Graph comprehensiveness. Each digraph in an ARC is constructed such that it contains every path between the source and destination endpoints that is used in the real network, and does not contain any paths that are infeasible in the real network, under arbitrary infrastructure faults. We say such a digraph is *comprehensive*, because it encodes all possible,

and no impossible, forwarding behaviors.

With a comprehensive ARC, verifying the invariants I1–I4 in Table 3.1 for arbitrary link failures boils down to *checking simple graph-level attributes*. For example, assume the graphs in Figure 3.1b are comprehensive. Suppose we want to verify that "subnet T can never send traffic to subnets S or U under any link failure scenario" in the network shown in Figure 3.1a. This can be done by checking if T and S (or T and U) are in separate connected components of the graphs for the corresponding traffic classes (the upper-center and lower-center graphs in Figure 3.1b). Because T and U are in the same connected component in the upper-center graph, there is *some* link failure scenario in which the invariant is violated and T can send traffic to U (e.g., when the B − D link fails).

Graph precision. To aid operators in debugging violations, and allow for fast equivalence testing, the edge weights in each digraph are assigned such that, after removing edges corresponding to arbitrary failed links, the *min-cost path* in the digraph between the source and destination vertices is the exact path taken in the real network. We say such a graph is *precise*, because it encodes the network's actual forwarding behavior under arbitrary link failures.

For example, when there are no link failures in the network in Figure 3.1a, traffic from S to U takes the path S → B → C → U, which is the min-cost path in the lower-right graph in Figure 3.1b. When the B−C link fails, the actual and min-cost path is S → B → D → C → U. While in this example edge weights are the same as OSPF cost metrics, in a real ARC the weights are a function of the relative rank of specific routing protocols, AS paths, and network links.

When the digraph is precise, we can produce all min-cost paths[1] from T to U as counter-examples to the aforementioned invariant. The operator can use this to add the missing ACL to C and prevent T and U from ever communicating. Additionally, we can check the equivalence of two control planes by directly comparing the graphs contained in their ARC.

[1]By producing *all* min-cost paths we can capture the effects of multipath routing.

If each graph in each control plane's ARC has the same vertices and edges, and the edge weights are proportional, then the control planes are equivalent.

3.2 Challenges in Generating a Network's ARC

The main challenge in constructing a network's ARC is determining the appropriate vertices, edges, and weights to use for the constituent graphs to ensure they are comprehensive and precise.

Modeling the collective behavior of multiple routing instances in a series of weighted digraphs in the ARC is enabled by the fact that *most routing protocols used in today's data centers employ a cost-based path selection algorithm.* For example, OSPF uses Dijkstra's algorithm to compute min-cost paths from a source to all destinations; RIP computes shortest paths using the Bellman-Ford algorithm. BGP associates cost labels with paths based on numeric metrics: e.g., operator-defined local preference, path length, and multi-exit discriminator (MED) [4, 85]. These have similar properties to link costs used in IGPs, except BGP costs are per-path rather than per-link.

While these similarities allow us to use weighted digraphs to model routing behavior, differences between protocols introduce at least two challenges:

1. In the actual control plane, interior and exterior gateway protocols (IGPs and EGPs, respectively) compute routes at different *granularities.* An IGP treats each router as a node, while an EGP views each AS as a node.
2. Each routing protocol uses a different *currency* for expressing link and path costs/preferences: e.g., a link with an OSPF cost of 1 may be less desirable than an AS path whose local preference is 1, or vice versa. Thus, we cannot directly add or compare costs between protocols.

There are other subtle aspects of network control planes that also impact our modeling:

- *Traffic-class-specific policies.* Only certain classes of traffic are blocked by a data/control

plane ACL.

- *Redistribution of routes between routing instances.* A routing process may advertise routes computed by another routing instance, allowing traffic to traverse a path composed of segments selected by different protocols.
- *Selection of routes based on AD.* When multiple routing processes on the same device identify a route to a destination, only the route from the process with the lowest administrative distance (AD) is installed in the device's global routing information base (RIB) [99].

In the next two sections, we describe how we structure and generate the ARC's digraphs to accommodate the above issues. We focus on networks that use OSPF, RIP, eBGP (with internal ASes), static routes, AD-based route selection, route redistribution, data plane ACLs, and/or route filters. These are the constructs we find throughout the hundreds of data center networks we study in Section 3.6, allowing us to produce comprehensive ARCs for all of these networks and precise ARCs for 97% of the networks. A subset of these same features are used in Facebook's data centers [37] and our university data center. We do not handle OSPF areas, BGP local preferences, circular route redistribution, or other routing protocols. However, we believe our algorithms (Sections 3.3 and 3.4) could be extended to support OSPF areas and other routing protocols that employ a min-cost path selection algorithm (e.g., EIGRP).

3.3 Extended Topology Graphs

A network's physical topology may seem like a natural starting point for the ARC's graphs. By having a vertex for each router and an edge for each physical link, we can assign edge weights based on the per-interface cost metrics defined for IGPs (e.g., OSPF and RIP) and the AS preferences defined for BGP. However, this is too coarse to express route selection and redistribution policies between routing processes running on the same device.

To accommodate these features, we introduce an abstraction called an *extended topology graph* (ETG). Figure 3.2b shows the ETG for the example control plane depicted in Figure 3.2a. Vertices in the ETG correspond to individual routing processes.[2] Directed edges represent inter- and intra-device communication paths between routing processes. These include *hardware paths*—a single physical link or multiple physical links that form a layer-2 network—and *software paths*—inter-process communication channels used to exchange information between processes on the same device.

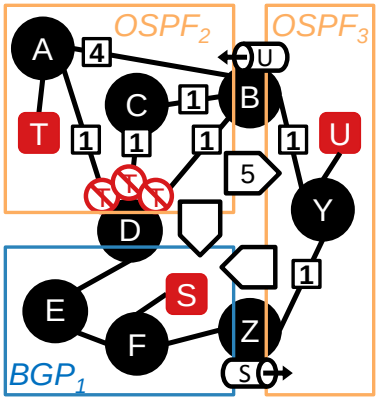
Some aspects of a network's routing control plane only apply to specific traffic classes: e.g., data plane ACLs, route filters, and static routes. To accommodate these features, an ARC includes a *customized ETG for each traffic class*. As mentioned earlier, a traffic class represents the set of traffic flowing from one *endpoint group*—a set of related hosts, subnets, etc.—to another. We use the network prefixes in device configurations, including prefixes assigned to interfaces, advertised by routing processes, and referenced in ACLs, as the basis for determining a network's endpoint groups. Because some prefixes may overlap, we use standard firewall rule optimization algorithms [68] to compute a set of non-overlapping prefixes. We generate a list of traffic classes by enumerating all possible pairings of prefixes.

Modeling forwarding behavior at the level of routing processes results in an ARC that is not protocol-independent. This model is nevertheless useful to answer control plane verification questions. In Section 3.5.2, we show how to transform an ETG from a process-based to an interface-based model, resulting in a protocol-independent ARC that can be useful for equivalence testing.

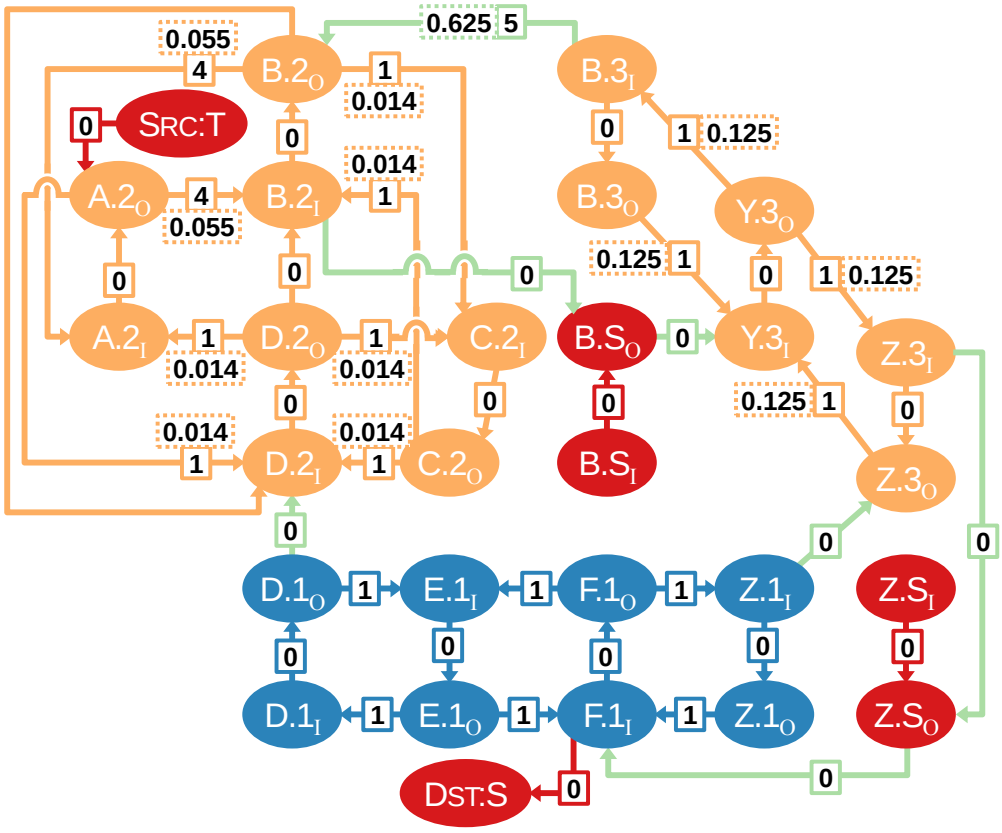
3.3.1 Constructing ETGs

Vertices. The ETG contains two vertices (*in* and *out*) for each routing process. For example, the processes on routers B, Y, and Z for routing instance $OSPF_3$ in Figure 3.2a are represented by vertices $\mathsf{B}.3_{\mathrm{I}}$, $\mathsf{B}.3_{\mathrm{O}}$, $\mathsf{Y}.3_{\mathrm{I}}$, $\mathsf{Y}.3_{\mathrm{O}}$, $\mathsf{Z}.3_{\mathrm{I}}$, and $\mathsf{Z}.3_{\mathrm{O}}$ in Figure 3.2b. We use two

[2]Static routes are also viewed as a routing process.



(a) Control plane: circles represent routers and rectangles represent routing instances; links between routers are labeled with OSPF costs; no-entry symbols represent ACLs blocking traffic destined for T; arrows between routing instances indicate route redistribution, and specify the cost assigned to such routes; tubes with arrows represent static routes that are redistributed in the direction of the arrow



(b) Extended topology graph (ETG) for the for the T → S traffic class: the structure and weights are the same for all traffic classes, with the exception of endpoint edges, edges removed due to ACLs, and static route vertices; weights in dashed boxes are assigned by our scaling algorithm to model the fixed costs assigned to redistributed routes.

Figure 3.2: Example feature-rich network and its ETG

vertices per process in order to accommodate route selection and redistribution (described in detail below). We identify a network's routing processes from the `router` stanzas in device configurations [106].

We also add special source and destination vertices (SRC and DST, respectively) to the ETG to represent the source and destination endpoints associated with the traffic class.

Inter-device edges. The *out* vertex for a routing process on one device is connected to the *in* vertex for a process on another device if: (i) the two devices are connected by a (sequence of) physical link(s),[3] and (ii) the routing processes participate in the same routing instance. Such an *inter-device edge* thus represents two things. First, it represents the direct exchange of routing information (e.g., link-state updates or AS-level path advertisements) within a routing instance. Second, it represents a possible physical path over which data traffic may be forwarded due to the RIB entries resulting from the aforementioned exchange of routing information.

For example, in the network shown in Figure 3.2a, the BGP_1 routing process on router E may compute a route to the subnet S via router F as a result of routing information sent by the BGP_1 process on router F. The flow of routing information from F to E and the resulting flow of data traffic from E to F is represented by the edge from $\mathsf{E}.1_{\mathrm{O}}$ to $\mathsf{F}.1_{\mathrm{I}}$ in Figure 3.2b. There is a similar edge from $\mathsf{F}.1_{\mathrm{O}}$ to $\mathsf{E}.1_{\mathrm{I}}$, because routing information also flows from E to F and may result in the flow of data traffic from F to E. Note that inter-device edges always go from an *out* vertex to an *in* vertex and point in the direction data traffic flows, which is the inverse of the direction routing information flows.

Unlike routes computed by IGP and BGP processes, static routes are not based on advertisements from a specific neighboring process. Thus, we connect a static route's *out* vertex to the *in* vertices for *all processes on the next hop device*—i.e., the device with an interface whose IP address matches the next hop IP specified in the static route.

Intra-device edges. The ETG also contains edges between the vertices associated with

[3]We assume two devices are physically connected if they each have an interface that participates in the same subnet [106].

routing processes running on the same device. A routing process's *in* vertex is connected to: (i) the process's *out* vertex, and (ii) the *out* vertex of any other process on the device that redistributes routes into the process. Such *intra-device edges* represent the exchange of routing information and the flow of data traffic resulting from route computation within a process and route redistribution between processes, respectively. For example, the route computation within the routing process for $OSPF_3$ on B is represented by the edge from $\mathtt{B}.3_{\mathrm{I}}$ to $\mathtt{B}.3_{\mathrm{O}}$ in Figure 3.2b, and the redistribution of routes from routing instance $OSPF_2$ to $OSPF_3$ is represented by the edge from $\mathtt{B}.3_{\mathrm{I}}$ to $\mathtt{B}.2_{\mathrm{O}}$. As above, intra-device edges point in the opposite direction that routing information flows.

Endpoint edges. Edges are added from the SRC vertex to a routing process's *out* vertex if the device on which the process runs can be directly reached by the source endpoint(s) using layer-2 forwarding: e.g., $\textsc{Src} \rightarrow \mathtt{A}.2_{\mathrm{O}}$ in Figure 3.2b. Similarly, edges are added from a routing process's *in* vertex to the DST vertex if the device on which the process runs can directly reach the destination endpoint(s): e.g., $\mathtt{F}.1_{\mathrm{I}} \rightarrow \textsc{Dst}$. For traffic classes whose source endpoint is external, we add an edge from SRC to the *out* vertices of all processes that send external route advertisements; we add similar edges for external destinations.

Factoring in ACLs and route filters. Data plane ACLs prevent particular classes of traffic from entering or leaving a router. Similarly, route filters prevent a routing process from advertising particular prefixes to a process on another device, or a process on the same device through route redistribution.

To account for these filtering mechanisms, we prune some edges from the ETG. In particular, we prune an inter-device edge if: (i) there is an outgoing or incoming data plane ACL configured on the interfaces associated with the physical link(s) the edge represents, and (ii) the ACL blocks the traffic class associated with the ETG. We also prune an inter-device edge if a route filter that blocks the traffic class's destination prefix has been applied to the process whose *out* vertex is incident with the edge. Similarly, we prune an intra-device edge if a route filter that blocks the traffic class's destination prefix is applied to routes redistributed

by the process whose *out* vertex is incident with the edge.

3.4 Computing ETG Edge Weights

While comprehensive ETGs are sufficient for verifying many important invariants, such as I1–I4 in Table 3.1, precise ETGs are required for generating counterexamples or testing equivalence (I5). The key challenge in constructing precise ETGs is determining the appropriate edge weights such that the min-cost path through the ETG matches the actual path taken in the network under arbitrary infrastructure faults. Next, we describe how to assign such weights to different types of edges. In Appendix A, we prove the resulting ETGs are precise.

3.4.1 Endpoint edges

When assigning weights to endpoint edges, we must consider the route selection policies of the devices to which the source and destination endpoints are connected.

Source edges. When the source is connected to a device with *one routing process*, then the best route (if any) computed by that process is always used. Thus, the edge from Src to the process's *out* vertex is assigned a weight of 0.

If the device has *multiple routing processes*, then a route computed by a process with a lower AD is preferred over a route computed by a process with a higher AD. We model this by assigning edge weights proportional to a process's AD. The weight of an edge from Src to $r.i_{\mathrm{I}}$ is set to

$$\mathsf{AD}_i * \max_{i' \in I_r} \left(\sum_{e \in E_{i'}} w_e \right) \tag{3.1}$$

where I_r is the set of routing instances in which router r participates, $E_{i'}$ is the set of edges originating from the *in* and *out* vertices for the routing processes in instance i', and w_e is the weight assigned to edge e. This ensures the cost of a path originating at a process with a higher AD is always more expensive than the longest possible path through a routing instance whose process has a lower AD.

Destination edges. When the destination is directly connected to a device, the device always sends traffic directly to the destination; no other route is ever preferred. Thus, edges to Dst are assigned a weight of 0.

3.4.2 Inter-device edges for IGPs

For inter-device edges connecting RIP or single-area OSPF processes, we directly assign the cost metric specified in the device configurations. If no cost is explicitly defined, we assign the DEFAULT-RIP-COST or DEFAULT-OSPF-COST defined by the device vendor. For example, edges $\mathsf{A}.2_{\mathrm{O}} \rightarrow \mathsf{B}.2_{\mathrm{I}}$ and $\mathsf{B}.2_{\mathrm{O}} \rightarrow \mathsf{A}.2_{\mathrm{I}}$ in Figure 3.2b are assigned the OSPF cost configured on the $\mathsf{A} - \mathsf{B}$ link in Figure 3.2a.

3.4.3 Inter-device edges for eBGP

eBGP processes inside the network. We model the primary path selection criterion used by eBGP for computing paths: AS path length. In the absence of iBGP (which we show in Section 3.6 is not used in the networks we study), each autonomous system (AS) can only have a single eBGP speaker (i.e., process) that is directly connected to the eBGP speakers of neighboring ASes.[4] Thus, the length of an AS path is simply the number of eBGP processes traversed. We capture this by assigning a weight of 1 to inter-device edges connecting eBGP processes. For example, edges $\mathsf{F}.1_{\mathrm{O}} \rightarrow \mathsf{E}.1_{\mathrm{I}}$ and $\mathsf{E}.1_{\mathrm{O}} \rightarrow \mathsf{D}.1_{\mathrm{I}}$ in Figure 3.2b are assigned weight 1.

eBGP processes outside the network. We cannot precisely model paths that depend on advertisements from external eBGP processes, because we do not know what length paths will be advertised. As discussed in Section 3.3.1, if the source (destination) is external, we simply add an edge from Src to the *out* vertex (to Dst from the *in* vertex) of every eBGP

[4] An AS without iBGP can have multiple eBGP speakers, but each eBGP speaker can only compute paths through ASes with which it has a direct connection; different eBGP processes in the same AS cannot directly exchange routes.

process inside the network that peers with an eBGP process outside the network. Edge weights are assigned to these endpoint edges as described above.

3.4.4 Intra-device edges for route redistribution

As discussed in Section 3.3.1, route redistribution is modeled via intra-device edges connecting the *in* vertex of one routing process to the *out* vertex of another process. When assigning weights to these edges, we must consider: (1) the fixed costs assigned to redistributed routes—e.g., routes redistributed into an OSPF routing instance are assigned a static cost (specified in the device configuration) regardless of the path selected by the redistributing instance—and (2) the relative priority of the redistributing process compared to other routing processes running on the same device. Below, we use the terms *redistributor* and *redistributee* to refer to the routing process from which and to which routes are redistributed, respectively; we also assume the processes are associated with routing instances i' and i, respectively.

Fixed costs. Modeling the fixed costs is challenging, because paths through the ETG include the weights associated with edges from multiple routing instances, whereas in the actual network each routing instance only considers the costs associated with its own links and redistributed routes. We address this by scaling down the weights of edges in the redistributor's routing instance. We reduce the weights such that even the highest cost path through the redistributor's instance (i') is less than the minimum difference between any two paths in the redistributee's routing instance (i). For example, in Figure 3.2b the minimum difference in cost between any two paths through routing instance BGP_1 is 1, so we scale the weights of edges in instance $OSPF_3$ such that the longest path through $OSPF_3$ ($\mathsf{Z} \rightarrow \mathsf{Y} \rightarrow \mathsf{B}$) has a cost less than 1. As a result, the cost of the path through the redistributor's routing instance does not impact the selection of the path through the redistributee's instance, which mimics the behavior of the actual routing control plane.

We can recursively scale edge weights to accommodate sequences of routing instances that redistribute routes. In fact, our approach works with any network where the route

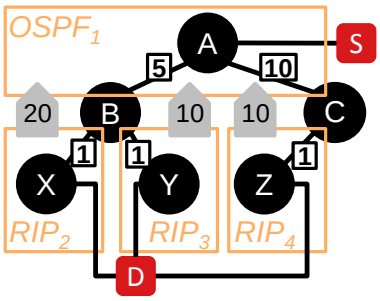


Figure 3.3: Control plane where the costs assigned to redistributed routes are incongruent with processes' ADs: ADs match the routing instance numbers; the path $\mathsf{S} \rightarrow \mathsf{A} \rightarrow \mathsf{B} \rightarrow \mathsf{Y} \rightarrow \mathsf{D}$ is the cheapest, but cannot be taken because the RIP_2 process on B has a route to D and a lower AD

redistribution policy is acyclic. The scaling factor f_i for the weights of edges in routing instance i is computed as

$$f_i = \frac{\min_{i \in I_{i'}} g_i}{1 + \sum_{e \in E_{i'}} w_e} \tag{3.2}$$

where $I_{i'}$ is the set of routing instances into which instance i' redistributes routes, g_i is the minimum difference in cost between any two paths through routing instance i, $E_{i'}$ is the set of edges originating from the *in* and *out* vertices associated with the routing processes in instance i', and w_e is the weight assigned to edge e.

Administrative distance. A route can only be redistributed when: (1) the redistributor is the only routing process on the device that has a route to the destination, or (2) the redistributor has the lowest AD among the processes that have a route to the destination. To model this behavior, we must ensure that the weight assigned to an intra-device redistribution edge is congruent with the redistributor's relative priority.

More formally, if $\mathsf{AD}_{i'} < \mathsf{AD}_{i''} < ..$, then it must be the case that $c_{i',i} < c_{i'',i} < ...$, where $c_{i',i}$ is the fixed cost assigned to routes redistributed from routing instance i' to instance i. If this does not hold, then we cannot construct a precise ETG. Note that we cannot simply increase the weight of the intra-device route redistribution edge to achieve such congruence, as this will impact the redistributee's perceived cost of a redistributed route; Figure 3.3 shows an example.

Construct	Comp.	Precise
OSPF	yes	single area
RIP	yes	yes
eBGP	yes	select only by path length
Static routes	yes	yes
ACLs	yes	yes
Route filters	yes	yes
Route redistribution	yes	acyclic + costs congruent with ADs

Table 3.2: Control plane constructs modeled in ARC

Summary. Table 3.2 summarizes the protocols and features for which an ETG is comprehensive and precise. In Appendix A, we prove that a network with any combination of these constructs results in ETGs that are comprehensive and, if the listed constraints are met, precise. In the next section, we describe how to use a network's ARC for verification and equivalence testing.

3.5 Using ARC to Check Invariants

ARC enables us to check important invariants across arbitrary failure scenarios. It is particularly well suited for verifying invariants that pertain to properties of a path. In such cases, verification/equivalence testing is a matter of (dis)proving that an ETG, or a pair of ETGs for different control planes, has a specific *graph-level* characteristic. This section describes our verification and equivalence testing algorithms that at their essence compute such graph characteristics. Furthermore, we describe how to use precise ETGs to generate counter-examples when violations occur. These help an operator take corrective actions before a buggy control plane is made "live" on the network.

3.5.1 Verifying Security/Availability Invariants

Invariants I1 to I4 in Table 3.1 can be expressed as graph characteristics that can be computed on a comprehensive ETG using polynomial-time graph traversal algorithms.

I1: Always blocked. For security reasons, an operator may want to ensure that a particular traffic class is always blocked. For this to be true under arbitrary infrastructure faults, there must not exist a path from SRC to DST in the traffic class's ETG. We can check for the existence of a path by performing a depth-first traversal of the ETG starting from SRC. If DST remains unvisited, then the property holds. Otherwise, assuming the ETG is precise, we provide the shortest path as a counterexample.

I2: Always reachable with $< \mathtt{k}$ infrastructure faults. To improve availability, an operator may want to ensure that a particular destination $\mathtt{d}$ can always be reached from a particular source $\mathtt{s}$ as long as there are fewer than $\mathtt{k}$ link failures in the network. To verify this, we can leverage properties of graph cuts. In particular, according to Menger's Theorem, the maximum number of edge-disjoint paths from $\mathtt{s}$ to $\mathtt{d}$ in a digraph equals the minimum number of edges whose removal separates $\mathtt{s}$ and $\mathtt{d}$ [30]. Thus, as long as the ETG has at least $\mathtt{k}$ edge-disjoint paths from $\mathtt{s}$ to $\mathtt{d}$, $\mathtt{d}$ will always be reachable from $\mathtt{s}$.

Finding the number of edge-disjoint paths in an arbitrary acyclic digraph is NP-Complete [129], but in a unit-weight graph the problem reduces to computing the max-flow/min-cut. Because we are only concerned with the presence of paths, and not which paths are chosen under specific infrastructure faults, we can safely convert the weight of all inter-device edges in the ETG to 1 and the weight of intra-device edges to ∞. We set the weight of intra-device edges to ∞, because we are only concerned with counting physical-link-disjoint paths, not device-disjoint paths, and a weight of ∞ allows multiple physical-link-disjoint paths to traverse the same device. We compute the max-flow/min-cut on the ETG with modified weights to identify the number of edge-disjoint paths. When the max-flow is $\geqslant \mathtt{k}+1$, the invariant is satisfied.

When the invariant is violated, we produce a counter-example set of edges that form a cut of size $\leqslant \mathtt{k}$.

I3: Always isolated. For security or performance reasons, an operator may want to ensure that two disjoint traffic classes ($\mathtt{s}_1 \rightarrow \mathtt{d}_1$ and $\mathtt{s}_2 \rightarrow \mathtt{d}_2$; $\mathtt{s}_1 \neq \mathtt{s}_2$, $\mathtt{d}_1 \neq \mathtt{d}_2$) can never

simultaneously traverse the same link. Thus, the preferred path for $s_1 \rightarrow d_1$ must never overlap with the preferred path for $s_2 \rightarrow d_2$ under any scenario. Such overlap is possible in some scenario if the ETG for $s_1 \rightarrow d_1$ has an edge in common with the ETG for $s_2 \rightarrow d_2$. An extreme scenario is where all links have failed except those used in paths that contain the common edge. The traffic isolation invariant is guaranteed to hold only if the ETGs for the two traffic classes do not have *any edges in common*, a property we can easily check. However, prior to checking this property, we recursively remove all vertices (excluding SRC and DST) whose in- or out-degree is 0; these vertices are dead-ends and can never be part of a path from SRC to DST.

If the pruned ETGs have any edges in common, we return the set of common edges as a counter-example.

I4: Always traverse a middlebox. When a network includes middleboxes, such as firewalls, an operator may want to ensure that traffic always traverses some instance of the middlebox under arbitrary infrastructure faults. To verify this, we augment the ETG to include special vertices that represent middlebox instances. Then we remove all middlebox nodes from the ETG and check if there exists a path from SRC to DST. If such a path exists, then there is some path that may be taken by the traffic that does not traverse a middlebox instance; we return this path as a counterexample.

Other invariants. Other important security and availability invariants can also be verified by computing graph-level attributes on the ARC. For example, we can verify traffic "always traverses a chain of middleboxes" by removing the vertices associated with one type of middlebox at a time, and checking if there exists a path from a vertex associated with one of the preceding middleboxes in the chain to a vertex associated with one of the following middleboxes in the chain. We can verify forwarding of particular traffic class is "always loop free" by checking that the ETG does not have a cycle containing a static route vertex and one or more vertices associated with processes in the same routing instances. We omit details for brevity.

3.5.2 Equivalence Testing

Invariant I5, equivalence, differs from the other invariants in three respects: (1) equivalence testing involves multiple ARCs; (2) it requires precise ARCs, because the actual paths taken in the network are the attributes under scrutiny; and (3) it is implemented by comparing ETGs, rather than computing graph characteristics of ETGs. However, prior to comparing the ETGs from different ARCs, we must make two transformations to the ETGs.

Convert process-based ETGs to interface-based ETGs. The above ARC, as we mentioned in Section 3.3, encodes processes. This prevents us from determining if *any* two control planes are *equivalent*, because the two control planes may use a different set of routing instances, causing their ETGs to contain a different set of vertices and edges. To address this issue, we convert our process-based ETGs into *interface-based ETGs*, which depends only on the physical network topology, not the routing processes running atop it. As an example, Figure 3.4 shows the transformed ETG that corresponds to the upper-left ETG in Figure 3.1b.

In particular, we take the following steps:

1. Replace each process's *in* and *out* vertices with an *in* and *out* vertex for each *physical interface* over which the process sends/receives route advertisements.
2. Replace the inter-device edges that used to connect the *out* vertex of a process P on one device to the *in* vertex of a process P' on another device with an edge connecting the *out* vertex of the interface over which P sends advertisements to P' to the *in* vertex of the interface over which P' receives advertisements from P.
3. Replace the intra-device edge E that used to connect a routing process's *in* and *out* vertices by a set of edges that connect the *in* vertex of each interface associated with the process to the *out* vertex of every other interface associated with the process; the edge weight is the same as the edge weight that was assigned to E. Note that an edge is not created between the *in* and *out* vertices of the same interface, because a router

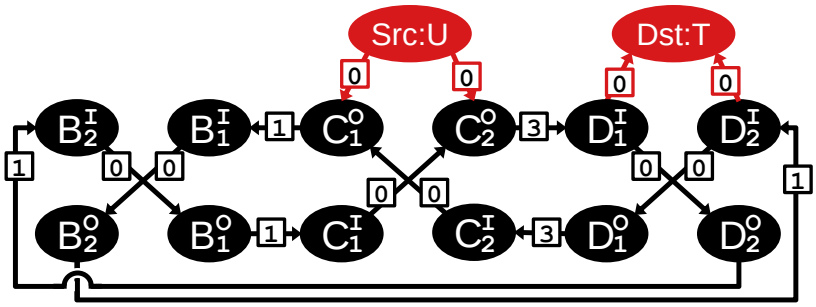


Figure 3.4: Part of the interface-based ARC for the example control plane in Figure 3.1a

will never send traffic out the same interface on which it arrived.

4. For each intra-device edge that connected the *in* vertex of a routing process P to the *out* vertex of another routing process P′, create an intra-device edge from the *in* vertex associated with each of P's interfaces to the *out* vertices associated with P′'s interfaces; again, the weight of these edge is the same as the weight of the original edge.

An ARC constructed in this manner represents a network's routing behavior with the same fidelity as the ARC described earlier, because it captures the exact same pathways between routers (no inter-device edges are added), and models at a fine granularity the same software pathways within routers.

Convert edge weights to canonical weights. There are infinitely many ARCs that differ only in the scale of their edge weights. All of these will produce the same data plane under all infrastructure faults, and hence are equivalent. To ensure we can detect such equivalence, we must *reduce all edge weights to canonical weights*. In other words, we compute the lowest possible weight for every edge in every ETG in the ARC such that the relative order of all possible loop-free paths between Src and Dst in each ETG is the same as using the original weights. We can perform such a reduction using a linear program; we omit details for brevity.

After applying the above transformations, we can test the equivalence of two control planes by checking whether their ARCs have the same vertices, edges, and edge weights. This is facilitated by the fact that vertices are always named based on the device interfaces to which they pertain. Thus, vertices and their incident edges can be easily matched across

ARCs.

3.6 Implementation & Evaluation

We implemented the ARC generation process described in Sections 3.3 and 3.4 and the verification tasks described in Section 3.5.1 in Java. We use Batfish [66] to parse Cisco IOS configurations. From these, we extract traffic classes and generate ETGs. We use JGraphT [15] to apply common graph algorithms (Dijkstra's shortest path, max-flow/min-cut, etc.) to the generated ETGs and obtain the information required to verify a particular property. Our tool outputs the results of the requested verification for all of a network's traffic classes.

We evaluate ARC along two different dimensions: (1) How efficiently can we represent real network control planes using ARC? (2) How quickly can we verify key invariants using ARC? How does this compare to state-of-the-art control plane verification tools (e.g., Batfish [66])?

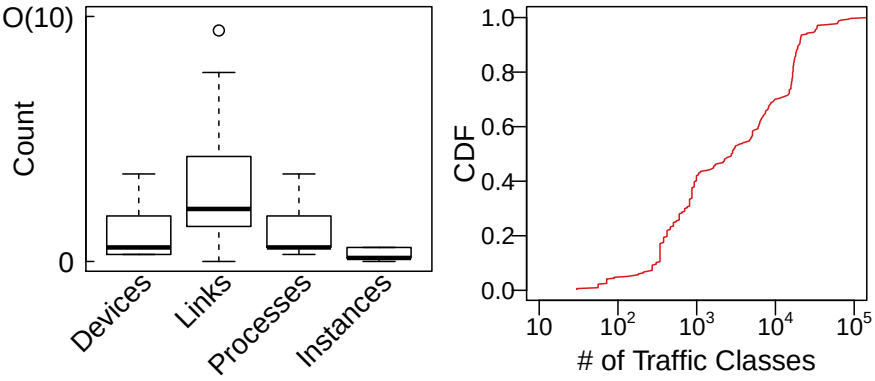
Dataset. In our evaluation we use configurations from about one-third (314) of the data center networks we studied in Chapter 2.[5] These networks have between two and a few tens of routers connected using between one and several tens of physical links (Figure 3.5a).

Two-thirds of the 314 networks have a single routing process on each device, while the remaining third have two processes per device on average (ignoring static routes). Similarly, two-thirds of the networks have a single routing instance, while the rest have a handful of instances (Figure 3.5a). As shown in Table 3.3 (and discussed in Chapter 2), only two routing protocols are used—OSPF (37% of networks) and eBGP (all networks)—along with static routes (27% of networks). Only one network has OSPF processes that use multiple areas, and only 10 networks have eBGP processes that use local preference (in addition to AS path length) for computing routes. Route redistribution occurs in 5% of the networks; in

[5]We exclude the remaining two-thirds of the OSP's networks from our evaluation, because these networks: (1) have no routers—only switches and middleboxes; (2) do not contain any Cisco devices—our current implementation is limited to IOS and NX-OS configurations; or (3) use configuration constructs that Batfish [66] does not parse.

Protocols	% of Networks	Modifiers	% of Networks
OSPF	37.6%	ACLs	100.0%
single area	37.3%	Route filters	84.1%
eBGP	100.0%	Route redistribution	5.4%
no local preferences	96.8%	acylic	5.4%
Static routes	27.1%	costs align with ADs	5.4%

Table 3.3: Control plane constructs in the OSP's networks



(a) Network size (b) Number of traffic classes

Figure 3.5: Scale of the OSP's networks

all cases, the redistribution conforms to the constraints necessary to produce a precise ARC (Section 3.4.4).

The number of distinct traffic classes ranges from less than 100 to more than 100K (Figure 3.5b). There are less than 10K traffic classes in 69% of the networks, and less than 1000 in 41% of networks. As shown in Table 3.3, the OSP uses route filters (84% of networks) and ACLs (all networks) to selectively block certain traffic classes.

By cross-referencing Tables 3.2 and 3.3, it is clear that we can generate a comprehensive ARC for *all* 314 networks, and a precise ARC for 97% of the networks (those with one OSPF area and no use of BGP local preference).

ARC Efficiency. We now examine how efficiently we can represent the OSP's network control planes using ARC. We consider both the time to generate the ARC and the ARC's size, and we show how this relates to the size and complexity of a network. We generate ARCs and verify invariants using a machine with a quad-core Intel Xeon 2.8GHz CPU and

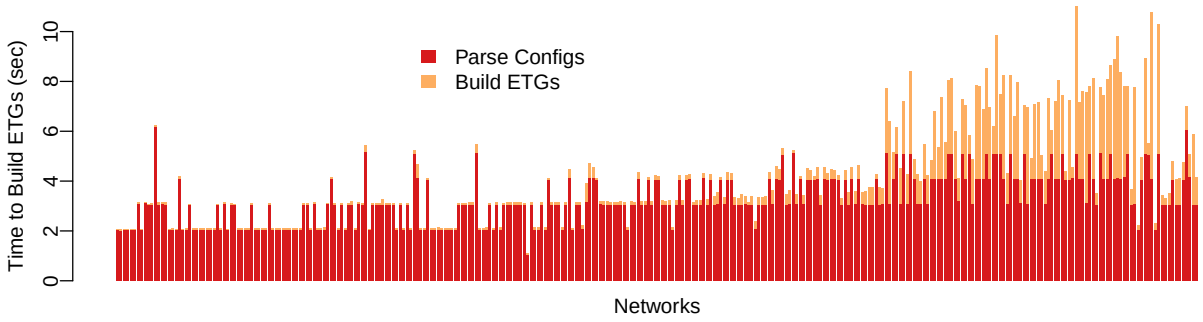


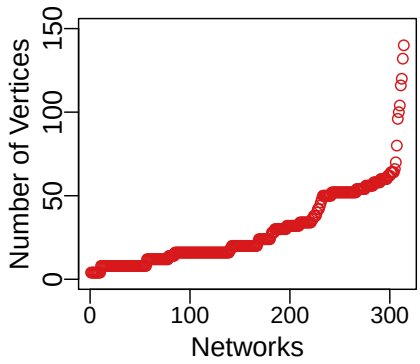
Figure 3.6: Time required to generate ARC for the OSP's networks: networks are sorted by number of traffic classes

24GB of RAM.

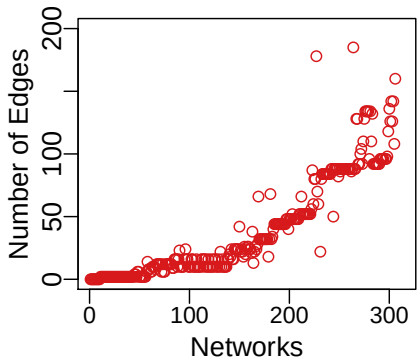
Figure 3.6 shows the time required to generate the ARC for each of the networks. ARC generation takes less than 5s for 78% of the networks, and at most 11.8s across all the networks we study. The majority of the time (85% on average) is spent parsing network configurations; this time is roughly correlated with the number of devices in the network (Pearson correlation co-efficient of 0.58). The remaining time is dedicated to constructing the ETGs; this time is roughly correlated with the number of traffic classes in the network (Pearson correlation co-efficient of 0.62), because the ARC contains an ETG for every traffic class.

Figure 3.7 characterizes the size of the generated ETG for each network. We observe that ETGs are relatively compact: 45% (45%) of the ETGs have fewer than 20 vertices (edges) and 74% (70%) have fewer than 50. By design, the number of vertices is directly correlated with the number of routing processes (including static routes) in the network for which the ETG is generated.

As mentioned above, the number of ETGs required for each network is a function of the number of traffic classes (Figure 3.5b). Although this seems substantial, 78% of networks' ETGs take less than 100MB of space when stored as serialized Java objects; more efficient encoding schemes could significantly reduce this. Furthermore, we show in the next section that exhaustively verifying key invariants for all of a network's traffic classes takes less than a second for most networks.

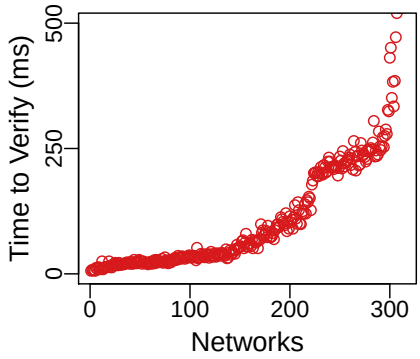


(a) Number of vertices

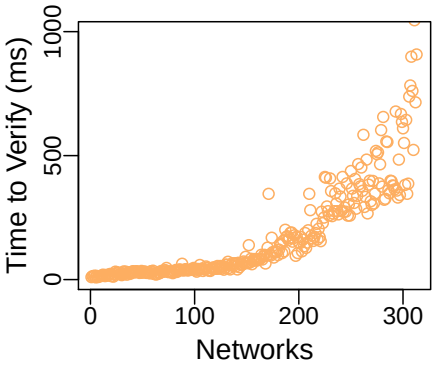


(b) Number of edges

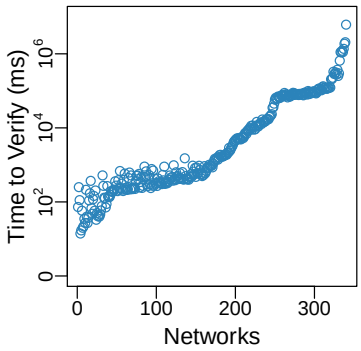
Figure 3.7: Size of the ETGs for the OSP's networks: networks are sorted by number of vertices in the ETG



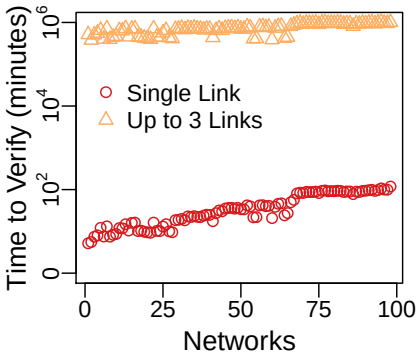
(a) I1: Always blocked, using ARC



(b) I2: Always reachable with $< k$ infrastructure faults, using ARC



(c) I3: Always isolated, using ARC



(d) Reachability, using Batfish

Figure 3.8: Time required to check key invariants: (a-c) show the time required to check the invariants for all traffic classes (or pairs of traffic classes) using ARC; networks are sorted by number of traffic classes. (d) shows the time required by Batfish to verify (lack of) reachability across a limited set of infrastructure faults; networks are sorted by number of links.

Verification Efficiency. We next examine how efficiently we can verify the invariants discussed in Section 3.5.1. Figure 3.8 shows the time required to verify invariants I1–I3 for all traffic classes (or pairs of traffic classes) for each of the networks.[6] We observe that invariant I1 can be checked for arbitrary link failures and all traffic classes in less than 500ms for 97% of the networks, and 62% of the networks can be checked in less than 100ms. The time per traffic class ranges from 8µs to 347µs (median 21µs).

The time required to verify invariant I2 is slightly higher, because computing max-flow/min-cut is more complex than checking if two nodes reside in separate connected components. However, for 99% of the networks, this property can be checked in less than 1s, and 54% of networks can be checked in less than 100ms. In the worst case, verification takes 1.13s. The time per traffic class ranges from 7µs to 467µs (median 32µs). Note that the time required for checking this property is independent of the value of k.

Invariant I3 takes substantially longer to verify, because we check all pairs of traffic classes, as opposed to each individual traffic class. It takes about 1.7 hours to check all pairs of traffic classes in the network with the largest number of traffic classes ($> 100\mathsf{K}$), but this property can be checked in less than 1 minute for all pairs of traffic classes in 73% of networks. In practice, only a subset of traffic classes in a network require isolation, so the number of traffic class pairs that need to be checked is substantially smaller.

Comparison with Batfish. To put our performance results in perspective, we compare the speed of ARC-based control plane verification against Batfish [66], a state-of-the-art network configuration analysis tool.[7] We ran Batfish on the device configurations from one-third (100) of the networks; we chose networks of varying size and complexity. We ran Batfish's "failure consistency" checker, which verifies that each traffic class is consistently blocked or allowed when any one of the network's links fails. This is similar to verifying invariants I1 and I2 ($\mathsf{k} = 2$) using ARC, except verification with ARC covers all link failures, not just single link

[6] We do not check invariant I4, because we do not know *where* middleboxes reside in the OSP's networks, only which and how many middleboxes exist.

[7] We do not compare against other verification tools [87, 88, 91, 105], because they only consider the current network data plane.

failures.

Figure 3.8d shows the time required for Batfish to check the reachability of all traffic classes under a limited set of link failures. We observe that the time taken by Batfish to check all single link failure scenarios is at least *three orders of magnitude larger* than the time required for ARC-based verification to check *all* link failure scenarios. If we were to run Batfish for all scenarios with up to 3 link failures, the time would further increase by up to *five orders of magnitude* making Batfish impractical to use in this case.

The time required by Batfish to verify invariants across a set of link failure scenarios is a function of: (1) the number of scenarios, and (2) the time required to generate the data plane and verify the invariant for each scenario. In our experiments, Batfish takes between 48s and 131s (median 92s) to generate the data plane and verify the invariant for each link failure scenario. With ARC, the time required to verify invariants across arbitrary link failure scenarios is a function of: (1) the number of traffic classes, and (2) the time required to generate the ETG and verify the invariant for each traffic class. As mentioned above, the median verification time per ETG for invariant I1 is 21µs and the median ETG build time is 98µs. Thus, a network with a single link would need to have over 773K traffic classes in order for ARC to be less efficient than Batfish.

In summary, ARC has a significant performance advantage over state-of-the-art verification tools.

3.7 Related Work

Data plane verifiers. One class of network verifiers [33, 87, 88, 91, 105] build a model of the network's current data plane based on snapshots of device forwarding tables or software-defined network (SDN) control messages. Because they verify the current data plane, these tools cannot proactively check if a reachability invariant, such as those listed in Table 3.1, would be satisfied if links or devices failed. Likewise, data plane verifiers cannot be used for equivalence testing.

Control plane verifiers. A different class of network verifiers operate on the device configurations, allowing them to detect errors in the control plane.

Some of these verifiers model specific devices (e.g., firewalls [31, 149]) or protocols (e.g., BGP [64] or IPsec [77]) in isolation. As such, they are not well suited for verifying today's data center networks, which make use of multiple device types and routing protocols (Section 2.2). In particular, modeling devices and protocols in isolation prevents these tools from checking properties for traffic classes whose source and destination endpoints are connected to different devices or routing domains.

Other control plane verifiers [47, 62, 65, 98] look for inconsistencies in device configurations, both across devices and relative to standard organizational practices. However, configurations that are internally consistent may still result in functional failures when infrastructure faults occur: e.g., a link failure within an OSPF routing instance may cause the new shortest path to pass between two VLANs, thus violating an invariant that communication between a source endpoint in one VLAN and a destination endpoint in the other VLAN is always blocked.

A recently developed tool, Batfish [66], models several routing protocols and their interactions using Datalog. This allows Batfish to *generate data plane models* for a set of infrastructure fault scenarios and verify an invariant holds across the generated data planes. Unfortunately, tools such as Batfish are slow because they do not abstract the network at all, but instead try to mimic low level protocol interactions and generate a full data plane. This can take as long as a few minutes (Section 3.6). Furthermore, Batfish must generate the data plane for every possible infrastructure fault scenario. To verify the reachability invariants listed in Table 3.1 for all single and two-link failure scenarios, Batfish must generate and examine $O(|E|^2)$ data planes, where E is the set of links in the network. In the worst case, Batfish must generate an exponential (in $|E|$) number of data planes, making it impractical; this occurs in the case of equivalence testing.

Dynamic checkers. Unlike static verifiers, dynamic checkers [150] can check for both

functionality (e.g., reachability) and performance problems. However, such checking cannot be done proactively, so applications or end hosts may be impacted by functional failures before they are detected.

Algebraic modeling. Lastly, a significant body of work has focused on modeling routing protocols and their interactions using algebras [32, 76, 100, 130, 131]. However, these algebras are designed for analyzing the low-level message exchanges of routing protocols and the intermediate states of route computation, whereas the goal of ARC is to verify the behavior of the network in steady state. Of course, ARC's abstraction of individual routing protocols' behavior does not capture all the nuances of the routing protocols used in data centers, limiting the extent to which ARC can handle certain policies—e.g., mutual route-redistribution [25, 99].

3.8 Summary

This chapter introduced an *abstract representation for control planes* (ARC) that models the forwarding behavior of data center networks at a higher level of abstraction than existing verification tools. This enables network operators to proactively and efficiently verify that a given control plane configuration will result in data planes that satisfy important functional requirements, even under arbitrary link and router failures. Furthermore, ARC allows a network operator to efficiently check the equivalence of two control planes, which can be useful when refactoring a network's design to avoid practices that increase the likelihood of network problems (Section 2.4). Our evaluation of ARC using a subset of the data center networks we studied in Chapter 2 shows that most verification tasks complete in less than a second, which is orders of magnitude faster than state-of-the-art tools.

While ARC is a valuable framework for reducing functional failures related to the routing control plane, it leaves middleboxes—another important element of the data center network and major contributor to data center failures (Section 2.4)—largely unaddressed. In particular,

ARC only ensures traffic is *routed to* a middlebox instance; it does not ensure the middlebox *correctly processes* the received traffic. We address this gap in the next chapter.

4 MAINTAINING MIDDLEBOX FUNCTIONALITY AND PERFORMANCE USING OPENNF

In Chapter 2, we showed that the diversity of device types present in a data center network strongly impacts the frequency of network problems. Such diversity arises from the presence of middleboxes, including firewalls, load balancers, application delivery controllers (ADCs), and intrusion detection/prevention systems (IDSs/IPSs). Middleboxes are essential for improving the security and performance of services running within the data center, so removing some of these devices in an effort to reduce network failures is impractical. Instead, we must look for other ways to reduce the frequency of problems arising from middleboxes.

ARC (Chapter 3) addresses one possible cause of middlebox-related problems: it ensures traffic is *always* routed to a middlebox instance (assuming at least one instance is active and reachable) even amidst arbitrary link failures. However, ARC does not ensure the middlebox correctly processes the received traffic, nor does it address other common middlebox issues that may lead to functional or performance failures. Recent studies [75, 113] of middlebox-induced failures in some of the data center networks we studied in Chapters 2 and 3 show that common problems include: connectivity errors (e.g., link flaps), hardware faults, software faults (e.g., unexpected reboots), overload, and misconfiguration (Table 4.1). The last of these aligns with our findings in Chapter 2, where we showed that configuration changes have a strong impact on the frequency of network problems.

We believe several of the aforementioned issues can be eliminated (or reduced) by replacing individual hardware appliances—the norm in today's data center networks [75, 113]—with a collection of software instances that expose a "one-big-middlebox" abstraction. Such an abstraction provides the illusion of a *monolithic, always available, high performing middlebox*. The middlebox is configured through a single, centralized interface, thereby reducing the likelihood of configuration incompatibilities (e.g., mismatched cryptographic keys) between individual middlebox instances [113]. Moreover, running software instances

	Setup		
Issue	**Hardware appliances**	**One-big-middlebox without OpenNF**	**One-big-middlebox with OpenNF**
Connectivity errors	Yes	Reduced/eliminated	
Hardware faults	Yes	Reduced/eliminated	
Software faults	Yes	Unchanged	
Misconfiguration	Yes	Reduced	
Overload	Yes	Exacerbated	Reduced/eliminated
Missing/inconsistent state	–	New Issue	Eliminated

Table 4.1: Possible causes of middlebox failures

atop generic compute resources eliminates the long repair (and outage) times associated with hardware faults [113], as new middlebox instances can be quickly launched on another server without needing to wait for a specialized device to be repaired or replaced. Finally, software middleboxes can reduce the failures arising from connectivity errors, as new middlebox instances can be launched on compute nodes anywhere in the data center network.

However, a one-big-middlebox abstraction backed by a collection of software instances running in virtual machines or containers is not a panacea. Software faults, some types of misconfiguration, and overload problems persist in this setting (Table 4.1). In fact, overload problems may be exacerbated, as software middleboxes are generally slower than their hardware counterparts [55]. Furthermore, unless great care is taken in managing the lifecycle of middlebox instances and the distribution of traffic among them, a new class of problems may arise due to missing or inconsistent middlebox state.

To illustrate how such problems can arise, consider a scenario in which traffic must be redistributed among a pair of middlebox instances (e.g., a pair of IDSs) to reduce the load on one of the instances and avoid a performance failure (Figure 4.1). Middleboxes' operations are typically stateful—i.e., the processing of one packet influences the processing of a later packet from the same connection or end host—so each middlebox instance will have state objects corresponding to the in-progress flows traversing that instance. When some flows are rerouted from the overloaded instance (IDS_1) to the underloaded instance (IDS_2), the underloaded instance may process the subsequent packets of these flows incorrectly,

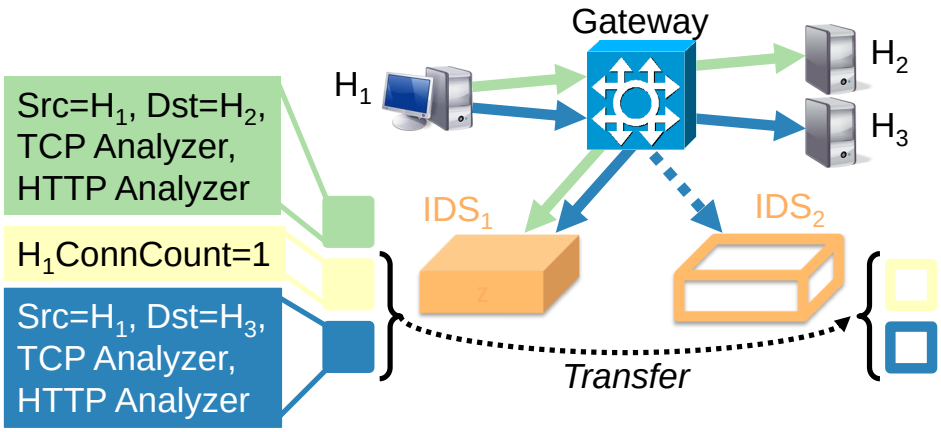


Figure 4.1: A scenario requiring scale-out and special handling of middlebox state to avoid performance and functional failures: The IDS (e.g., Bro [112]) processes a copy of all traffic entering/leaving the data center to detect port scans and malware in HTTP flows. For each active flow, the IDS maintains a connection object with source and destination IPs and ports, as well as several analyzer objects with protocol-specific state (e.g., current TCP sequence numbers or partially reassembled HTTP payloads). It also maintains host-specific connection counters. If a second IDS instance is launched and the blue (darker) flow is reassigned to the second instance to avoid performance issues, then the flow-specific state must be moved and the host-specific state must be copied or shared to ensure no attacks go undetected.

because the state established during the processing of the flows' prior packets is not available on the underloaded instance. Unlike forwarding tables, which can be reconstructed by a network's control plane, middlebox state depends on prior data traffic and hence cannot be reconstructed. In order to avoid a functional failure due to the missing state, we must either: (1) transfer or replicate the state associated with the rerouted flows from the overloaded to the underloaded instance, or (2) assign only new flows to the underloaded instance. The latter has the drawback of not reducing the load on the overloaded instance until some in-progress flows complete, thus this strategy may still result in a performance failure.

A similar problem arises if a middlebox instance loses connectivity, and we must reroute flows to a different middlebox instance to avoid a functional failure. However, in this scenario, the only way to avoid a failure is to reroute all flows away from the unconnected instance and transfer or replicate their state.[1] Likewise, when a middlebox instance crashes or restarts due

[1]We assume each middlebox instance has a dedicated management interface, as is the case with the hardware appliances in the networks we studied in Chapters 2 and 3, which is not subjected to the same connectivity errors as the interfaces and links over which data traffic arrives.

a software or hardware fault, we can only avoid a functional failure if we have proactively replicated the failed instance's state.

This chapter introduces a middlebox state management framework, called OpenNF, that prevents functional failures due to missing or inconsistent middlebox state. In particular, OpenNF *provides efficient, coordinated control of middlebox state and the traffic a middlebox receives* in order to allow quick, safe, and fine-grained reallocation of flows across middlebox instances. Using OpenNF, middlebox vendors can create rich control applications that expose a one-big-middlebox abstraction and dynamically redistribute packet processing responsibilities and state to avoid functional and performance failures from arising due to overload, lost connectivity, or software or hardware faults (Table 4.1).

We begin this chapter with an overview of the goals of a one-big-middlebox abstraction and the middlebox state management requirements they induce (Section 4.1). We then present an overview of OpenNF (Section 4.2), followed by our APIs for interacting with middleboxes (Section 4.3) and our algorithms for safely transferring and replicating middlebox state (Sections 4.4 and 4.5). Afterwards, we show how our APIs can be used to prevent functional and performance failures amidst infrastructure faults and workload changes (Section 4.6). Finally, we provide an overview of our implementation (Section 4.7), and we evaluate OpenNF's benefits and overhead using open source middleboxes and traffic traces from cloud and private data centers (Section 4.8).

4.1 Goals and Requirements

Goals. A one-big-middlebox abstraction should provide the illusion of an *infinitely scalable, always available, predictably performing middlebox*. The abstraction should mask functional or performance issues that arise on the underlying middlebox instances and guarantee output consistency—i.e., the aggregate output of the collection of underlying instances should be equivalent to the output produced by a single monolithic middlebox instance [121].

Additionally, the abstraction should be realized efficiently, using only the minimum resources required to maintain availability and performance.

Requirements. Achieving these goals requires managing the lifecycle of middlebox instances and the distribution of traffic among them as follows:

- *Predictable performance* requires redistributing traffic to eliminate hotspots and launching new middlebox instances when the current collective processing capacity is insufficient to satisfy performance service level objectives (SLOs).
- *High availability* requires rerouting traffic to different (possibly new) middlebox instances when an instance is down (due to a hardware or software fault) or unreachable (due to connectivity errors).
- *Efficiency* requires consolidating traffic and terminating unneeded middlebox instances as quickly as possible.
- *Output consistency* requires preserving the preconditions assumed by the underlying middlebox instances: e.g., middleboxes generally assume they can access (and update) any state they created while processing prior packets from the same flow.

Unfortunately, the requirements imposed by these goals are in conflict. For example, as discussed in the opening of this chapter, redistributing in-progress flows to eliminate a hotspot will cause the state for those flows to be unavailable at the middlebox instance to which the traffic is assigned, thus violating the requirement that we preserve the preconditions assumed by the middlebox. In the case of a load balancer, this may cause the rerouted connections to be reset, or the subsequent packets may be sent to a different server than the prior packets, which may cause the server to mishandle or reset the connections. Similarly, in the case of an IDS or IPS, malicious content in the rerouted connections may not be detected, because signature matching is split between the original and new instance, and the latter lacks information about prior packets. To achieve predictable performance, efficiency, *and* output consistency, we need a way to *transfer subsets of middlebox state* in concert with

routing updates. Depending on how traffic is split between middlebox instances and the granularity at which a middlebox maintains state (e.g., per-connection, per-host, per-subnet, etc.), we may also need to *share subsets of middlebox state* such that multiple middlebox instances can access and update the same state if each of them receives part of the traffic to which the state pertains.

Rerouting traffic to maintain high availability can cause similar issues. However, when a middlebox instance crashes due to a hardware or software fault, we have no way to obtain its state in order to achieve output consistency. Instead, we must proactively *replicate middlebox state* to ensure it can be made available at a new middlebox instance in the event the current instance crashes and traffic must be rerouted. Furthermore, to maintain efficiency, we want to avoid running a dedicated standby instance for every active middlebox instance. A more efficient approach is to divide the affected traffic among (a subset of) the remaining middlebox instances [120]; this requires the ability to replicate *subsets* of middlebox state to different instances.

Existing Solutions. Many application-agnostic solutions have been developed for transferring or replicating state associated with applications running in a hardware-independent environment. For example Xen [49] and CRIU [8] allow an entire virtual machine or process, respectively, to be migrated to another host. Similarly, Remus [52] facilitates high frequency replication of an entire virtual machine with minimal pauses in execution. However, these solutions are unsuitable for addressing the above requirements, because they do not allow migration or replication of a *subset* of a middlebox instance's state. For example, if we wanted to divide a middlebox instance's traffic among two instances, we would need to launch a clone of the original instance in order to effectively "transfer" the middlebox state associated with the rerouted traffic. The additional, unneeded state included in the clone not only wastes memory, but more crucially can cause undesirable middlebox behavior: e.g., an IDS may generate false alerts (we quantify this in Section 4.8.5). Moreover, this approach prevents state from multiple middlebox instances from being transferred to a single middlebox instance,

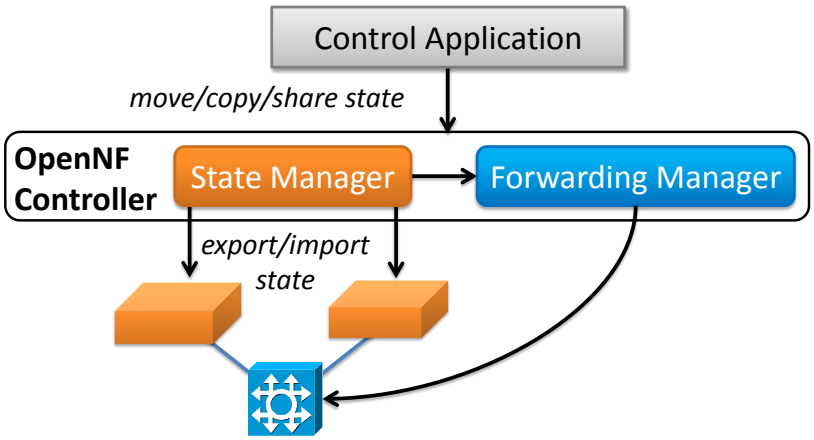


Figure 4.2: OpenNF architecture

precluding our ability to consolidate middlebox instances in order to maintain efficiency. Thus, we advocate *trading-off the need for middlebox code modifications in exchange for efficiency and performance.*

Many middlebox-oriented solutions with similar goals to ours have also been developed [86, 120, 121, 127]. However, as we describe later, these solutions: require significantly restructuring how a middlebox internally organizes and allocates state [86, 120, 121] (Section 4.3), do not provide important safety guarantees [121] (Section 4.4), and restrict how traffic can be redistributed [127] (Section 4.6).

4.2 OpenNF Architecture

OpenNF is a novel control plane architecture (Figure 4.2) that satisfies the aforementioned requirements. In this section, we outline our key ideas; the next three sections provide the details.

OpenNF allows control applications (Section 4.6) to closely manage the functionality and performance of a data center's middleboxes to avoid failures. Based on middlebox output or external input, control applications: (1) determine the precise sets of flows that specific middlebox instances should process, (2) direct the controller to provide the needed state at each instance, including both flow-specific state and state shared between flows, and (3) ask

the controller to provide certain guarantees on state and state operations.

In turn, the OpenNF controller (Sections 4.4 and 4.5) encapsulates the complexities of distributed state control and, when requested, guarantees loss-freedom, order-preservation, and consistency for state and state operations. We design two novel schemes to overcome underlying race conditions: (1) an *event abstraction* that the controller uses to closely observe updates to state, or to prevent updates but know what update was intended, and (2) *"tracer" packets* that help the controller determine when all of a flow's outstanding packets have arrived at a middlebox instance. Using just the former, the controller can ensure move operations are loss-free, and state copies are eventually consistent. By carefully sequencing state updates or update prevention (scheme 1) with scheme 2, the controller can ensure move operations are loss-free and order-preserving; we provide a formal proof in Appendix B. Lastly, by buffering events corresponding to intended updates and handling them one at a time in conjunction with piece-meal copying of state, the controller can ensure state copies are strongly or strictly consistent.

OpenNF defines a standard middlebox interface (Section 4.3) for a controller to request events or the export or import of internal middlebox state. We *leave it to the middleboxes to furnish all state matching a filter* (e.g., a 5-tuple) specified in an export call, and to determine how to merge existing state with state provided in an import call. This requires modest additions to middleboxes—we show in Section 4.8.3 that the necessary modifications increase middlebox code size by 3–8%, mostly due to serialization functions—and, crucially, does not restrict, or require modifications to, the internal state data structures that middleboxes maintain. Furthermore, we use the well-defined notion of a flow (e.g., TCP connection) as the basis for specifying which state to export and import. This naturally aligns with the way middleboxes already create, read, and update state.

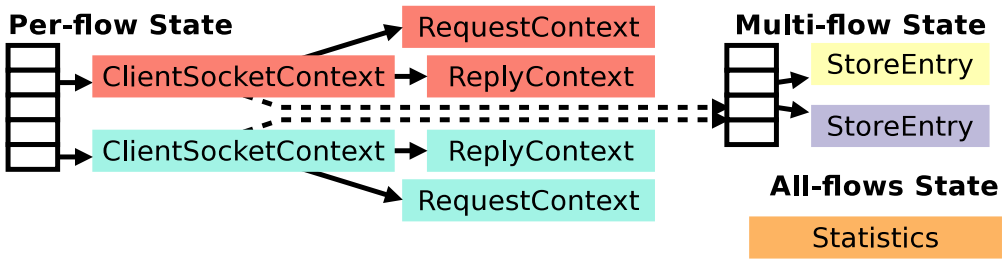


Figure 4.3: Middlebox state taxonomy, with state from the Squid caching proxy as an example

4.3 Middlebox API

In this section, we describe the design of OpenNF's middlebox API. To ensure a variety of middleboxes can be easily integrated into OpenNF, we must address two challenges: (1) account for the diversity of middlebox state and (2) minimize middlebox modifications.

State Taxonomy. To address the first challenge, we must identify commonalities in how middlebox state is allocated and accessed across various middleboxes. To this end, we examined several types of middleboxes from a variety of vendors, including: NATs [13], IDSs [112], load balancers [3, 10], caching proxies [26], WAN optimizers [35], and traffic monitors [19, 23].

We observe that *state created or updated by a middlebox while processing traffic applies to either an individual flow (e.g., a TCP connection) or a collection of flows.* As shown in Figure 4.1, the Bro IDS [112] maintains connection and analyzer objects for each TCP/UDP/ICMP flow and state for each host summarizing observations relating to all flows involving that host. Similarly, as shown in Figure 4.3, the Squid caching proxy [26] maintains socket context, request context, and reply context for each client connection and cache entries for each requested web object. Most middleboxes also have state which is updated for every packet or flow the middlebox processes: e.g., statistics about the number of packets/flows the middlebox processed.[2]

Thus, as shown in Figure 4.3, we classify middlebox state based on *scope*, or how many

[2]Middleboxes also have configuration state. It is read but never updated by middleboxes, making it easy to handle; we omit the details for brevity.

flows a middlebox-created piece of state applies to—one flow (*per-flow*), multiple flows (*multi-flow*), or all flows (*all-flow*). In particular, per-flow state refers to structures/objects that are read or updated only when processing packets from the same flow (e.g., a TCP connection), while multi-flow state is read or updated when processing packets from multiple, but not all, flows.

Thinking about each piece of middlebox-created state in terms of its association with flows provides a natural way for reasoning about how a control application should move/copy/share state. For example, a control application that routes all flows destined for a host H to a specific middlebox instance can assume the instance will need all per-flow state for flows destined for H and all multi-flow state which stores information related to one or more flows destined for H. This applies even in the case of seemingly non-flow-based state: e.g., the fingerprint table in a redundancy eliminator is classified as all-flows state, and cache entries in a Squid caching proxy are multi-flow state that can be referenced by client IP (to refer to cached objects actively being served), server IP, or URL.

Assuming traffic is never split among middlebox instances at a sub-flow granularity, a control application should always move (e.g., when rebalancing load) or copy (e.g., when creating a replica in case of an infrastructure fault) per-flow state, as packets from the same flow will never be processed by more than one middlebox instance at the same time. Whether to move, copy, or share multi-flow state depends on the granularity of a middlebox's analyses relative to the granularity at which traffic is split among middlebox instances. We generally expect control applications (Section 4.6) to be written by the middlebox vendor who is well equipped to specify the appropriate operations.

Other middlebox state management frameworks either draw no association between state and flows [83, 86], or they do not distinguish between multi-flow and all-flows state [121]. This makes it difficult to know the exact set of state to move, copy, or share when flows are re-routed. For example, in the Squid caching proxy, cached web objects (multi-flow states) that are currently being sent to clients must be copied to avoid disrupting these in-progress

connections, while other cached objects may or may not be copied depending on the SLOs a control application needs to satisfy (e.g., high cache hit ratio vs. fast scale out).[3]

We also discovered during our examination of middleboxes that they tend to: (1) allocate state at many points during flow processing—e.g., when the Bro IDS is monitoring for malware in HTTP sessions, it allocates state when the connection starts, as protocols are identified, and as HTTP reply data is received—and (2) organize/label state in many different ways—e.g., the Squid caching proxy organizes some state based on a traditional 5-tuple and some state based on a URL. Other frameworks [121] assume middleboxes allocate and organize state in particular ways (e.g., allocate state once for each flow), which means middleboxes may need significant changes to use these frameworks.

Middlebox API to Export/Import State. We leverage our taxonomy to design a simple API for middleboxes to export and import pieces of state; it requires minimal middlebox modifications. In particular, we leverage the well defined notion of a flow (e.g., TCP or UDP connection) and our definition of state scope to allow a controller to specify exactly which state to export or import. State gathering and merging is delegated to middleboxes which perform these tasks within the context of their existing internal architecture.

For each scope we provide three simple functions: get, put, and delete. More formally, the functions we expect each middlebox to implement are:

`multimap<`*flowid*`,`*chunk*`> getPerflow(`*filter*`)`

`void putPerflow(multimap<`*flowid*`,`*chunk*`>)`

`void delPerflow(list<`*flowid*`>)`

`multimap<`*flowid*`,`*chunk*`> getMultiflow(`*filter*`)`

`void putMultiflow(multimap<`*flowid*`,`*chunk*`>)`

`void delMultiflow(list<`*flowid*`>)`

`list<`*chunk*`> getAllflows()`

`void putAllflows(list<`*chunk*`>)`

[3]Middlebox-specific state sharing features, such as inter-cache protocols in Squid, can also be leveraged, but they do not avoid the need for per-flow state, and some multi-flow state, to be moved or copied.

A *filter* is a dictionary specifying values for one or more standard packet header fields (e.g., source/destination IP, network protocol, source/destination ports), similar to match criteria in OpenFlow [108].[4] This defines the set of flows whose state to get/put/delete. Header fields not specified are assumed to be wildcards. The `getAllflows` and `putAllflows` functions do not contain a *filter* because they refer to state that applies to all flows. Similarly, there is no `delAllflows` function because all-flows state is always relevant regardless of the traffic a middlebox is processing.

A *chunk* of state consists of one or more related internal middlebox structures, or objects, associated with the same flow (or set of flows): e.g., a chunk of per-flow state for the Bro IDS contains a `Conn` object and all per-flow objects it references (Figure 4.1). A corresponding *flowid* is provided for each chunk of per-flow and multi-flow state. The *flowid* is a dictionary of header fields and values that describe the exact flow (e.g., TCP or UDP connection) or set of flows (e.g., host or subnet) to which the state pertains. For example, a per-flow *chunk* from the Bro IDS has a *flowid* that includes the source and destination IPs, ports, and transport protocol, while a multi-flow *chunk* containing a counter for an end host has a *flowid* that only includes the host's IP.

When `getPerflow` or `getMultiflow` is called, the middlebox is responsible for identifying and providing all per-flow or multi-flow state that pertains to flows matching the *filter*. Crucially, *only fields relevant to the state are matched against the filter*; other fields in the *filter* are ignored: e.g., in the Bro IDS, only the IP fields in a *filter* will be considered when determining which end host connection counters to return. This API design avoids the need for a control application to be aware of the way a middlebox internally organizes state. Additionally, by identifying and exporting state on-demand, we avoid the need to change a middlebox's architecture to conform to a specific memory allocation strategy [121].

The middlebox is also responsible for replacing or combining existing state for a given flow (or set of flows) with state provided in an invocation of `putPerflow` (or `putMultiflow`). Common

[4]Some middleboxes may also support extended *filter*s and *flowid*s that include header fields for other common protocols: e.g., the Squid caching proxy may include the HTTP URL.

methods of combining state include adding or averaging values (for counters), selecting the greatest or least value (for timestamps), and calculating the union or intersection of sets (for lists of addresses or ports). State merging must be implemented by individual middleboxes because the diversity of internal state structures makes it prohibitive to provide a generic solution.

Middlebox API to Observe/Prevent State Updates. The API described above does not interpose on internal state creations and accesses. However, there are times when we need to prevent a middlebox instance from updating state—e.g., while state is being moved—or we want to know updates are happening—e.g., to determine when to copy state.

OpenNF uses two mechanisms to prevent and observe updates: (1) having middleboxes generate packet-received events for certain packets—the controller tells the middlebox which subset of packets should trigger events—and (2) controlling how middleboxes should act on the packets that generate events—process normally, buffer locally, or do not process them.

Specifically, we add the following functions to the API:

`void enableEvents(`*filter*`,`*action*`)`
`void disableEvents(`*filter*`)`

The *filter* defines the set of packets that should trigger events; it has the same format as described above. The *action* may be `process-normally`, `buffer-locally`, or `do-not-process`; any buffered packets are released to the middlebox for processing when events are disabled. The events themselves contain a copy of the triggering packet. The functions are implemented in a shared library that is linked with each middlebox during compilation (Section 4.7); hooks into the shared library must be added to a middlebox's packet receive function.

In the next two sections, we discuss how events are used to realize important guarantees on state and state operations.

4.4 Controller API: Move Operation

The OpenNF controller uses the middlebox API to implement operations for moving, copying, and sharing state between middlebox instances while guaranteeing loss-freedom, order-preservation, and various forms of consistency, if requested. The main challenge in moving, copying, and sharing state is designing suitable, low-overhead mechanisms to provide the necessary guarantees. In this section, we show how we use events together with tracer packets to provide a loss-free and order-preserving `move` operation (we provide a formal proof of these guarantees in Appendix B). In the next section, we describe how OpenNF's `copy` and `share` operations provide eventual, strong, or strict consistency for state required by multiple middlebox instances.

4.4.1 Syntax and semantics

OpenNF's `move` operation transfers the state *and* traffic for a set of flows from one middlebox instance (*srcInst*) to another (*dstInst*). Its syntax is:

`move`(*srcInst*, *dstInst*, *filter*, *scope*, *properties*)

As in the middlebox API, the set of flows is defined by *filter*; a single flow is the finest granularity at which a move can occur. The *scope* argument specifies which class(es) of state (per-flow and/or multi-flow) to move, and the *properties* argument defines whether the move should be loss-free and order-preserving.

This design appropriately balances OpenNF's generality and complexity. Not offering some guarantees would reduce complexity but make OpenNF insufficient for use with many middleboxes—e.g., a redundancy eliminator [35] will incorrectly reconstruct packets when re-ordering occurs. Similarly, always enforcing the strongest guarantees would simplify the API but make OpenNF insufficient for scenarios with tight SLOs—e.g., a loss-free and order-preserving move is unnecessary for a NAT, and the latency increase imposed by these guarantees (Section 4.8.1) could cripple VoIP sessions.

(a) Off-path middlebox (b) On-path middlebox

Figure 4.4: Assumed topologies for move operation

4.4.2 Implementation

We now describe how the `move` operation is implemented and various guarantees are realized. In what follows, *gw* denotes a gateway through which all packets matching *filter* will pass before diverging on their paths to reach *srcInst* and *dstInst* (Figure 4.4). The gateway is responsible for directing network traffic to the appropriate middlebox instance in the data center. The gateway may be implemented as: a dedicated (virtual) appliance whose architecture is similar to a load balancer, an SDN switch, or a router whose forwarding behavior is carefully controlled using static routes or fake route advertisements designed to provide SDN-like control [143]. Furthermore, we assume TCP-based control channels are used between the OpenNF controller and middleboxes, so middlebox API calls, state, and events are not lost or reordered. However, packets may be lost (but we assume not reordered) on the network paths from *gw* to *srcInst* and *gw* to *dstInst*.

No Guarantees. For a move without guarantees, the controller executes the following steps in sequence: (1) call `getPerflow` on *srcInst*, (2) call `delPerflow` on *srcInst*, (3) call `putPerflow` on *dstInst*, and (4) update the flow table on *gw* to forward the affected flows to *dstInst*. Each step is executed only after the controller has received confirmation that the middlebox instance or *gw* as completed the preceding step. To move multi-flow state as well (or instead), the analogous multi-flow functions are also (instead) called. For the rest of this section, we assume the *scope* is per-flow, but our ideas can easily be extended to multi-flow state.

With the above sequence of steps, packets corresponding to the state being moved may continue to arrive at *srcInst* from the start of `getPerflow` until after the forwarding change at

gw takes effect and all packets in transit to *srcInst* have arrived and been read from the NIC and operating system buffers. A simple approach of dropping these packets when *srcInst* receives them [121] prevents *srcInst* from establishing new state for the flows or failing due to missing state. But this is only acceptable in scenarios where an application is willing to tolerate the effects of skipped processing: e.g., scan detection in the Bro IDS will still function if some TCP packets are not processed, but it may take longer to detect scans. Alternatively, a middlebox may be on the forwarding path between flow endpoints (Figure 4.4b), e.g., a Squid caching proxy, in which case dropped TCP packets will be retransmitted, although throughput will be reduced.

Loss-free move. In some situations loss is problematic: e.g., the Bro IDS's malware detection script will compute incorrect md5sums and fail to detect malicious content if part of an HTTP reply is missing; we quantify this in Section 4.8.2. Similarly, a monitoring middlebox that measures traffic volumes for billing purposes will undercount traffic if packets are lost. Thus, we need a move operation that satisfies the following property:

> ***Loss-free:*** *All state updates resulting from packet processing should be reflected at the destination instance, and all packets the switch receives should be processed.*

The first half of this property is important for ensuring all information pertaining to a flow (or group of flows) is available at the instance where subsequent packet processing for the flow(s) will occur, and that information is not left, or discarded, at the original instance. The latter half ensures a middlebox does not miss gathering important information about a flow.

In an attempt to be loss-free, Split/Merge halts, and buffers at the controller, all traffic arriving at *gw* while migrating per-flow state [121]. However, when traffic is halted, packets may already be in-transit to *srcInst*, or sitting in NIC or operating system queues at *srcInst*. Split/Merge drops these packets when they (arrive and) are dequeued at *srcInst*. This ensures that *srcInst* does not attempt to update (or create new) per-flow state after the transfer of state has started, guaranteeing the first half of our loss-free property. However, dropping

packets at *srcInst* violates the latter half. While we could modify Split/Merge to delay state transfer until packets have drained from the network and local queues, it is impossible to know how long to wait, and extra waiting increases the delay imposed on packets buffered at the controller.

What then should we do to ensure loss-freedom in the face of packets that are in-transit (or buffered) when the move operation starts? In OpenNF, we leverage events raised by middleboxes. Specifically, the controller calls `enableEvents(`*filter*`,do-not-process)` on *srcInst*, and waits for the operation to complete, before calling `getPerflow`. This causes *srcInst* to raise an event for each received packet matching *filter*. The events are buffered at the controller until the `putPerflow` call on *dstInst* completes. Then, the packet in each buffered event is sent to *dstInst* via the control channel; any events arriving at the controller after the buffer has been emptied are handled immediately in the same way. Lastly, the flow table on *gw* is updated to forward the affected flows to *dstInst*.

Calling `disableEvents(`*filter*`)` on *srcInst* is unnecessary, because packets matching *filter* will eventually stop arriving at *srcInst* and no more events will be generated. Nonetheless, to eliminate the need for *srcInst* to check if it should raised events for incoming packets, the controller can issue this call after several minutes—i.e., after all packets matching *filter* have likely arrived or timed out.

Order-preserving move. In addition to loss, middleboxes can be negatively affected by re-ordering. For example, the "weird activity" policy script included with the Bro IDS will raise a false "SYN_inside_connection" alert if the IDS receives and processes SYN and data packets in a different order than they were actually exchanged by the connection endpoints. Another example is a redundancy elimination decoder [35] where an encoded packet arriving before the data packet w.r.t. which it was encoded will be silently dropped; this can cause the decoder's data store to rapidly become out of sync with the encoders.

Thus, we need a move operation that satisfies the following:

Order-preserving: *All packets should be processed by a middlebox instance in the*

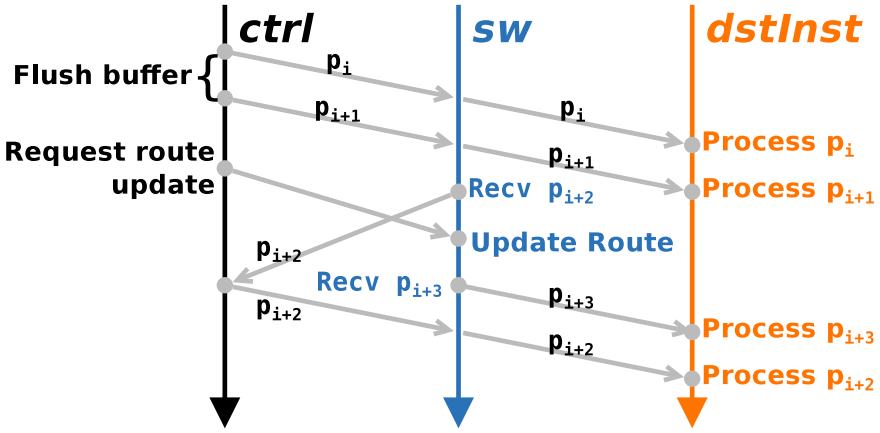


Figure 4.5: Order-preserving problem in Split/Merge

order they were forwarded to the middlebox instance by the switch.

This property applies within one direction of a flow (e.g., process SYN before ACK), across both directions of a flow[5] (e.g., process SYN before SYN+ACK), and across flows sent to the same middlebox instance (e.g., process an FTP get command before the SYN for the new transfer connection). In other words, we enforce a total order on the packets in a group of flows (defined by *filter*) assigned to the same middlebox instance, but we do not enforce a total order over all packets. If the latter is necessary for correct middlebox operation (e.g., if FTP control and data flows are sent to different IDS instances), then a control application can leverage our share operation with strict consistency (Section 4.5).

Unfortunately, neither Split/Merge nor the loss-free move described above are order-preserving. The basic problem in both systems is a race between flushing packets buffered at the controller and changing the forwarding table at *gw* to forward all packets to *dstInst*. Figure 4.5 illustrates the problem in the context of Split/Merge. Even if all buffered packets (p_i and p_{i+1}) are flushed before the controller requests a forwarding table update at *gw*, another packet (p_{i+2}) may arrive at *gw* and be forwarded to the controller before *gw* applies the forwarding table update. Once the update is applied, *gw* may start forwarding packets

[5]If packets in opposite directions do not traverse a common gateway before reaching the middlebox—e.g., a NAT is placed between two gateways—then we lack a vantage point to know the total order of packets across directions, and we cannot guarantee such an order unless it is enforced by a flow's end-points—e.g., a server will not send SYN+ACK until the NAT forwards the SYN from a client.

```
1  eventReceivedFromSrcInst(event)
2      if event.packet == TRACER_PKT then
3          srcProcessedLastPkt ← true
4      if shouldBufferEvents then                          // Buffer packets from srcInst
5          eventQueue.enqueue(event.packet)
6      else                                                // Send immediately to dstInst
7          mark(event.packet, do-not-buffer)
8          send(event.packet, dstInst)
9  eventReceivedFromDstInst(event)
10     if event.packet == TRACER_PKT then
11         signal(DST_PROCESSED_LAST_PKT)                                  // wait @ 26
12 moveLossfreeOrderpreserve(srcInst, dstInst, filter)
13     shouldBufferEvents ← true
14     srcInst.enableEvents(filter, do-not-process)     // Send srcInst's packets to controller
15     chunks ← srcInst.getPerflow(filter)                          // Start state transfer
16     srcInst.delPerflow(chunks.keys)
17     dstInst.putPerflow(chunks)                                  // Finish state transfer
18     foreach event in eventQueue do                 // Send controller's buffer to dstInst
19         mark(event.packet, do-not-buffer)
20         send(event.packet, dstInst)
21     shouldBufferEvents ← false
22     dstInst.enableEvents(filter, buffer-locally)       // Buffer unmarked packets at dstInst
23     gw.install(filter, dstInst)                                        // Reroute traffic
24     repeat                                                    // Transmit tracer packet
25         gw.forward(TRACER_PKT, srcInst)
26         signaled ← wait(DST_PROCESSED_LAST_PKT, TIMEOUT)
27     until signaled                              // Wait for dstInst to receive tracer packet
28     dstInst.disableEvents(filter)       // Release and process packets buffered at dstInst
```

Figure 4.6: Pseudo-code executed by the controller for a loss-free and order-preserving move

(p_{i+3}) to *dstInst*, but the controller may not have received the packet p_{i+2} from *gw*. Thus, the packet p_{i+2} will be forwarded to *dstInst* after a later packet of the flow (p_{i+3}) has already been forwarded to *dstInst*.

We use a combination of events and tracer packets to guarantee a loss-free *and* order-preserving move. Figure 4.6 has psuedo-code for the steps.

We start with the steps used for a loss-free move, through calling `putPerflow` on *dstInst* (lines 12–17). After `putPerflow` completes we extract the packet from each buffered event, mark it with a special "do-not-buffer" flag, and send it to *dstInst* (lines 18–20); any events arriving at the controller after the buffer has been emptied are handled immediately in the same way (lines 6–8). Then, we call `enableEvents(`*filter*`,buffer-locally)`on *dstInst* (line 22),

so that any packets forwarded directly to *dstInst* by *gw* will be buffered; note that the packets marked with "do-not-buffer" are not buffered.

After enabling buffering on *dstInst*, we update the forwarding entry for *filter* on *gw* to forward matching packets to *dstInst* (line 23). We then send a tracer packet—i.e., an empty data packet marked with a special flag—to *gw* to be forwarded to *srcInst* (line 25). Since we have already re-routed the relevant traffic to *dstInst*, the tracer packet should be the last packet to arrive at *srcInst*. Thus, when we get an event from *srcInst* containing the tracer packet (lines 1–3), we know that *srcInst* has processed the last regular packet. We pass the tracer packet along to *dstInst* just as we do with all regular packets (lines 6–8).

Lastly, we wait (line 26) for an event from *dstInst* indicating it has received the tracer packet. Because the tracer packet is the last packet the controller sends to *dstInst*, an event from *dstInst* indicating it has received this packet (lines 9–11) implies that *dstInst* has processed all packets that were originally delivered to *srcInst*. Thus, we can safely call `disableEvents(`*filter*`)` on *dstInst* to release any packets that had already been sent to *dstInst* by *gw* and were buffered at *dstInst* (line 28).[6] Since the tracer packet could be lost on the path from *gw* to *srcInst*, we wait with a timeout and retransmit the tracer packet if necessary (lines 24-27).

In Appendix B, we formally prove that this sequence of steps is loss-free and order-preserving, even when network paths are lossy. Providing these guarantees when there is reordering on the path from *gw* to *srcInst* remains an open problem.

4.4.3 Optimizations

Supporting the above guarantees introduces both efficiency and scalability challenges. In particular, packets must be buffered at the controller from the time `enableEvents` is called until after `putPerflow` completes. This imposes high latency overhead on packets arriving during the transfer and requires significant CPU and memory capacity at the controller. For

[6] Packets arriving at *dstInst* continue to be buffered until the buffer has been emptied.

example, consider a scenario where we move 800 flows between two instances of the Bro IDS [112], while traffic is flowing at a rate of 20K packets/second (0.15Gbps). Buffering packets at the controller increases average packet latencies by over 110x (≈75ms), requires a buffer capacity of 38K packets, and consumes 6% of the controller's CPU[7].

Furthermore, latency overhead and resource demands grow as the controller receives more packets (due to an increase in traffic rate and/or an increase in state export/import time). For example, when the packet rate in the above scenario rises from 4K to 20K packets/second, latency overhead rises from 5ms to 75ms—a 15x increase in latency with just a 5x increase in traffic rate. As the traffic rate increases beyond a few Gbps, or state export/import time increases beyond a few tens of milliseconds, the controller may not be able to provide the required buffer capacity, especially if multiple moves are occurring simultaneously; the resulting buffer overflow will violate the loss-free guarantee that prompted buffering in the first place.

We introduce four optimizations to address these issues and avoid functional and performance failures during a loss-free or order-preserving move.

Pipelining. Our first optimization aims to reduce latency overhead, as well as buffer demands, by reducing the total time taken to complete a move operation. In particular, we leverage the fact that `getPerflow` and `putPerflow` operations can be, at least partially, executed *in parallel*. Rather than returning all requested states as a single result, the *srcInst* can return each *chunk* of per-flow state immediately, and the controller can immediately call `putPerflow` with just that *chunk*.[8] The forwarding update at *gw* occurs after the `getPerflow` and all `putPerflow` calls have returned.

Peer-to-peer (P2P) transfer. Our second optimization similarly aims to reduce the total time taken to complete a move operation by *avoiding triangular routing*. In other words, we transfer state and packets directly from *srcInst* to *dstInst* without passing through the

[7]Our controller machine is equipped with a 4-core 2.8GHz Intel Xeon CPU and 6GB of memory.

[8]Alternatively, an application could issue multiple pipelined moves that each cover a smaller portion of the flow space. However, this requires more forwarding rules in *gw* and requires the application to know how flows are divided among the flow space.

controller. This accelerates the state transfer and reduces the CPU and memory burden on the controller.

This optimization relies on two new middlebox API functions (implemented in the shared library described in Section 4.7):

`void transfer(`*dstInst*`,`*filter*`,`*scope*`,`*properties*`)`
`void accept(`*srcInst*`)`

The former is invoked on *srcInst* to indicate it should: (1) connect to *dstInst*, (2) export and send any state of type *scope* pertaining to active flows in *filter*, and (3) send packets that arrive during the state transfer to be processed by *dstInst*. The latter is invoked on *dstInst* to indicate it should: (1) listen for an incoming socket connection from *srcInst*, (2) import any state received over the connection, and (3) buffer and process any packets received over the connection. Both functions can be implemented in a middlebox-agnostic manner using the existing export (`getPerflow`/`getMultiflow`/`getAllflows`), import (`putPerflow`/`putMultiflow`/`putAllflows`), and event raising functions presented in Section 4.3.

Late locking and early release. The additional latency imposed on redirected packets can be further reduced by following a *late locking and early release* (LLER) strategy. For late-locking, the controller calls `getPerflow` on *srcInst* with a special flag instructing *srcInst* to enable events for each flow just before the corresponding per-flow state is prepared for export (avoiding the need to call `enableEvents` for all flows beforehand).[9] Also, once `putPerflow` for a specific *chunk* returns, the controller (or *dstInst* if conducting a P2P transfer) can release any events pertaining to that *chunk*.

Packet reprocessing. Our final optimization fundamentally changes the way OpenNF handles packets that arrive during a move operation. In particular, we allow *srcInst* to continue processing all packets that arrive during a move operation. This processing: (1) updates state at *srcInst*, and (2) produces any output corresponding to the received packets. For example, a network address translator (NAT) rewrites packet headers and outputs the

[9]Internally, the middlebox calls `enableEvents` on a per-flow basis prior to exporting the flow's state.

modified packets, while an IDS logs alerts if malicious traffic is detected. If a packet matching *filter* triggers an update to state, a copy of the packet is sent to *dstInst*, where it is *reprocessed* to bring the transferred state "up to speed." The packet is reprocessed by *dstInst* solely to *obtain any state updates* the middlebox requires to process future packets. For example, a NAT needs the mapping created during the processing of the first packet of a flow in order to correctly process future packets from the flow, and an IDS needs metadata from prior packets to detect attacks that span multiple packets or flows. We suppress all packet and log output at *dstInst* during reprocessing, because *srcInst* has already produced such output. The *srcInst* marks packets with a "do-not-output" flag before sending them to the controller (or *dstInst* if conducting a P2P transfer) to inform *dstInst* that output should be suppressed.

Packet reprocessing still requires copies of packets to be buffered at the controller (or *dstInst*), but this buffering is no longer part of the critical path, because *srcInst* processes the original packet and produces the corresponding output. In particular, for every packet matching *filter* that arrives at *srcInst* during a loss-free move, *srcInst*: (1) sends the packet to the controller (or *dstInst*), and (2) processes the packet normally. This means *packet processing is only delayed by the time it takes a middlebox to copy the packet.*[10] Furthermore, given that *dstInst* reprocesses packets solely to obtain state updates, there is no need to reprocess a packet if no state is updated when the packet is processed at *srcInst.*[11] This eliminates the need to raise an event for/buffer such packets, thereby reducing resource demands at the controller (and *dstInst*).

In addition to reducing latency overhead and resource demands, packet reprocessing allows OpenNF to *guarantee a move is loss-free or order-preserving, even when the controller's (or dstInst's) packet buffer overflows.* Buffer overflow normally causes packets, and their corresponding state updates to be lost. However, with packet reprocessing, the middlebox state on *srcInst* is always up-to-date, because packets matching *filter* that arrive at *srcInst*

[10]We assume middleboxes (have been modified to) lock the appropriate state objects while a packet is being processed in order to prevent a partially modified object from being exported.

[11]Or the updated state is not critical for avoiding a functional failure: e.g., only statistics are updated.

during the move are always processed by *srcInst*. Thus, we can always re-export state from *srcInst*, and import the updated state on *dstInst*, without worrying about processing buffered, or dropped, packets at *dstInst*.

Note that maintaining output equivalence in the presence of packet reprocessing requires a middlebox's execution to be *output deterministic* [34]. In other words, if a middlebox starts from some state S_i and processes a sequence of packets $P_{i+1}...P_{i+k}$, it is essential that the resulting output is always $O_{i+1}...O_{i+k}$. Without this property, *dstInst* could end up producing different output ($O'_{i+1}..O'_{i+k}$) than *srcInst* after reprocessing the packets, and the behavior of *dstInst* going forward could differ from the behavior that would have resulted if only *dstInst* had processed the packets or the move operation had never occurred. Output determinism is a weaker property than *value determinism*—which is enforced by other frameworks designed to make middleboxes highly-available [127]—and correspondingly requires fewer middlebox modifications to enforce.

Additional Optimizations. It may be possible to further reduce state transfer and packet buffering time through the use of additional optimizations. For example, we could employ ideas used in live virtual machine migration [49] and send multiple rounds of state deltas before stopping packet processing to send a final delta; the middlebox instrumentation required to track state deltas could be automated using tools like StateAlyzr [89]. However, we choose not to employ such optimizations in OpenNF to balance the trade-off between performance and the extent of middlebox modifications required.

4.5 Controller API: Copy and Share Operations

While OpenNF's `move` operation enables state to be transferred between middlebox instances, OpenNF's `copy` and `share` operations address applications' need for the same state to be readable and/or updateable at multiple middlebox instances and, potentially, for updates made at one instance to be reflected elsewhere. For example, to avoid functional failures due to infrastructure faults, a backup middlebox instance needs to keep an updated copy of all

per-/multi-/all-flows state (Section 4.1). Similarly, a control application that distributes an end-host's flows among multiple IDS instances to avoid a performance failure needs updates to the host connection counter at one instance to be reflected at the other instances in order to effectively detect port scans (and avoid a functional failure).

OpenNF's `copy` operation can be used when state consistency is *not required* or *eventual* consistency is desired, while `share` can be used when *strong* or *strict* consistency is desired. Note that eventual consistency is akin to extending our loss-free property to multiple copies of state, while strict consistency is akin to extending both our loss-free and order-preserving properties to multiple middlebox instances.

Copy operation. OpenNF's copy operation clones state from one middlebox instance (*srcInst*) to another (*dstInst*). Its syntax is:

`copy(`*srcInst*`,`*dstInst*`,`*filter*`,`*scope*`)`

The *filter* argument specifies the set of flows whose state to copy, while the *scope* argument specifies which class(es) of state (per-flow, multi-flow, and/or all-flows) to copy.

The `copy` operation is implemented using the get and put calls from the middlebox API (Section 4.3). No change in forwarding rules at *gw* occurs as part of copy, because state is not deleted from *srcInst*; *srcInst* can continue processing traffic and updating its copy of state. It is up to control applications to separately initiate a change in forwarding at *gw* if the situation warrants (e.g., by directly interacting with the *gw* or calling `move` for some other class of state).

Eventual consistency can be achieved by occasionally re-copying the same set of state. As described in Section 4.3, a middlebox will automatically replace or combine the new and existing copies when `putPerflow`, `putMultiflow`, and `putAllflows` are called. Since there are many possible ways to decide when state should be re-copied—based on time, middlebox output, updates to middlebox state, or other external factors—we leave it to control applications to issue subsequent copy calls. As a convenience, we provide a function for control applications to become *aware* of state updates:

`void notify(`*filter*`,`*inst*`,`*enable*`,`*callback*`)`

When invoked with *enable* set to true, the controller calls `enableEvents(`*filter*`,process-normally)` on middlebox instance *inst*, otherwise it calls `disableEvents(`*filter*`)` on *inst*. Events are asynchronous, so notifying the controller has a minimal impact on *inst*'s performance. For each event the controller receives, it invokes the provided *callback* function in the control application.

Share operation. Strong and strict consistency are more difficult to achieve, because state reads and updates must occur at each middlebox instance in the same global order. For strict consistency this global order must match the order in which packets are received by *gw*. For strong consistency the global order may differ from the order in which packets were received by *gw*, but updates for packets received by a specific middlebox instance must occur in the global order in the order the instance received the packets.

Both cases require synchronizing reads/updates across all middlebox instances (`list<`*inst*`>`) that are using a given piece of state. OpenNF's `share` operation provides this:

`void share(list<`*inst*`>,`*filter*`,`*scope*`,`*consistency*`)`

The *filter* and *scope* arguments are the same as above, while *consistency* is set to `strong` or `strict`.

Events can again be used to keep state strongly consistent. The controller calls `enable-Events(`*filter*`,do-not-process)` on each instance, followed by a sequence of get and put calls to initially synchronize their state. When events arrive at the controller, they are placed in a FIFO queue labeled with the *flowid* for the flow group to which they pertain; flows are grouped based on the coarsest granularity of state being shared (e.g., per-host or per-prefix).

For each queue, one event at a time is dequeued, and the packet it contains is marked with a "force-processing" flag and sent to the originating middlebox instance. The middlebox instance processes the packet and raises an event, which signals to the controller that all state reads/updates at the middlebox are complete. The controller then calls `getMultiflow`

(or `getPerflow`, `getAllflows`) on the originating middlebox instance, followed by `putMultiflow` (or `putPerflow`, `putAllflows`) on all other instances in `list<`*inst*`>`. Then, the next event is dequeued and the process repeated.

Since events from different middleboxes may arrive at the controller in a different order than packets were received by *gw*, we require a slightly different approach for strict consistency. The controller must receive packets directly from *gw* to know the global order in which packets should be processed. We therefore update all relevant forwarding entries in *gw*—i.e., entries that both cover a portion of the flow space covered by *filter* and forward to an instance in `list<`*inst*`>`—to forward to the controller instead. We then employ the same methodology as above, except we invoke `enableEvents` with *action* set to `process-normally` and queue packets received from *gw* rather than receiving packets via events.

We adopt the above approach of copying state after the processing of each packet to minimize the changes required to middleboxes. In particular, we can reuse the same middlebox API functions (Section 4.3) used for `move` and `copy` operations. However, this approach comes at a significant performance cost (Section 4.8.1). Thus, control applications should use `share` judiciously. Applications should also consider which multi-/all-flows state is required for accurate packet processing, and, generally, invoke copy or share operations on this state prior to moving per-flow state. More efficient state sharing mechanisms, such as a replicated state machine based on ABCAST [44] or CBCAST [45], can be implemented with additional middlebox modifications.

4.6 Control Applications

Using OpenNF, we have written control applications that achieve the performance and high availability goals described in Section 4.1. We expect that middlebox vendors, or someone familiar with a middlebox's high-level operation, will normally write such applications. We use the Bro IDS [112] in both the applications we present, but different middleboxes and goals place different requirements on both the granularities of state operations and the guarantees

```
1  standbys ← {}
2  initStandby(primaryInst, stbyInst, filter)
3      standbys[primaryInst].put(filter, stbyInst)
4      filter.nw_proto ← TCP
5      filter.tcp_flags ← SYN | RST
6      notify(filter, primaryInst, true, updateStandby)
7      filter.tcp_flags ← *
8      filter.tcp_dst ← 80
9      filter.nw_src ← 10.0.0.0/8
10     notify(filter, primaryInst, true, updateStandby)
11 updateStandby (event)
12     primaryInst ← event.src
13     filter ← extractFlowId(event.pkt)
14     stbyInst ← standbys[primaryInst][filter]
15     copy(primaryInst, stbyInst, filter, PER)
```

Figure 4.7: Application for maintaining high availability

needed. Despite these differences, control applications are relatively simple to implement. We describe them below.

High availability. The first application (Figure 4.7) maintains a collection of hot standbys for each Bro IDS instance with an eventually consistent copy of all per-flow and multi-flow state. Note that each standby is responsible for a subset of the primary instance's traffic, and each standby is also the primary instance for a different set of traffic, similar to the setup in Pico Replication [120]. Such a setup is more efficient than having a dedicated hot standby, as done by FTMB [127].

The `initStandby` function is invoked to initialize a standby (`stbyInst`) for a subset of the traffic (*filter*) assigned to an IDS instance (`primaryInst`). It notes which `primaryInst` and traffic subset the standby is associated with and requests notifications from `primaryInst` for packets in *filter* whose corresponding state updates are important for scan detection and browser identification—TCP SYN, SYN+ACK, and RST packets and HTTP packets sent from a local client to an external server. The copy is made eventually consistent when these key packets are processed, rather than recopying state for every packet. In particular, events are raised by `primaryInst` for these packets and the controller invokes the `updateStandby` function. This

```
1 movePrefix(prefix, oldInst, newInst)
2     copy(oldInst, newInst, {nw_src: prefix}, MULTI)
3     move(oldInst, newInst, {nw_src: prefix}, PER, LOSSFREE)
4     while true do
5         sleep(60)
6         copy(oldInst, newInst, {nw_src: prefix}, MULTI)
7         copy(newInst, oldInst, {nw_src: prefix}, MULTI)
```

Figure 4.8: Application for maintaining predictable performance

function copies the appropriate per-flow state from `primaryInst` to the corresponding `stbyInst`. When an infrastructure fault occurs, the forwarding rules in the gateway are updated to forward the appropriate prefixes to `stbyInst` instead of `primaryInst` (code not shown).

Predictable performance. The second application (Figure 4.8) monitors the CPU load on the Bro IDS instances and calculates a new distribution of local network prefixes when load becomes imbalanced. If a subnet is assigned to a different IDS instance, the `movePrefix` function is invoked. This function calls `copy` to clone the multi-flow state associated with scan detection, followed by `move` to perform a loss-free transfer of the per-flow state for all active flows in the subnet.

We copy, rather than move, multi-flow state because the counters for port scan detection are maintained on the basis of ⟨external IP, destination port⟩ pairs, and connections may exist between a single external host and hosts in multiple local subnets. An order-preserving move is unnecessary because re-ordering would only potentially result in the scan detector failing to count some connection attempts, and, in this application, we are willing to tolerate moderate delay in scan detection. However, to avoid missing scans completely, we maintain eventual consistency of multi-flow state by invoking copy in both directions every 60 seconds.

4.7 Implementation

Our OpenNF prototype consists of: (1) a controller that implements our `move`, `copy`, and `share` operations (Sections 4.4 and 4.5); (2) a shared library that middleboxes use for communicating

with the controller, conducting P2P transfers, and handling packets from events; and (3) several modified middleboxes—Bro, PRADS, and iptables—that implement our middlebox API (Section 4.3).

Controller. The OpenNF controller is written as a module atop Floodlight [9] (≈4.7K lines of Java code). The controller listens for connections from middleboxes and launches two threads—for handling state operations and events—for each middlebox. The controller and middleboxes exchange JSON messages to invoke middlebox API functions, provide function results, and send events. We use an OpenFlow [108] switch as the gateway (*gw*). The interface with control applications is event-driven.

Shared library. The shared library (≈2.6K lines of C code) is used by middleboxes for communicating with the controller, conducting P2P transfers, and handling packets from events. The shared library implements: wrappers for the state export/import API calls (Section 4.3), which call middlebox-specific export/import functions (described below); the `enable/disableEvents` API calls, along with the filtering and buffer management mechanisms they require; and `transfer` and `accept` API calls for P2P transfers. Like the controller, the shared library launches two threads for handling operations and events.

To inject packets (from events forwarded by the controller or a peer middlebox) into the middlebox's packet input stream, we use virtual Ethernet (veth) interfaces and bridging (Figure 4.9). When a middlebox starts, the shared library creates a bridge and two veth pairs—*vethP2Pin*/*vethP2Pbr* and *vethNFbr*/*vethNFin*. The interfaces in a veth pair are virtually wired together, such that a packet sent on one veth interface is received by the other veth interface. We add *vethP2Pbr*, *vethNFbr*, and *ethIn* to the bridge. We then configure the middlebox to read packets from *vethNFin*. When a packet arrives on *ethIn*, it passes through the bridge to *vethNFbr*, causing it to be received by *vethNFin* and read by the middlebox (solid red path in Figure 4.9). When a packet arrives on a controller or peer connection—or a packet that previously arrived on the connection is released from a buffer—we send the packet on *vethP2Pin*; the packet is received by *vethP2Pbr*, sent across the bridge to *vethNFbr*,

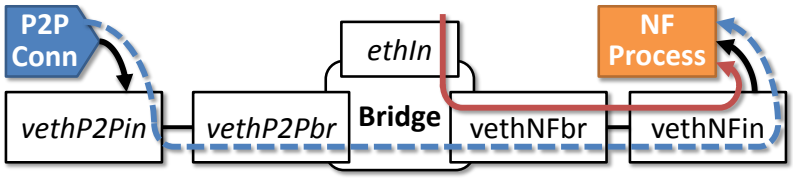


Figure 4.9: Setup for injecting packets from events into *dstInst*'s input stream: solid red path is taken by normal packets, dashed blue path is taken by packets that require reprocessing

received by *vethNFin*, and read by the middlebox (dashed blue path).

Middleboxes. We implemented middlebox-specific handlers for each middlebox API function. We discuss the middlebox-specific modifications below, and evaluate the extent of these modifications in Section 4.8.3.

Bro IDS [112] performs a variety of security analyses defined by policy scripts. The `get/putPerflow` handlers for Bro lookup (using linear search) and insert `Connection` objects into internal hash tables for TCP, UDP, and ICMP connections. The key challenge is serializing these `Connection` objects and the many other objects (>100 classes) they refer to; we wrote custom serialization functions for each of these objects using Boost [5]. We also added a *moved* flag to some of these classes—to prevent Bro from logging errors during `delPerflow`—and a mutex to the `Connection` class—to prevent Bro from modifying the objects associated with a flow while they are being serialized. Lastly, we added library calls to Bro's main packet processing loop to raise events when a received packet matches a filter on which events are enabled.

PRADS asset monitor [23] identifies and logs basic information about active hosts and the services they are running. The `get/putPerflow` and `get/putMultiflow` handlers for PRADS lookup and insert `connection` and `asset` structures, which store flow meta data and end host operating system and service details, respectively, in the appropriate hash tables. If an `asset` object provided in a `putMultiflow` call is associated with the same end host as an `asset` object already in the hash table, then the handler merges the contents of the two objects. The `get/putAllflows` handlers copy and merge, respectively, a global statistics structure.

Iptables [13] is a firewall and network address translator integrated into the Linux kernel.

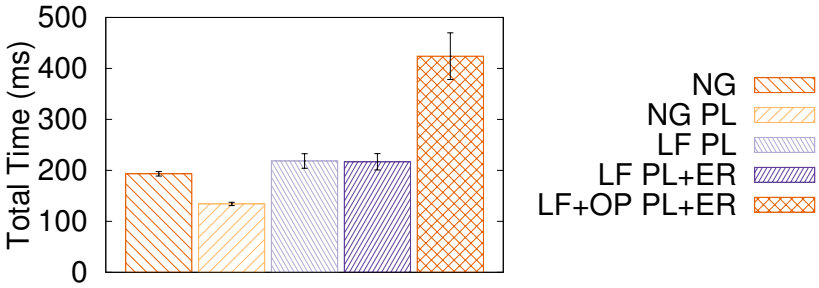
The kernel tracks the 5-tuple, TCP state, security marks, etc. for all active flows; this state is read/written by iptables. We wrote an agent that uses libnetfilter_conntrack [16] to capture and insert this state when `get/putPerflow` are invoked. There is no multi-flow or all-flows state in iptables.

4.8 Evaluation

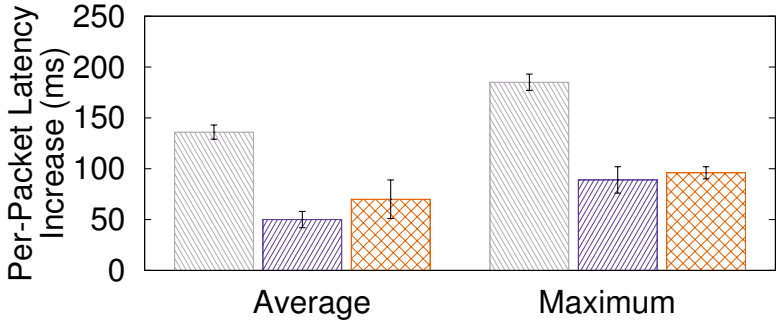
Our evaluation of OpenNF answers the following key questions:

- Can state be moved, copied, and shared efficiently even when guarantees on state or state operations are requested by applications? To what extent do our optimizations improve efficiency?
- What benefits do control applications see from the ability to move, copy, or share state with varying guarantees?
- How efficiently can middleboxes export and import state, and do these operations impact middlebox performance? How many modifications must be made to middleboxes to support the middlebox API?
- How is OpenNF's efficiency impacted by the scale of a middlebox deployment? To what extent do our optimizations reduce resource demands at the controller?
- To what extent do existing middlebox control planes hinder the ability to satisfy a combination of high-level objectives?

The testbed we use for our evaluation consists of an OpenFlow-enabled HP ProCurve 6600 switch and four mid-range servers (Quad-core Intel Xeon 2.8GHz, 8GB, 2 x 1Gbps NICs) that run the OpenNF controller and modified middleboxes and generate traffic. We use replayed university-to-cloud-data-center network traffic traces [78], along with synthetic workloads.



(a) Total `move` time



(b) Per-packet latency increase

Figure 4.10: Efficiency of `move`: with no guarantees (NG), loss-free (LF), and loss-free and order-preserving (LF+OP) with and without pipelining (PL) and early-release (ER) optimizations; traffic rate is 2500 packets/sec; times are averaged over 5 runs and the error bars show 95% confidence intervals

4.8.1 Efficiency of move, copy, and share operations

We first evaluate the efficiency of our move, copy, and share operations when guarantees are requested on state or state operations. We use two PRADS asset monitor instances (PRADS_1 and PRADS_2) and replay our university-to-cloud-data-center trace at 2500 packets/second. We initially send all traffic to PRADS_1. Once it has created state for 500 flows (≈80K packets have been processed), we `move` *all* flows and their per-flow state, or `copy` *all* multi-flow state, to PRADS_2. To evaluate sharing with strong consistency, we instead call `share` (for all multi-flow state) at the beginning of the experiment, and then replay our traffic trace. During these operations, we measure the number of dropped packets, the added latency for packets contained in events from PRADS_1 or buffered at PRADS_2, and the total operation time (for move and copy only). Although the specific values for these metrics vary based on the middlebox, scope, filter granularity (i.e., number of flows/states affected), and packet

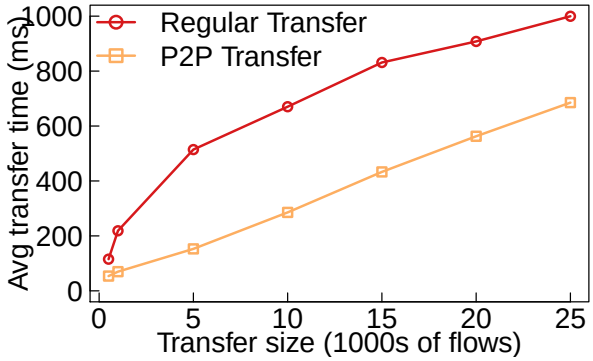


Figure 4.11: Improvements in `move` time with P2P transfers

rate, the high-level takeaways still apply.

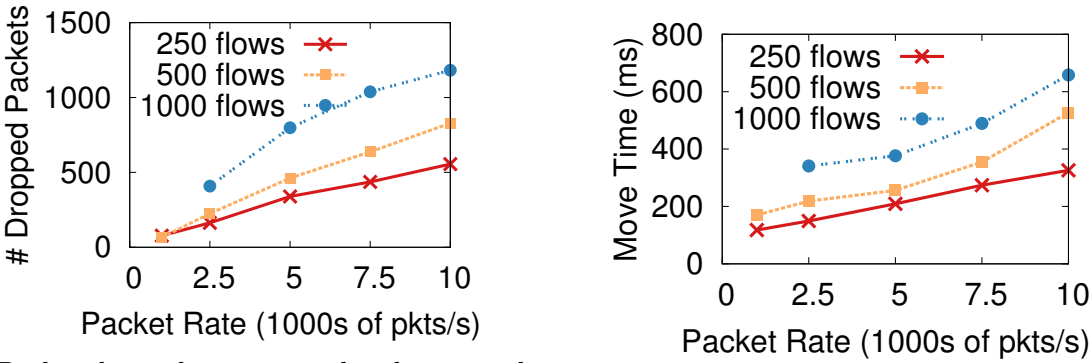
Move operation. Figure 4.10 shows our results for `move` with varying guarantees and optimizations.

A move without any guarantees or optimizations (NG) completes in 193ms. This time is primarily dictated by the time required for the middlebox to export (89ms) and import (54ms) state; we evaluate middlebox operations in detail in Section 4.8.3. The remaining 50ms is spent processing control messages from the middleboxes and updating forwarding at *gw*.

Our pipelining optimization (Section 4.4.3) can reduce the total time for the move operation (NG PL) to 134ms by exporting and importing state (mostly) in parallel. The time can be reduced even further using a P2P transfer. Figure 4.11 shows the time to complete a `move` operation for varying numbers of flows with and without P2P transfer; each data point is averaged across three iterations. We observe that P2P transfer reduces the average `move` time by at least 31%, and up to 70%, depending on the amount of state transferred.

However, even these faster versions of move come at a cost: *up to 225 packets are dropped!* Figure 4.12a shows how the number of drops changes as a function of the packet rate and the number of flows whose state is moved. We observe a linear increase in the number of drops as the packet rate increases, because more packets will arrive in the time window between the start of `move` and the routing update taking effect.

A loss-free move (LF PL in Figure 4.10) avoids drops by raising events. However, the



(a) Packet drops during a pipelined `move` with no guarantees

(b) Total time for a pipelined loss-free `move`

Figure 4.12: Impact of packet rate and number of per-flows states on pipelined `move` with and without a loss-free guarantee

410 packets contained in events may each incur up to 185ms of additional latency. (Packets processed by PRADS_1 before the move or PRADS_2 after the move do not incur additional latency.) Additionally, the total time for the move operation increases by 62% (84ms). Figure 4.12b shows how the total move time scales with the number of flows affected and the packet rate. We observe that the total time for a pipelined loss-free move increases more substantially at higher packet rates. This is because more events are raised, and the rate at which the packets contained in these events can be sent to PRADS_2 becomes limited by the event rate our controller can sustain. The average and maximum per-packet latency increase for packets contained in events also grows with packet rate for the same reason: e.g., the average (maximum) per-packet latency increase is 465ms (573ms) for a pipelined loss-free move of 500 flows at a packet rate of 10K packets/sec (graph not shown).

Our early-release optimization (Section 4.4.3) can decrease the additional packet latency. At a rate of 2500 packets/sec, the average per-packet latency overhead for the 326 packets contained in events drops to 50ms (LF PL+ER in Figure 4.10b), a 63% decrease compared to LF PL; at 10K packets/sec this overhead drops to 201ms, a 99% decrease.

In addition to added packet latency, a loss-free move also introduces re-ordering: 657 packets (335 from events + 322 received by PRADS_2 while packets from events are still arriving) are processed out-of-order with a pipelined loss-free move. However, this re-ordering

can be eliminated with an order-preserving move.

A loss-free and order-preserving move with pipelining and early-release optimizations (LF+OP PL+ER in Figure 4.10) takes 96% (208ms) longer than a fully optimized loss-free-only move (LF PL+ER) due to the additional steps involved. Furthermore, packets buffered at PRADS_2 (100 packets on average), while waiting for all packets originally sent to PRADS_1 to arrive and be processed, each incur up to 96ms of additional latency (7% more than LF PL+ER). Thus, control applications can benefit from choosing an alternative version of move if they do not require both guarantees.

Copy and share operations. A pipelined copy takes 111ms, with no packet drops or added packet latency, as there is no race between a routing update and copying state. In contrast, a share operation that keeps multi-flow state strongly consistent adds at least 13ms of latency to every packet, with more latency incurred when a packet must wait for the processing of an earlier packet to complete. This latency stems from the need to call `getMultiflow` and `putMultiflow` on PRADS_1 and PRADS_2, respectively, after every packet is processed, because our events only provide hints as to whether state changed but do not inform us if the state update is significant. For example, every packet processed by the PRADS asset monitor causes an update to the last seen timestamp in the multi-flow state object for the source host, but only a handful of special packets (e.g., TCP handshake and HTTP request packets) result in interesting updates to the object. However, adding more PRADS asset monitor instances (we experimented with up to 6 instances) does not increase the latency because `putMultiflow` calls can be issued in parallel. In general, it is difficult to efficiently support strong consistency of state without more intrinsic support from a middlebox, e.g., information on the significance of a state update.

Move operation with packet reprocessing. We now evaluate the benefits of packet reprocessing using the Bro IDS, instead of PRADS asset monitor, because an IDS's packet processing is more advanced and places more stress on the system. We conduct a loss-free move affecting 800 active flows while replaying traffic at varying rates—up to 52K packets

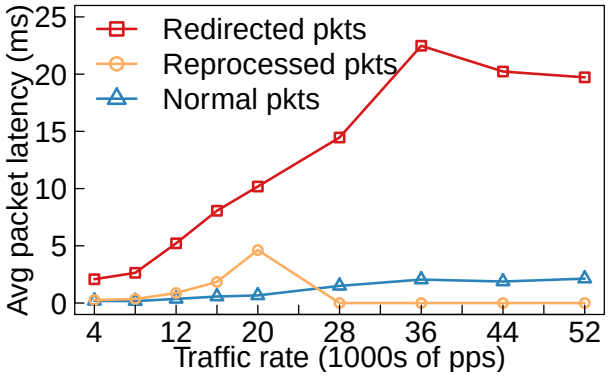


Figure 4.13: Improvements in latency overhead

per second (0.4Gbps). We measure per-packet latency as the time elapsed between a packet arriving at *gw* and a packet begin output by an IDS instance.[12] We compare the latency of: (1) packets processed normally by *srcInst* prior to the `move` operation, (2) packets in *filter* that are both processed and output by *srcInst* and reprocessed by *dstInst*, and (3) packets in *filter* that arrive during the move but are only processed by *dstInst*. The last case captures the latency overhead without the optimization.

Figure 4.13 shows the average packet latency in each of the three cases—labeled *normal*, *reprocessed*, and *redirected*, respectively. First, we observe that the latency for normal packets plateaus, and the latency for redirect packets dips, when the traffic rate is above 36K pps (0.3Gbps). This is the point where we exceed the packet processing capacity of a single IDS instance. More importantly, below this rate, we observe that packet reprocessing offers an 81% to 94% reduction in packet latency overhead compared to OpenNF's original design. However, packets in *filter* arriving during a move operation still incur an ≈2x inflation in latency compared to packets processed prior to the move. This overhead stems from the middlebox using some of its resources to export state.

4.8.2 Importance of guarantees

We next evaluate the importance of the guarantees offered by OpenNF. Our methodology is similar to our experiments above, except we use the Bro IDS with a malware detection

[12] We modified the Bro IDS to output packets after they are processed.

	Number of alerts			
Alert	**Baseline**	**NG**	**LF**	**LF+OP**
Incorrect File Type	26	25	24	26
Malware Hash Registry Match	31	28	27	31
Total	57	53	51	57

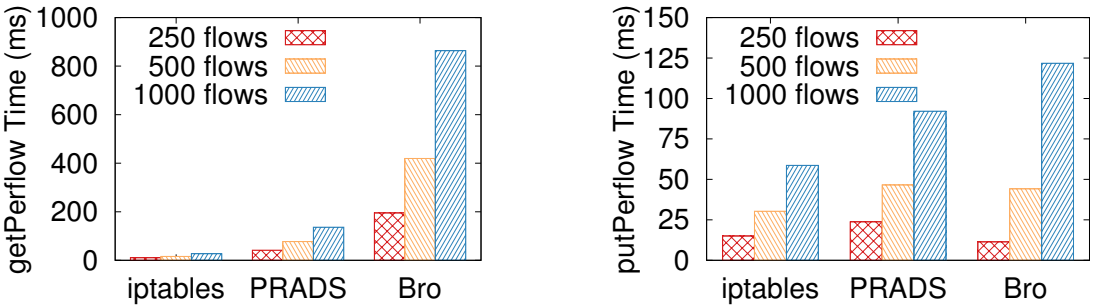
Table 4.2: Effects of different guarantees

script [6], and we replay a trace of malware traffic [17] at 1000 packets/second for 40 seconds. This represents the small fraction of malicious traffic we expect might be present in a data center's overall traffic. We issue a `move` operation (with the pipelining optimization) after 15 seconds.

We compare the alerts raised by the Bro IDS when no move is performed (baseline) versus when a no guarantee (NG), loss-free (LF), or loss-free plus order-preserving (LF+OP) move is performed. Table 4.2 shows the type and number of alerts raised under each scenario. We observe that 7% and 10% of the alerts occurring during our 40 second experiment are missed with a no guarantee or loss-free move, respectively. More alerts are missed with loss-free because re-ordering is introduced. In contrast, no alerts are missing with a loss-free plus order-preserving move. Depending on the frequency of load redistribution, the overall fraction of alerts missed may be higher or lower. Regardless, our experiment illustrates that the guarantees offered by OpenNF are essential to accurately monitor network traffic when packet processing is dynamically redistributed; this is especially true for middleboxes where accuracy is paramount—e.g., middleboxes that measure traffic volumes for billing purposes.

4.8.3 Middlebox API

The time required to export and import state at middleboxes directly impacts how quickly a `move` or `copy` operation completes and how much additional packet latency is incurred when `share` is used. We thus evaluate the efficiency of OpenNF's middlebox operations (Section 4.3) for the middleboxes we modified. We also examine how much code was added to the middleboxes to support these operations.



(a) Time for `getPerflow` (b) Time for `putPerflow`

Figure 4.14: Efficiency of state export and import

API call processing. Figures 4.14a and 4.14b show the time required to complete a `getPerflow` and `putPerflow` operation, respectively, as a function of the number of flows whose state is exported/imported. We observe a linear increase in the execution time of `getPerflow` and `putPerflow` as the number of per-flow state chunks increases. The time required to (de)serialize each *chunk* of state and send it to (receive it from) the controller accounts for the majority of the execution time. Additionally, we observe that `putPerflow` completes at least 2x faster than `getPerflow`; this is due to deserialization being faster than serialization. Overall, the processing time is highest for Bro because of the size and complexity of the per-flow state. The results for multi-flow state are qualitatively similar.

We also evaluate how middlebox performance is impacted by the execution of middlebox operations. In particular, we measure average per-packet processing latency (including queuing time) during normal middlebox operation and when a middlebox is executing a `getPerflow` call. Among the middleboxes, the PRADS asset monitor has the largest relative increase—5.8% (0.120ms vs. 0.127ms), while the Bro IDS has the largest absolute increase—0.12ms (6.93ms vs. 7.06ms). In both cases, the impact is minimal, implying that middlebox operations do not significantly degrade middlebox performance.

Middlebox changes. To quantify the middlebox modifications required to support our middlebox API, we counted the lines of code (LOC) that we added to each middlebox (Table 4.3). The counts do not include the shared library used with each middlebox for

Middlebox	LOC added for serialization	Total LOC added	Increase in code
Bro IDS	2.9K	3.3K	4.0%
PRADS asset monitor	0.1K	1.0K	9.8%
iptables	0.6K	1.0K	n/a

Table 4.3: Additional code to implement OpenNF's middlebox API

communication with the controller: ≈2.6K LOC. At most, there is a 9.8% increase in LOC[13], most of which is state serialization code that could be automatically generated [7]. Thus, the middlebox changes required to support OpenNF are minimal.

4.8.4 Controller scalability

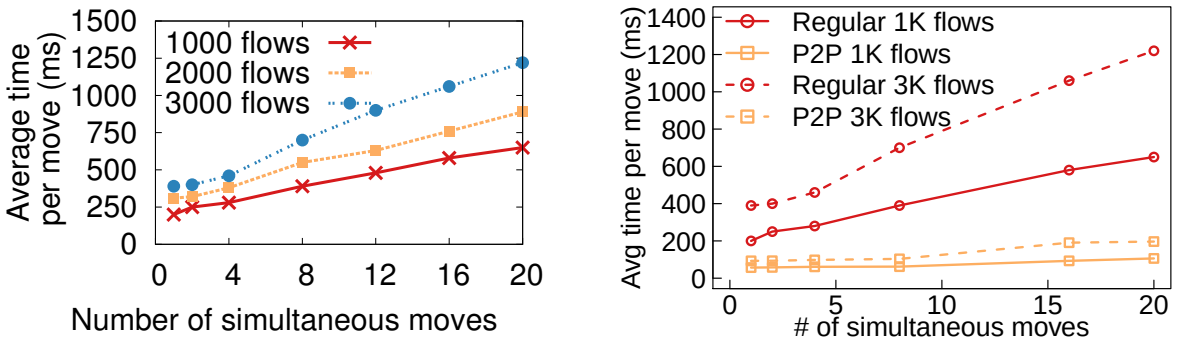
Since the controller oversees all `move`, `copy`, and `share` operations, its ability to scale is crucial. We thus measure the performance impact of conducting simultaneous operations across many pairs of middleboxes.

To isolate the controller from the performance of individual middleboxes, we use "dummy" middleboxes that replay traces of past state in response to `getPerflow`, simply consume state for `putPerflow`, and infinitely generate events during the lifetime of the experiment. The traces we use are derived from actual state and events sent by PRADS asset monitor while processing our cloud data center traffic trace. PRADS asset monitor state objects are small (202 bytes on average), so a move operation for this middlebox imposes the lowest load on the controller and thus represents the best case for controller scalability.

Figure 4.15a shows the average time per loss-free `move` operation as a function of the number of simultaneous operations. The average time per operation increases linearly with both the number of simultaneous operations and the number of flows affected.

We profiled our controller using HPROF [12] and found that during a `move` operation threads are busy reading from sockets most of the time. This implies that the volume of data the controller must handle is the primary bottleneck. One way to overcome this bottleneck is

[13]We do not calculate an increase for iptables because we wrote a user-level tool to export/import state rather than modifying the Linux kernel.



(a) Without P2P transfers

(b) With P2P transfers

Figure 4.15: Performance of concurrent loss-free `move` operations

to optimize the size of state transfers using compression. We ran a simple experiment and observed that, for a `move` operation for 500 flows, state can be compressed by 38%, reducing execution latency from 110ms to 70ms.

Our P2P transfer optimization (Section 4.4.3) can also significantly reduce controller overhead and improve scalability. Figure 4.15b shows the average time per operation when between 1 and 20 `move` operations occur simultaneously. We observe the transfer time per move is near constant with P2P transfers, regardless of the number of flows affected by each move operation. In contrast, without P2P transfers, the state transfer time steadily increases as the number of simultaneously operations increases.

4.8.5 Comparison with alternative approaches

Lastly, we compare the ability to maintain predictable middlebox performance using OpenNF versus alternative approaches [8, 39, 70, 84, 116]. We start with one Bro IDS instance ($\texttt{Bro}_1$) and replay our cloud data center traffic trace at a rate of 2500 packets/sec for 2 minutes. We then double the traffic rate, add a second Bro IDS instance ($\texttt{Bro}_2$), and `move` all HTTP flows to $\texttt{Bro}_2$ (other flows remain at $\texttt{Bro}_1$); 2 minutes later we scale back down to one instance. We use Bro because its state is the most complex among the middleboxes we modified

VM replication. This approach takes a snapshot of the current state in an existing middlebox instance ($\texttt{Bro}_1$) and copies it to a new instance ($\texttt{Bro}_2$) as is. Since, VM replication

does not do fine-grained state migration, we expect it to have unneeded states in all instances. We quantify unneeded state by comparing: a snapshot of a VM running the Bro IDS that has not yet received any traffic (*base*), a snapshot taken at the instant of scale up (*full*), and snapshots of VMs that have only received either HTTP or other traffic prior to scale up (*HTTP* and *other*). *Base* and *full* differed by 22MB. *HTTP* and *other* differed from *base* by 19MB and 4MB, respectively; these numbers indicate the the volume of state the IDS had at the time of snapshot for (active and inactive) HTTP and other flows, respectively. Consequently, these numbers represent the overhead imposed by the unneeded state at the two Bro IDS instances. In contrast, the amount of state moved by OpenNF (i.e., per-flow and multi-flow state for all active HTTP flows) was 8.1MB. More crucial are the correctness implications of unneeded state: we found 3173 and 716 incorrect entries in conn.log at the two Bro IDS instances, because the migrated HTTP (other) flows terminate abruptly at $\texttt{Bro}_1$ ($\texttt{Bro}_2$).

Scaling without re-balancing active flows. Frameworks that steer only new flows to new scaled out middlebox instances leave existing flows to be handled by the same middlebox instance [70]. Thus, $\texttt{Bro}_1$ continues to remain bottlenecked until some of the flows traversing it complete. Likewise, in the case of scale in, middleboxes are unnecessarily "held up" as long as flows are active. We observe that ≈9% of the HTTP flows in our cloud trace were longer than 25 minutes; this requires us to wait for more than 25 minutes before we can safely terminate $\texttt{Bro}_2$, otherwise we may miss detecting some attacks.

4.9 Related Work

Prevalence of middlebox failures. Several measurement studies have highlighted the potential for [61] and prevalence of [60, 75, 113] middlebox availability and performance problems. As discussed earlier, common causes of these problems include: connectivity errors, hardware and software faults, overload, and misconfiguration [113].

Trend towards software middleboxes. In the past few years, network operators and researchers have begun to advocate for the deployment of software middleboxes in place of hardware appliances [20, 54, 124]. The biggest proponents of software middleboxes have been Internet and telecom service providers, through a movement known as network functions virtualization (NFV) [20]. However, there has also been a push for using software middleboxes in the data center [54], as well as replacing hardware middleboxes in enterprise networks with software middleboxes hosted to a cloud data center [128].

Virtualization and orchestration. Several virtualization frameworks have been designed specifically for software middleboxes [36, 107, 124] to help address performance and resource efficiency issues that arise with this deployment model. In particular, they speed-up middlebox instance launch times [107] and reduce redundant operations (e.g., packet parsing) by consolidating several types of middlebox instances on the same generic compute node [36, 124]. Other researchers have argued that middleboxes should be placed in every hypervisor within a data center [54]; this model improves efficiency by avoiding the need for dedicated compute nodes to run middleboxes and improves performance by reducing the need to traverse extra hops in the data center when routing traffic between end hosts.

A complementary set of frameworks are designed for orchestrating the deployment of middlebox instances [22, 70, 111, 116]. These frameworks automatically monitor for overload and failures and, when necessary, launch additional middlebox instances at optimal locations within a data center. The frameworks also handle the interconnection of middlebox instances and the distribution of traffic between them. The challenge of interconnecting different types of middlebox instances in a sequence, commonly known as *service chaining* [119], has also been addressed independently of managing the lifecycle of middlebox instances [63, 84, 118].

The aforementioned virtualization and orchestration efforts help make the "one-big-middlebox" abstraction possible, but they leave gaps that are at least partially filled by OpenNF. In particular, some orchestration frameworks [70, 111] (re)distribute traffic in a way that maintains affinity—i.e., a flow is routed to the same middlebox instance throughout

its lifetime [144]—thus avoiding the need for explicit management of middlebox state. This approach has also been commonly used when designing scalable intrusion detection/prevention systems [123, 142, 148]. Other middlebox orchestration frameworks [116] redistribute traffic between middlebox instances without considering the implications for middlebox state. The latter will lead to functional failures, while the former may not address overload fast enough to avoid performance failures. Thus, there is a need for OpenNF's explicit state management.

Virtual machine and process replication. Generic solutions for transferring or replicating the state of virtualized applications—e.g., virtual machine migration [39, 49], virtual machine replication [52, 56], and process migration [8, 57, 103, 109]—only allow middlebox instances to be moved or cloned in their entirety. While cloning could be used as a way to transfer state to a new middlebox instance, the additional unneeded state included in a clone can cause undesirable middlebox behavior: e.g., an IDS may generate false alerts (Section 4.8.5). Such an approach is also unsuitable when transferring state prior to scale down, because it does not allow state from multiple middlebox instances to be consolidated. Furthermore, the inability to replicate only part of a virtualized middlebox's state limits the high availability strategies that can be employed using existing virtual machine replication solutions [52, 56]. In particular, we must maintain a dedicated standby instance for each middlebox instance, rather than making several existing middlebox instances—that are also processing other traffic—each responsible for a subset of a middlebox instance's traffic in the event of an infrastructure fault [120].

Understanding middlebox state. Finer-grained management of middlebox state requires an understanding of: (1) what types of state middleboxes maintain, (2) how the state is organized, and (3) how the state is used for middlebox operations. We explored the first two questions in Section 4.3; designers of frameworks similar to OpenNF [86, 120, 121] (described in detail below) have done the same. Efforts to build abstract models of middleboxes [83] or parallelize middleboxes [48] have explored the first and last of these questions through manual or programmatic inspection of specific middleboxes. Finally, StateAlyzr [89] explores

all three questions through programmatic analysis of arbitrary middleboxes. These works have informed, and can help us further refine, the design of OpenNF's middlebox APIs (Section 4.3); StateAlyzr in particular can help automate the task of modifying middleboxes to support our APIs.

Distributed state management. Frameworks designed for managing the state of generic distributed applications provide one possible avenue for manging middlebox state. One approach is to place state in a key/value store (e.g., Dynamo [53], FaRM [59], RAMCloud [110], etc.) rather than having state reside on application servers. Such an approach has been employed by StatelessNF [86]. However, modifying an existing middlebox to accommodate this approach requires changing all state accesses/updates in the middlebox code to communicate with the remove data store. Furthermore, for middleboxes which access and update state frequently—e.g., an IDS which updates state on every packet [112]—the latency penalty of communicating with the key/value store may be prohibitively expensive, even if key/value store accesses occur over low-latency channels (e.g., remote-direct memory access [59, 110]).

Another approach is to broadcast updates to all instances of a middlebox such that their view of state remains in sync (e.g., using CBCAST [45] or ABCAST [44], Apache Kafka [2], etc.). However, this approach also requires substantial modifications to middlebox code. Additionally, receiving all updates (in a consistent order) may not be sufficient to guarantee middlebox correctness: e.g., a state update made by one IDS instance may need to be reflected at another IDS instance before the latter processes a packet from another flow, otherwise the ordering of events perceived by the IDS may not correctly match an attack signature [48]. Nonetheless, these approaches may be useful to implementing more efficient state sharing (Section 4.5).

Middlebox state management. Given the complexity and performance issues with applying existing distributed state management approaches to middleboxes, several recent systems have been designed explicitly for handling middlebox state. Split/Merge [121] and Pico Replication [120] provide a shared library that middleboxes use to create, access, and

modify state through a pre-defined API. The API uses nondescript keys for multi-flow state, making it difficult to know the exact states to move and copy when flows are rerouted. Furthermore, the API only allows one state allocation per flow, requiring some internal middlebox state and packet processing logic to be significantly restructured. In contrast, OpenNF does not change how middlebox state is organized and allocated (Section 4.3).

In Split/Merge [121], an orchestrator is responsible for coordinating middlebox scaling and load balancing by invoking a simple *migrate* operation that reroutes a group of flows and moves corresponding middlebox state. Unfortunately, the migrate operation can cause state updates to be lost or reordered, because packets arriving at a middlebox instance after migrate is initiated are dropped, and a race exists between updating network forwarding and resuming the flow of traffic (which is halted when migrate starts). OpenNF carefully addresses such race conditions and guarantees state transfers are loss-free and order-preserving (Section 4.4).

In Pico Replication [120] and FTMB [127], all recently updated middlebox state is cloned to a backup instance at fixed intervals to protect against infrastructure faults. To ensure the backup remains consistent with the current state of network flows, these frameworks either buffer output until the state has been successfully replicated [120] or keep a log of input packets such that the replica can be brought "up to speed". The former introduces significant latency overhead, making it impractical for low-latency data center networks. The latter is similar to OpenNF's packet reprocessing optimization (Section 4.4.3), with extra emphasis on ensuring packet processing is deterministic; thus, it complements OpenNF.

4.10 Summary

This chapter introduced a *middlebox state management framework* (OpenNF) that facilitates the implementation of a "one-big-middlebox" abstraction backed by a collection of software middlebox instances. Such a deployment model reduces or eliminates middlebox-related connectivity, configuration, and infrastructure problems, thereby helping data center network operators avoid middlebox-induced functional and performance failures. OpenNF ensures

middleboxes have the state they need to correctly process the packets they receive. Moreover, OpenNF ensures that updates to middlebox state are not lost or reordered and copies of state are kept eventually, strongly, or strictly consistent, so as to avoid causing a functional failure when middlebox state is transferred or replicated.

We showed how to achieve the aforementioned guarantees using two novel techniques: (1) an *event abstraction* that allows the OpenNF controller to closely observe updates to middlebox state, or to prevent updates but know what update was intended, and (2) *tracer packets* that allow the controller to determine when all of a flow's outstanding packets have arrived at a middlebox instance. We also showed how to reduce the performance and resource overhead of these guarantees using several optimizations, including fine-grained buffering, peer-to-peer transfers, and packet reprocessing. Our thorough evaluation of OpenNF using traces of traffic exchanged with a public cloud data center showed that these optimizations offer significant benefits in terms of efficiency and scalability: e.g., our optimizations can reduce latency overhead by up to 99%, 70%, and 94%, respectively.

Although we have presented OpenNF in the context of a "one-big-middlebox" abstraction, it can also aid network operators in reducing the impact of network maintenance. For example, using OpenNF, network operators can quickly drain the traffic and state from a middlebox instance so the instance can be reconfigured or upgraded without needing to terminate active connections or wait for the connections to end normally. Additionally, OpenNF has applicability beyond a traditional data center. In particular, OpenNF can help address many of the same performance and high availability challenges faced by network service providers transitioning to network functions virtualization (NFV) [20].

5 CONCLUSION AND FUTURE WORK

In this thesis, we have conducted measurements and proposed solutions to understand and mitigate some of the most common causes of functional and performance failures in data center networks. In this closing chapter, we summarize our key contributions and present directions for future research that can help further reduce the frequency and severity of data center network failures.

5.1 Contributions and Impact

Identifying Problematic Network Management Practices. In Chapter 2, we introduced a *management plane analytics* (MPA) framework [74] that characterizes the relationships between network management practices and the frequency of network problems. Data center network operators can apply MPA to their networks to: (1) determine the top k practices that cause an increase in the frequency of network problems, and (2) predict, in an ongoing fashion, what impact a specific set of management practices will have on the health of individual networks. To achieve this, MPA uses carefully selected analysis and learning techniques, including mutual information, propensity score matching, boosting, and oversampling. These techniques help overcome the challenges introduced by the sometimes non-linear, overlapping, and skewed relationships between management practices and the rate of network problems. We have released the source code for MPA [18] so organizations can apply it to their own networks.

We also presented the results of applying MPA to over 850 data center networks operated by a large online service provider. In particular, we provided the first in-depth characterization of the network management practices used in modern data centers, and we identified several practices that strongly impact the frequency of problems in these networks: e.g., the number of devices, the number and type of configuration changes, and the number of device types. In several instances, these findings contradict network operators' perceptions: e.g., the fraction

of changes where an access control list is modified has a non-trivial impact on the frequency of network problems despite a majority opinion that the impact is low.

Checking Control Plane Correctness. Inspired by our observation that configuration changes have a strong impact on the frequency of network problems, we introduced a framework for proactively and efficiently detecting configuration errors that lead to functional failures in the presence of infrastructure faults (Chapter 3). Our *abstract representation for control planes* (ARC) [72] models a data center network's forwarding behavior at a higher level than other tools by taking advantage of the fact that: (1) data center networks tend to use a limited set of routing protocols which interact in very specific ways, and (2) important control plane analysis tasks often require computing *only properties* of forwarding paths, not the paths themselves. Consequently, verifying key invariants—e.g., traffic between subnets S_1 and S_2 is *always* blocked or *always* traverses a middlebox—boils down to computing simple characteristics of the graphs that makeup a control plane's ARC. Similarly, checking the equivalence of two control planes—e.g., that a simplified control plane always generates the same data plane as the original control plane under arbitrary infrastructure faults—simply requires comparing the graphs in each control plane's ARC. By applying our framework to a subset of the data center networks we analyzed in Chapter 2, we showed that proactive verification with ARC is three to five orders of magnitude faster than state-of-the-art network verification tools [66]. We have also made our implementation publicly available [1] to allow network operators to check their own data center networks.

Maintaining Middlebox Functionality and Performance. In Chapter 4, we introduced a framework that complements ARC by helping prevent functional and performance failures in middleboxes—another key element of data center networks and, as shown in Chapter 2 and other studies [75, 113], a strong contributor to data center network failures. Our middlebox state management framework, OpenNF [71, 73], facilitates the realization of a "one-big-middlebox" abstraction by enabling middlebox state to be replicated, transferred, or shared between middlebox instances. Moreover, OpenNF ensures that updates to middlebox state

are not lost or reordered and copies of state are kept eventually, strongly, or strictly consistent, so as to avoid causing a functional failure when middlebox state is transferred or replicated. We achieve this by introducing: (1) an event abstraction that allows a middlebox controller to closely observe/prevent updates to middlebox state, and (2) tracer packets that allow the controller to determine when all of a flow's outstanding packets have arrived at a middlebox instance. In our evaluation of OpenNF, we used three popular open-source middleboxes and traces of traffic exchanged with a cloud data center to show that these guarantees can be achieved with minimal latency overhead and resource demands thanks to a suite of novel optimizations.

OpenNF has garnered significant attention from both academia and industry. In the two years since we released the OpenNF source code [28], it has been downloaded by over 90 individuals and a few of them have published papers describing improvements in OpenNF's design [94]. Additionally, OpenNF was awarded the Internet Research Task Force's Applied Networking Research Prize for being "relevant for transitioning into shipping Internet products and related standardization efforts" [93]. This award exemplifies the fact that OpenNF addresses an important problem for which network operators seek solutions. Finally, both network operators and vendors, including Big Switch Networks, AT&T, Nokia, and ON.lab, have affirmed that OpenNF addresses a real problem they or their customers face. Consequently, there is a significant opportunity for OpenNF to be commercialized and deployed in real networks.

5.2 Future Work

While the work presented in this thesis has made important headway towards reducing functional and performance failures in data center networks through safer network management, there are still many opportunities for reducing data center network failures. Below, we highlight a few possible directions for future research.

Guiding network management. Knowing *why* an operator configured a data center

network in a particular way—i.e., what was their *intent*?—can help us guide them toward better design and operational strategies for achieving the same objective. However, like management practices, network operators' intentions are rarely recorded, so the first step is to discover the intent of specific actions. For example, network operators may add a VLAN to: (1) isolate hosts and achieve the intent of blocking communication, or (2) shrink the size of the broadcast domain and achieve the intent of reducing congestion. We can infer such intents by extending software engineering research designed for analyzing commits in code repositories [102], as well as applying natural language processing algorithms to ticket logs from incident management systems [115]. Once operator intentions are known, we can explore various machine learning techniques for grouping operator actions by their intent and produce a library of possible approaches for realizing specific intents. This library, along with the predictive model produced by MPA (Chapter 2), can be used to design a system that recommends alternative, less-failure-prone approaches to network operators.

Automating control plane configuration repairs. While ARC (Chapter 3) automatically locates network configuration errors, it leaves the task of repairing these errors in the hands of network operators. Although some issues are easy to fix, others require non-obvious changes across multiple devices and control protocols: e.g., enabling communication between a pair of hosts may require adding several new routing adjacencies. Thus, exploring how to *automatically* identify the *minimal repair* that fixes an error is an important problem.

Generating correct control plane configurations. Network configuration errors are primarily the result of network operators mis-translating their intents into low-level configurations. These errors could be avoided if configurations were automatically generated from operator intents using a provably correct process. This requires addressing two major questions: (1) What is the "right" abstraction for network operators to specify their intents? (2) How do we automatically generate correct configurations from these intents? The outcomes of the intent analysis discussed above can help guide the development of such an abstraction, as well as inform our choice of configuration constructs—e.g., which routing algorithms to

use—during the synthesis process.

5.3 Closing Remarks

The plethora of critical services hosted in data centers has made data centers, and their constituent components, an integral part of our daily lives. Moreover, their importance will only increase with the proliferation of the "Internet of Things". Consequently, taking steps to reduce and prevent data center failures, especially failures in foundational components such as the network, should be a top priority for data center operators. We believe this thesis makes important headway in this direction by arming operators with the metrics and tools they need to eliminate some of the most common causes of data center network failures. It is our hope that continued advancements along the directions pursued in this thesis will make future data center networks orders of magnitude more reliable than they are today.

A PROVING ARC IS COMPREHENSIVE AND PRECISE

In this appendix, we prove that our methodology for generating ETGs (Sections 3.3 and 3.4) results in comprehensive and precise ETGs.

A.1 Comprehensiveness

We first prove that the methodology presented in Section 3.3 results in comprehensive ETGs. We start by showing that a precise ETG is also comprehensive. (Proofs of ETG precision are sketched in Section A.2.)

Theorem A.1. *A precise ETG is comprehensive.*

Proof. Let path P be the min-cost path in the ETG from Src to Dst under some infrastructure fault. Now assume the actual network has a more preferred path P' between the source and destination, but P' does not exist in the ETG. Because P' does not exist in the ETG, the min-cost path in the ETG is incorrect. This contradicts the assumption that the ETG is precise. Thus, a precise ETG must contain every path taken by the actual network under all possible infrastructure faults.

Now assume the ETG contains a path P'' from Src to Dst which is infeasible in the actual network. Also assume all edges not on the path have been removed due to infrastructure faults. The only, and hence min-cost, path through the ETG will be P''. Because P'' is infeasible in the actual network, the min-cost path in the ETG is incorrect. This contradicts the assumption that the ETG is precise. Thus, a precise ETG must not contain any paths that are infeasible in the actual network. □

For some route redistribution policies we cannot generate a precise ETG (details in Section 3.4). However, the above methodology still generates a comprehensive ETG. We first show that an ETG only contains a path between vertices from different routing instances if

the instances (transitively) redistribute routes (Lemmas A.2 and A.3). Then we use this to prove the ETG is comprehensive (Theorem A.4).

Notation. Let I be the set of routing instances in the network and R_i be the set of routers participating in routing instance i. Let $<_{\mathsf{R}}$ be a *relation* denoting route redistribution between routing instances—i.e., if instance i''s routes are distributed to instance i'', then $i' <_{\mathsf{R}} i''$ (or) $(i', i'') \in <_{\mathsf{R}}$. In the degenerate case, routes of an instance are visible within the instance; thus, the relation satisfies *reflexivity* ($i <_{\mathsf{R}} i$). It is also *transitive*: if $i <_{\mathsf{R}} i'$ and $i' <_{\mathsf{R}} i''$, then $i <_{\mathsf{R}} i''$. Henceforth, i, i', ... are not necessarily distinct unless explicitly stated.

Lemma A.2. *If there exists a path of finite length in the ETG from $r_1.i'_{\mathrm{I}}$ to $r_2.i_{\mathrm{O}}$, for any r_1, r_2, then, $i <_{\mathsf{R}} i'$.*

Proof. By construction of the ETG as described above, there is an edge from $r.i'_{\mathrm{I}}$ to $r.i_{\mathrm{O}}$, if and only if $i <_{\mathsf{R}} i'$. Also, note that by construction there is no direct edge between two *in* (I) vertices and two *out* (O) vertices. Thus, the number of edges in the path is strictly odd. With these facts, we prove using induction on the path length.

For path length of 1, there exists a direct edge from $r_1.i'_{\mathrm{I}}$ to $r_2.i_{\mathrm{O}}$; thus, $i <_{\mathsf{R}} i'$.

Let us assume the lemma holds for any path of length $2\mathsf{k}-1$, where k is the number of routers in the path. A path of length $2\mathsf{k}+1$ can be broken down into 3 parts: a redistribution edge $r_1.i'_{\mathrm{I}} \rightarrow r_1.i''_{\mathrm{O}}$, an inter-device edge $r_1.i''_{\mathrm{O}} \rightarrow r_3.i''_{\mathrm{I}}$, and a path of length $2\mathsf{k}-1$ from $r_3.i''_{\mathrm{I}}$ to $r_2.i_{\mathrm{O}}$. The first edge implies, $i'' <_{\mathsf{R}} i'$, and by assumption that the lemma is true for paths of length $2\mathsf{k}-1$, we have, $i <_{\mathsf{R}} i''$. Thus, using the transitivity property of route distribution, we have $i <_{\mathsf{R}} i'$. □

Lemma A.3. *If there exists a path of finite length in the ETG from $r_1.i'_{\mathrm{I}}$ to $r_2.i_{\mathrm{I}}$, (or) $r_1.i'_{\mathrm{O}}$ to $r_2.i_{\mathrm{I}}$, (or) $r_1.i'_{\mathrm{O}}$ to $r_2.i_{\mathrm{O}}$, for any r_1, r_2, then $i <_{\mathsf{R}} i'$.*

Proof. Similar recursive proof as shown in Lemma A.2. □

Theorem A.4. *An ETG is comprehensive when routes are redistributed between OSPF, RIP, and/or eBGP instances.*

Proof. Let S_i be the set of routers in routing instance i that are directly connected to the source (i.e., $S_i \subseteq R_i$). and D_i be the set of routers in routing instance i that are directly connected to the destination (i.e., $D_i \subseteq R_i$). By construction of the ETG as describe above, there is an edge from SRC to $s.i_{\mathrm{O}}$ for each $s \in S_i$ and from $d.i'_{\mathrm{I}}$ to DST for each $d \in D_{i'}$ for all $i' \in I$.

Assume $i' \not<_{\mathsf{R}} i$ but there is a path from $s.i_{\mathrm{O}}$ to $d.i'_{\mathrm{I}}$; this implies the ETG contains a path that is infeasible in the real network under arbitrary infrastructure faults. However, according to Lemma A.3, if there exists a path from $s.i_{\mathrm{O}}$ to $d.i'_{\mathrm{I}}$, then $i' <_{\mathsf{R}} i$. This is a contradiction. Hence, the ETG must not contain any paths that are infeasible in the real network.

Now assume $i' <_{\mathsf{R}} i$ but there is no path from $s.i_{\mathrm{O}}$ to $d.i'_{\mathrm{I}}$. Assuming advertisements/-traffic for the destination prefix are not blocked by route filters or ACLs, this implies the ETG does not contain every path between the source and destination endpoints that is used in the real network under arbitrary infrastructure faults. However, assume $i' <_{\mathsf{R}} i''$ and there is a path from $s.i_{\mathrm{O}}$ to $r.i''_{\mathrm{I}}$ and a path from $r.i'_{\mathrm{O}}$ to $d.i'_{\mathrm{I}}$. By Lemma A.3 $i'' <_{\mathsf{R}} i$, and transitivity $i' <_{\mathsf{R}} i$. By construction of the ETG as described above, we add an edge from $r.i''_{\mathrm{I}}$ to $r.i'_{\mathrm{O}}$ if and only if $i' <_{\mathsf{R}} i''$. Thus, the ETG contains a path from $s.i_{\mathrm{O}}$ to $d.i'_{\mathrm{I}}$. This is a contradiction. Hence, the ETG must contain every path that is feasible in the real network. □

A.2 Precision

We now show that the methodology presented in Section 3.4 results in precise ETGs. We start by considering scenarios without route redistribution.

Theorem A.5. *An ETG is precise when the source and destination are connected to routers participating in the same RIP or single-area OSPF routing instance and no routes are redistributed into the instance.*

Proof. The weights assigned to inter-device edges within the routing instance are proportional to the link weights used by OSPF or RIP. Because no routes are redistributed into the instance, a process's *in* vertex will only be connected to its own *out* vertex and the edge will be assigned a weight of 0. Contracting such edges does not impact the set of possible paths, nor their weights, in the ETG. The contracted graph will be equivalent to the graph used by OSPF or RIP to compute routes. Thus, the ETG is precise. □

Theorem A.6. *An ETG is precise when the source and destination are connected to routers participating in the same eBGP routing instance, AS path length is the only selection criterion used by the eBGP processes, and no routes are redistributed into the instance.*

Proof. Each inter-device edge between eBGP processes represents an AS hop. Assigning the same weight to every edge results in the weight of a path being proportional to the number of hops, which we assume is the only selection criteria. □

Now we consider scenarios with route redistribution.

Notation. We use the same notation presented in Section A.1. Additionally, let $B_{i',i''}$ be the set of routers that are configured to redistribute routes from routing instance i' to instance i''. Although a router $b \in B_{i',i''}$ may be configured to redistribute routes from routing instance i' to instance i'', routes redistribution actually occurs only if both of the following conditions hold: (1) the routing process for instance i' has a route to the destination, and (2) no higher priority process on b has a route to the destination. We denote the set of routers that actively distribute routes to the destination between i' and i'' by $B^{active}_{i',i''} \subseteq B_{i',i''}$.

Let $c_{i',i''}$ denote the cost of distributing routes from routing instance i' to instance i''.[1] These redistribution costs are included in the cost of intra-device edges in the ETG: e.g., an edge from a vertex $b.i''_{\mathrm{I}}$ to a vertex $b.i'_{\mathrm{O}}$ for some router $b \in B_{i',i''}$.

First, we extend our earlier lemmas to show that any routing instance a path traverses must (transitively) redistribute routes.

[1] Without loss of generality, we assume $c_{i',i''}$ is set to the same value on all routers in $B_{i',i''}$.

Lemma A.7. *If $i' <_R i$ and $i'' <_R i$, then any path through the ETG from vertex $r.i'_*$ to $r.i''_*$ does not contain a vertex $r.i'''_*$ such that $i''' \not<_R i$.*

Proof. By contradiction. If there exists such a vertex, then using above Lemma A.2, $i''' <_R i'$. Then, using transitivity, $i''' <_R i$. Thus, there exists no such vertex for which $<_R$ is not satisfied. □

Lemma A.8. *Any path from vertex $r.i_*$ to* Dst*, does not contain a vertex $r.i'''_*$ such that $i''' \not<_R i$.*

Proof. This lemma is a special case of Lemma A.7, if $i' = i$ and Dst is directly to attached to a router in instance i''. □

Next, we show that a path between two vertices associated with the same routing instance cannot pass through a vertex associated with a different routing instance, because this is not allowed in the actual network when route redistribution is a DAG.

Lemma A.9. *If $<_R$ is a partial order, then for $r_1, r_2 \in R_i$ and $|I_{r_1}| = |I_{r_2}| = 1$, any path in the ETG from r_1 to r_2, will not contain vertices of the form $r.i'_{\mathrm{I}}$ or $r.i'_{\mathrm{O}}$, where $i \neq i'$.*

Proof. If there exists a vertex of the form $r.i'_{\mathrm{I}}$ or $r.i'_{\mathrm{O}}$, then from Lemma A.2 and Lemma A.3, we have $i' <_R i$ and $i <_R i'$. The relation is thus not a partial order, violating our assumption. □

Now we show the path within a routing instance is precise.

Lemma A.10. *If $<_R$ is a partial order, then for routers $r_1, r_2 \in R_i$ and $|I_{r_1}| = |I_{r_2}| = 1$, the path in the ETG from r_1 to r_2, is identical to the shortest path computed by instance i.*

Proof. Using Lemma A.9, we know that, under the assumptions, all path are restricted to vertices in the ETG belonging to the same instance: i.e., $r.i_{\mathrm{I}}$ or $r.i_{\mathrm{O}}$, for $r \in I_r$. By construction of the ETG, we know that all the costs of vertices corresponding to a single routing instance are a multiple of the actual configured costs. Thus, shortest paths calculated

using the scaled down version of the actual costs are identical. From Theorems A.5 and A.6, the path within a routing instance is precise. □

Lemma A.11. *If $<_{\mathsf{R}}$ is a partial order and $i' <_{\mathsf{R}} i$ and $i'' <_{\mathsf{R}} i$, then the length of any path in the ETG from vertex $r.i'_*$ to $r.i''_*$ is less that the least difference between paths with vertices of instance i, denoted by g_i.*

Proof. By construction. As described above, if a routing instance distributes its route to another instance, its link costs in the ETG are scaled down by an order of magnitude. Together with Lemma A.7 we can say that all edges in the path have cost an order of magnitude less. □

When multiple routing processes on router s have a route to the destination, s chooses the route from the highest priority process, according to administrative distance (AD). The priority of routing instance i on router r is denoted by p_r^i; the instance with the lowest AD has priority 0, and the instance with the highest AD has priority $|I_r| - 1$

Axiom A.12. *For a router, b, belonging to multiple routing instances, if $b \in B_{i',i}^{active}$, then $b \notin B_{i'',i}^{active}$ for all, i'' such that $p_b^{i''} \neq p_b^{i'}$.*

Theorem A.13. *An ETG is precise when the redistribution of routes between OSPF, RIP, and/or eBGP processes is acyclic, and the fixed costs assigned to redistributed routes are congruent with the redistributor's AD.*

Proof. We will prove precision through induction on the routing instance. Let us assume that, for all $i : i <_{\mathsf{R}} i'$, the shortest path from vertex $r.i_{\mathsf{O}} \in R_i$ to Dst is precise. Under the assumption, we will then show that the path from $r.i'_{\mathsf{O}}$ to Dst is precise. The assumption we make is strong but valid, since, (1) using Lemma A.8, we know that any path to Dst only has vertices of instances that redistribute routes to i', and, (2) $<_{\mathsf{R}}$ is a partial order and hence there is no cyclical dependency between instances whose vertices have a precise path to the destination.

If $r.i'_{\mathrm{O}}$ belongs in a path to the destination, it implies that the routing process for instance i' on router r was the dominant process based on administrative distance and route availability to the destination. From the perspective of $r \in R_{i'}$, the minimum distance to the destination is given by:

$$\begin{aligned} \mathsf{H}(\mathfrak{r}, \textsc{Dst}) = \min \Bigg(& \min_{d \in D_{i'}} \mathsf{H}_{i'}(\mathfrak{r}, d), \\ & \min_{i, \mathfrak{b} \in B^{active}_{i,i'}} (\mathsf{H}_{i'}(\mathfrak{r}, \mathfrak{b}) + c_{i,i'}) \Bigg) \end{aligned} \tag{A.1}$$

where, $\mathsf{H}_{i'}(\mathfrak{x}, \mathfrak{y})$ is the distance from $\mathfrak{x}$ to $\mathfrak{y}$ as seen by i'. The actual path taken depends on which term in the right hand side of Equation A.1 achieves the minimum. If $D_{i'} = \varnothing$ and $B_{i,i'} = \varnothing, \forall i$, then there is no path to vertex Dst.

If one of $d \in D_{i'}$ achieves the minimum, then using Lemma A.10 we can show the shortest path to vertex Dst is precise.

Otherwise, if one of the terms in $\min_{i, \mathfrak{b} \in B^{active}_{i,i'}} \mathsf{H}_{i'}(\mathfrak{r}, \mathfrak{b}) + c_{i,i'}$ (say i^* and b_*) achieves the minimum, then the destination is reached through another routing instance. We show that the shortest path in the ETG from vertex $r.i'_{\mathrm{O}}$ to Dst passes through vertex $b_*.i^*_{\mathrm{O}}$.

Let us assume the shortest path to the destination does not go through, $b_*.i^*_{\mathrm{O}}$. Then there exists another vertex $b_\circ.i^\circ_{\mathrm{O}}$ which the path traverses to reach Dst.

Case 1: $b_\circ = b_*$ and $p^{i^\circ}_{b_*} > p^{i^*}_{b_*}$.

By assumption, we know the path from $b_*.i^\circ_{\mathrm{O}}$ is precise. If i° has a path to the destination, then $b_* \in B^{active}_{i^\circ,i'}$. Then, using Axiom A.12, $b_* \notin B^{active}_{i^*,i'}$. Thus, b_*, i^* would not be the minimizing term in Equation A.1 violating our assumption. Hence, i° has no path to the destination, and cannot be the shortest path.

Case 2: $b_\circ = b_*$ and $p^{i^\circ}_{b_*} < p^{i^*}_{b_*}$ (or) $b_\circ \neq b_*$.

Since it is the shortest path, we require

$$\begin{aligned} &\mathsf{F}(r.i'_{\mathrm{O}}, b_{\circ}.i^{\circ}{}_{\mathrm{O}}) + \mathsf{F}(b_{\circ}.i^{\circ}{}_{\mathrm{O}}, \textsc{Dst}) \\ &\quad < \mathsf{F}(r.i'_{\mathrm{O}}, b_{*}.i^{*}{}_{\mathrm{O}}) + \mathsf{F}(b_{*}.i^{*}{}_{\mathrm{O}}, \textsc{Dst}) \end{aligned} \tag{A.2}$$

where, $\mathsf{F}(.,.)$ is the distance along the shortest path in the ETG.

But from Lemma A.11 we know that,

$$\mathsf{F}(r.i'_{\mathrm{O}}, b_{\circ}.i^{\circ}{}_{\mathrm{O}}) - \mathsf{F}(r.i'_{\mathrm{O}}, b_{*}.i^{*}{}_{\mathrm{O}}) > g_{i'} \tag{A.3}$$

and,

$$\begin{aligned} 0 \leqslant \mathsf{F}(b_{\circ}.i^{\circ}{}_{\mathrm{O}}, \textsc{Dst}) < g_{i'} \\ 0 \leqslant \mathsf{F}(b.i_{\mathrm{O}}, \textsc{Dst}) < g_{i'} \end{aligned} \tag{A.4}$$

From Equations A.2 and A.4, we have

$$\begin{aligned} &\mathsf{F}(r.i'_{\mathrm{O}}, b_{\circ}.i^{\circ}{}_{\mathrm{O}}) - \mathsf{F}(r.i'_{\mathrm{O}}, b_{*}.i^{*}{}_{\mathrm{O}}) \\ &\quad < +\mathsf{F}(b_{*}.i^{*}{}_{\mathrm{O}}, \textsc{Dst}) - \mathsf{F}(b_{\circ}.i^{\circ}{}_{\mathrm{O}}, \textsc{Dst}) < g_{i'} \end{aligned} \tag{A.5}$$

Equation A.3 contradicts Equation A.5. Thus, there cannot exist $b_{\circ} \neq b_{*}$ and $i^{\circ} \neq i^{*}$. $\square$

B PROVING OPENNF'S MOVE OPERATION IS LOSS-FREE AND ORDER-PRESERVING

In this appendix, we prove that OpenNF's move operation is loss-free and order-preserving. As discussed in Section 4.4, we assume TCP-based control channels are used between the OpenNF controller and middleboxes, so southbound API calls, state, and events are not lost or reordered. Furthermore, packets may be lost (but we assume not reordered) on the network paths from *gw* to *srcInst* and *gw* to *dstInst*.

Notation. Let p_i be the $\mathfrak{i}^{\text{th}}$ packet for a flow f that arrives at *gw* and $\langle p \rangle_{i,j}$ be the sequence of packets from $\mathfrak{i}$ to $\mathfrak{j}$. Also, let $S_{i,j} = \phi_{S_{init}}(\langle p \rangle_{i,j})$ be the value of the per-flow state for f after processing $\langle p \rangle_{i,j}$ starting from initial state S_{init}.

The controller issues all southbound API calls and route updates so we can definitively order the actions in time:

- t_1 Enable events and packet dropping for f on *srcInst*
- t_2 Get per-flow state S from *srcInst*
- t_3 Put per-flow state S to *dstInst*
- t_4 Extract packets from events buffered on controller and send to *dstInst*
- t_5 Enable events and packet buffering for f on *dstInst*
- t_6 Change the route for f on *gw* to forward to *dstInst*
- t_7 Send a tracer packet to *gw* to forward to *srcInst*
- t_8 Disable events and release packet buffer for f on *dstInst*

We use the time points to refer to the completion of each action from the perspective of the node on which the action is performed.

Theorem B.1. *If a control application requests a loss-free move, OpenNF guarantees all state updates resulting from packet processing are reflected at the destination instance, and all packets the switch receives should be processed.*

Proof. Let p_k $(1 < \mathtt{k} < \mathtt{n})$ be the first packet for f to arrive at *gw* after t_6. Then, $\langle p \rangle_{1,k-1}$ will be forwarded to *srcInst*, and $\langle p \rangle_{k,n}$ will be forwarded to *dstInst*.

Of the packets forwarded to *srcInst*, let p_j $(1 < \mathtt{j} < \mathtt{k})$ be the first packet for f to be dequeued at *srcInst* after t_1. Then, $\langle p \rangle_{1,j-1}$ will be processed at *srcInst* before t_1, resulting in per-flow state $S_{1,j-1}$. In contrast, $\langle p \rangle_{j,k-1}$ will be sent to the controller in events and dropped at *srcInst*.

Since no packets are processed at *srcInst* after t_1, the state $S_{1,j-1}$ will be exported from *srcInst* at t_2 and imported on *dstInst* at t_3. Also, since no packets are forwarded to *dstInst* before t_6, there won't be state S to overwrite or combine during the import at *dstInst*.

Events for $\langle p \rangle_{j,k-1}$ may arrive at the controller anytime after t_1. If they arrive before t_3, they are buffered. Starting at t_3, packets are extracted from the buffered events and sent to *gw* to be forwarded to *dstInst*. If events from *srcInst* arrive at the controller after the buffer is empty (i.e., after t_4), the packets they contain are immediately sent to *dstInst*. Since the control channel from the controller to *dstInst* is reliable, all $p_i \in \langle p \rangle_{j,k-1}$ will arrive at *dstInst* and be processed.

After t_6, $\langle p \rangle_{k,n}$ will arrive at *dstInst*. They will be processed as they arrive. In summary, $\langle p \rangle_{1,j-1}$ will be processed at *srcInst*, with S_{init} as the initial per-flow state, and $\langle p \rangle_{j,n}$ will be processed at *dstInst*, with $S_{1,j-1}$ as the initial per-flow state, implying move is loss-free. □

Theorem B.2. *If a control application requests an order-preserving move, OpenNF guarantees all packets are processed by a middlebox instance in the order they were forwarded to the middlebox instance by the switch.*

Proof. As above, let p_k be the first packet for f to arrive at *gw* after t_6 and p_j be the first packet for f to be dequeued at *srcInst* after t_1. Then, $\langle p \rangle_{1,j-1}$ will arrive and be processed at *srcInst* in order before t_1. Similarly, $\langle p \rangle_{k,n}$ will arrive (after t_6) and be processed (after t_8) at *dstInst* in order. To guarantee order-preserving, *dstInst* must have $S_{1,k-1}$ by t_8. From above, we know *dstInst* will have $S_{1,j-1}$ by t_3. Thus, we need to show that *dstInst* will receive and process $\langle p \rangle_{j,k-1}$ in order after t_3 but before t_8.

Events for $\langle p \rangle_{j,k-1}$ may arrive at the controller anytime after t_1. Events arriving before t_3 are buffered, while events arriving after t_4 are handled immediately. The controller will extract the packets from these events, mark the packets with a "do-not-buffer" flag, and send them to *dstInst*. Since $\langle p \rangle_{j,k-1}$ is sent from *srcInst* to the controller through a TCP channel and from the controller to *dstInst* through a TCP channel, the order of packets within this sequence is preserved and none will be lost. The packets will be processed at *dstInst* as they arrive, resulting in per-flow state $S_{1,k-1}$ at *dstInst*.

Since a tracer packet is not sent to *gw* to be forwarded to *srcInst* until after t_6, no other packets for f will arrive at *srcInst* after the tracer packet (assuming no reordering occurs on the path from *gw* to *srcInst*). Thus, all events from *srcInst* for $\langle p \rangle_{j,k-1}$ will arrive at the controller before an event for the tracer packet. Furthermore, an event for packet p_{k-1} (or an earlier packet in $\langle p \rangle_{j,k-1}$ if p_{k-1} is dropped on the path from *gw* to *srcInst*) will be the last event to arrive before an event for the tracer packet. This allows the controller to know that p_{k-1} was the last packet forwarded to *srcInst*.

Once the controller receives an event for p_{k-1} from *dstInst*, it knows *dstInst* has processed $p_{1,k-1}$ and has the state $S_{1,k-1}$. Therefore, the controller can guarantee that *dstInst* has state $S_{1,k-1}$ by t_8 and move is order-preserving. □

The proofs can be extended to moves involving multi-flow state by expanding the notion of flow to actually refer to a group of flows.

BIBLIOGRAPHY

[1] Abstract representation for control planes. http://bitbucket.org/uw-madison-networking-research/arc.

[2] Apache Kafka. http://kafka.apache.org.

[3] Balance. http://inlab.de/balance.html.

[4] BGP best path selection algorithm. http://cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/13753-25.html.

[5] Boost C++ libraries. http://boost.org.

[6] Bro 2.1 documentation: detect-mhr.bro. http://bro.org/sphinx-git/scripts/policy/frameworks/files/detect-MHR.bro.html.

[7] C++ Middleware Writer. http://webebenezer.net.

[8] CRIU: Checkpoint/Restore In Userspace. http://criu.org.

[9] Floodlight OpenFlow Controller. http://floodlight.openflowhub.org.

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

[11] HP OpenView TrueControl software. http://support.openview.hp.com.

[12] HPROF. http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html.

[13] iptables. http://netfilter.org/projects/iptables.

[14] ITIL – IT service management. http://www.axelos.com/best-practice-solutions/itil.

[15] JGraphT. http://jgrapht.org.

[16] libnetfilter_conntrack project. http://netfilter.org/projects/libnetfilter_conntrack.

[17] Malware-traffic-analysis.net. http://malware-traffic-analysis.net.

[18] Management plane analytics tool. https://github.com/agember/mpa.

[19] nDPI. http://ntop.org/products/ndpi.

[20] Network functions virtualisation: Introductory white paper. http://www.tid.es/es/Documents/NFV_White_PaperV2.pdf.

[21] NIST/SEMATECH e-handbook of statistical methods. http://itl.nist.gov/div898/handbook.

[22] OpenMano. https://github.com/nfvlabs/openmano.

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

[24] Really Awesome New Cisco confIg Differ (rancid). http://shrubbery.net/rancid.

[25] Redistributing routing protocols. http://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/8606-redist.html.

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

[27] Traceroute.org. http://www.traceroute.org.

[28] OpenNF source code. http://opennf.cs.wisc.edu/code/, 2013.

[29] Cost of data center network outages. http://www.emersonnetworkpower.com/en-US/Resources/Market/Data-Center/Latest-Thinking/Ponemon/Pages/2016-Cost-of-Data-Center-Outages-Report.aspx, 2016.

[30] R. Aharoni and E. Berger. Menger's theorem for infinite graphs. *Inventiones mathematicae*, 2008.

[31] E. S. Al-Shaer and H. H. Hamed. Discovery of policy anomalies in distributed firewalls. In *INFOCOM*, 2004.

[32] M. A. Alim and T. G. Griffin. On the interaction of multiple routing algorithms. In *CoNEXT*, 2011.

[33] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. In *SIGCOMM*, 2008.

[34] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[35] A. Anand, V. Sekar, and A. Akella. SmartRE: An architecture for coordinated network-wide redundancy elimination. In *SIGCOMM*, 2009.

[36] J. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible open middleboxes with commodity servers. In *ANCS*, 2012.

[37] A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network. https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network.

[38] A. Balachandran, V. Sekar, A. Akella, S. Seshan, I. Stoica, and H. Zhang. Developing a predictive model of quality of experience for internet video. In *SIGCOMM*, 2013.

[39] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[40] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *NSDI*, 2009.

[41] T. Benson, A. Akella, and A. Shaikh. Demystifying configuration challenges and trade-offs in network-based ISP services. In *SIGCOMM*, 2011.

[42] T. Benson, S. Sahu, A. Akella, and A. Shaikh. A first look at problems in the cloud. In *HotCloud*, 2010.

[43] C. Bird, B. Murphy, N. Nagappan, and T. Zimmermann. Empirical software engineering at Microsoft Research. In *Computer Supported Cooperative Work (CSCW)*, 2011.

[44] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, 1987.

[45] K. P. Birman, A. Schiper, and P. Stephenson. Lightweigt causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, 1991.

[46] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[47] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford. The cutting EDGE of IP router configuration. *SIGCOMM Computer Communication Review*, 34(1):21–26, Jan. 2004.

[48] L. D. Carli, R. Sommer, and S. Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *CCS*, 2014.

[49] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[50] A. Cockcroft. A closer look at the Christmas Eve outage. `http://techblog.netflix.com/2012/12/a-closer-look-at-christmas-eve-outage.html`, December 2012.

[51] P. Comon. Independent component analysis, a new concept? *Signal Processing*, 36(3):287–314, Apr. 1994.

[52] B. Cully, G. Lefebvre, D. T. Meyer, M. Feeley, N. C. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, page 161, 2008.

[53] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[54] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. E. Anderson, and A. Krishnamurthy. ETTM: A scalable fault tolerant network manager. In *NSDI*, 2011.

[55] M. Dobrescu, K. J. Argyraki, and S. Ratnasamy. Toward predictable performance in software packet-processing platforms. In *NSDI*, 2012.

[56] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. COLO: COarse-grained LOck-stepping virtual machines for non-stop service. In *SoCC*, 2013.

[57] F. Douglis and J. K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software: Practice and Experience.*, 21(8):757–785, 1991.

[58] A. B. Downey. Using pathchar to estimate internet link characteristics. In *SIGCOMM*, 1999.

[59] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, 2014.

[60] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *CCS*, 2004.

[61] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the resource consumption of network intrusion detection systems. In *RAID*, 2008.

[62] K. El-Arini and K. Killourhy. Bayesian detection of router configuration anomalies. In *Proceedings of the ACM SIGCOMM Workshop on Mining Network Data (MineNet)*, 2005.

[63] S. K. Fayazbakhsh, L. Chaing, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *NSDI*, 2014.

[64] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, 2005.

[65] A. Feldmann and J. Rexford. IP network configuration for intradomain traffic engineering. *IEEE Network*, 15(5):46–57, Sep 2001.

[66] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

[67] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, Aug. 1997.

[68] E. W. Fulp. Optimization of network firewall policies using directed acyclic graphs. In *IEEE Internet Management Conference*, 2005.

[69] P. Garimella, Y. Sung, N. Zhang, and S. Rao. Characterizing VLAN usage in an operational network. In *SIGCOMM Workshop on Internet Network Management*, 2007.

[70] A. Gember, A. Krishnamurthy, S. St. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. Technical Report arXiv:1305.0209, 2013.

[71] A. Gember-Jacobson and A. Akella. Improving the safety, scalability, and efficiency of network function state transfers. In *HotMiddlebox*, 2015.

[72] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *SIGCOMM*, 2016.

[73] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *SIGCOMM*, 2014.

[74] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *IMC*, 2015.

[75] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, 2011.

[76] T. G. Griffin and J. L. Sobrinho. Metarouting. In *SIGCOMM*, 2005.

[77] H. Hamed, E. Al-Shaer, and W. Marrero. Modeling and verification of IPSec and VPN security policies. In *IEEE International Conference on Network Protocols (ICNP)*, 2005.

[78] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in EC2 and Azure. In *IMC*, 2013.

[79] M. Hollander and D. Wolfe. *Nonparametric statistical methods*. Wiley, 1973.

[80] M. Jain and C. Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput. In *SIGCOMM*, 2002.

[81] A. Jaquith. *Security Metrics: Replacing Fear, Uncertainty, and Doubt*. Addison-Wesley, 2007.

[82] D. D. Jensen, A. S. Fast, B. J. Taylor, and M. E. Maier. Automatic identification of quasi-experimental designs for discovering causal knowledge. In *KDD*, 2008.

[83] D. Joseph and I. Stoica. Modeling middleboxes. *IEEE Network*, 2008.

[84] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *SIGCOMM*, 2008.

[85] Juniper Networks. Understanding BGP path selection. http://juniper.net/documentation/en_US/junos12.1/topics/reference/general/routing-ptotocols-address-representation.html.

[86] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless network functions. In *HotMiddlebox*, 2015.

[87] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[88] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[89] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for nfv: Simplifying middlebox modifications using statealyzr. In *NSDI*, 2016.

[90] T. M. Khoshgoftaar, M. Golawala, and J. Van Hulse. An empirical study of learning from imbalanced data using random forest. In *International Conference on Tools with Artificial Intelligence (ICTAI)*, 2007.

[91] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[92] H. Kim, T. Benson, A. Akella, and N. Feamster. The evolution of network configuration: A tale of two campuses. In *IMC*, 2011.

[93] O. Kolkman. Rough guide to IETF 92: Welcome to Texas, y'all! https://www.internetsociety.org/blog/tech-matters/2015/03/rough-guide-ietf-92-welcome-texas-yall.

[94] B. Kothandaraman, M. Du, and P. Sköldström. Centrally controlled distributed VNF state management. In *HotMiddlebox*, 2015.

[95] S. S. Krishnan and R. K. Sitaraman. Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. In *IMC*, 2012.

[96] S. S. Krishnan and R. K. Sitaraman. Understanding the effectiveness of video ads: A measurement study. In *IMC*, 2013.

[97] S. D. Krothapalli, X. Sun, Y.-W. E. Sung, S. A. Yeo, and S. G. Rao. A toolkit for automating and visualizing VLAN configuration. In *SafeConfig*, 2009.

[98] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb. Detecting network-wide and router-specific misconfigurations through data mining. *IEEE/ACM Transactions on Networking*, 17(1):66–79, Feb 2009.

[99] F. Le, G. G. Xie, D. Pei, J. Wang, and H. Zhang. Shedding light on the glue logic of the internet routing architecture. In *SIGCOMM*, 2008.

[100] F. Le, G. G. Xie, and H. Zhang. Theory and new primitives for safely connecting routing protocol instances. In *SIGCOMM*, 2010.

[101] W. Li and A. W. Moore. A machine learning approach for efficient traffic classification. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2007.

[102] D. R. Licata, C. D. Harris, and S. Krishnamurthi. The feature signatures of evolving programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 281–285, 2003.

[103] M. Litzkow and M. Livny. Supporting checkpointing and process migration outside the UNIX kernel. In *USENIX ATC*, 1992.

[104] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *SOSP*, 2003.

[105] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[106] D. A. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjálmtýsson, and A. Greenberg. Routing design in operational networks: A look from the inside. In *SIGCOMM*, 2004.

[107] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *NSDI*, 2014.

[108] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, Mar. 2008.

[109] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *OSDI*, 2002.

[110] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.

[111] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: a framework for NFV applications. In *SOSP*, 2015.

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

[113] R. Potharaju and N. Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *IMC*, 2013.

[114] R. Potharaju and N. Jain. When the network crumbles: an empirical study of cloud network failures and their impact on services. In *SoCC*, 2013.

[115] R. Potharaju, N. Jain, and C. Nita-Rotaru. Juggling the jigsaw: Towards automated problem inference from network trouble tickets. In *NSDI*, 2013.

[116] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *SIGCOMM*, 2013.

[117] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[118] P. Quinn and U. Elzur. Network Service Header. Internet-Draft draft-ietf-sfc-nsh-04, IETF Secretariat, March 2016.

[119] P. Quinn and T. Nadeau. Problem statement for service function chaining. RFC 7498, April 2015.

[120] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A high availability framework for middleboxes. In *SoCC*, 2013.

[121] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System support for elastic execution in virtual middleboxes. In *NSDI*, 2013.

[122] D. B. Rubin. Using multivariate matched sampling and regression adjustment to control bias in observational studies. *Journal of the American Statistical Association*, 74:318–328, 1979.

[123] L. Schaelicke, K. B. Wheeler, and C. Freeland. SPANIDS: A scalable network intrusion detection loadbalancer. In *Conference on Computing Frontiers (CF)*, pages 315–322, 2005.

[124] V. Sekar, N. Egi, S. Ratnasamy, M. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *NSDI*, 2012.

[125] W. Shadish, T. Cook, , and D. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin, 2002.

[126] S. Shankland. Amazon cloud outage derails Reddit, Quora. `http://cnet.com/news/amazon-cloud-outage-derails-reddit-quora`, April 2011.

[127] J. Sherry, P. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Macciocco, M. Manesh, J. Martins, S. Ratnasamy, and L. R. S. Shenker. Rollback recovery for middleboxes. In *SIGCOMM*, 2015.

[128] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *SIGCOMM*, 2012.

[129] A. Slivkins. Parameterized tractability of edge-disjoint paths on directed acyclic graphs. *SIAM Journal of Discrete Mathematics*, 24(1):146–157, Feb. 2010.

[130] J. L. Sobrinho. Algebra and algorithms for qos path computation and hop-by-hop routing in the internet. *IEEE/ACM Transactions on Networking (TON)*, 10(4):541–550, 2002.

[131] J. L. Sobrinho. Network routing with path vector protocols: theory and applications. In *SIGCOMM*, 2003.

[132] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[133] T. Spetebroot, S. Afra, N. Aguilera, D. Saucez, and C. Barakat. From network-level measurements to expected quality of experience: The Skype use case. In *IEEE International Workshop on Measurements & Networking (M&N)*, 2015.

[134] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with Rocketfuel. *IEEE/ACM Transactions on Networking (TON)*, 2004.

[135] E. A. Stuart. Matching methods for causal inference: A review and a look forward. *Statistical Science*, 25, 2010.

[136] E. A. Stuart and D. B. Rubin. Best practices in quasi-experimental designs: Matching methods for causal inference. In *Best Practices in Quantitative Methods*, pages 155–176. Sage, 2008.

[137] Y. Sung, S. Rao, S. Sen, and S. Leggett. Extracting network-wide correlated changes from longitudinal configuration data. In *PAM*, 2009.

[138] Y. Sverdlik. Microsoft: Misconfigured network device led to Azure outage. `http://datacenterdynamics.com/servers-storage/microsoft-misconfigured-network-device-led-to-azure-outage/68312.fullarticle`, July 2012.

[139] D. Turner, K. Levchenko, J. C. Mogul, S. Savage, and A. C. Snoeren. On failure in managed enterprise networks. Technical Report HPL-2012-101, HP.

[140] D. Turner, K. Levchenko, S. Savage, and A. C. Snoeren. A comparison of syslog and IS-IS for network failure analysis. In *IMC*, 2013.

[141] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: Understanding the causes and impact of network failures. In *SIGCOMM*, 2010.

[142] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In *RAID*, 2007.

[143] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. Central control over distributed routing. In *SIGCOMM*, pages 43–56, 2015.

[144] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based server load balancing gone wild. In *Hot-ICE*, 2011.

[145] G. C. Whittaker. Network outages like NYSE, United Airlines, are the new natural disasters. http://popsci.com/network-outages-nyses-united-airlines-are-new-natural-disasters, July 2015.

[146] Z. Whittaker. Amazon Web Services suffers outage, takes down Vine, Instagram, others with it. http://zdnet.com/article/amazon-web-services-suffers-outage-takes-down-vine-instagram-others-with-it, August 2013.

[147] X. Wu, D. Turner, C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating datacenter network failure mitigation. In *SIGCOMM*, 2012.

[148] K. Xinidis, I. Charitakis, S. Antonatos, K. G. Anagnostakis, and E. P. Markatos. An active splitter architecture for intrusion detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 3(1):31–44, 2006.

[149] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: A toolkit for FIREwall Modeling and ANalysis. In *IEEE Symposium on Security and Privacy (SP)*, 2006.

[150] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *CoNEXT*, 2012.