Reasoning about Stateful Network Behaviors

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Seyed K. Fayaz
B.S., Computer Engineering, Tehran Polytechnic
M.S., Computer Science, Stony Brook University

Carnegie Mellon University
Pittsburgh, PA 15213

December 2016

Reasoning about Stateful Network Behaviors

Acknowledgments

I would like to thank everyone who helped me have a great Ph.D. journey. First, I want to thank my family who wholeheartedly supported my choice of getting a Ph.D. degree. Second, I am indebted to my Ph.D. advisor Vyas Sekar for believing in me and giving me the freedom and support to work on exciting research problems. Third, I would like to thank the wonderful students around me who helped me master the technical and personal skills that I needed to succeed. In particular, I enjoyed working with Yoshiaki Tobioka and Tianlong Yu, who in addition to being great hackers, are incredible individuals.

Besides the folks I frequently interacted with, there are many people who helped me grow along the way. I was fortunate to work with and learn from some of the brightest minds in the fields of networking, systems, and security: Michael Bailey, Teemu Koponen, Bruce Maggs, Ratul Mahajan, Jeff Mogul, Mike Reiter, Srini Seshan, Scott Shenker, George Varghese, and Minlan Yu. I found Tuesday systems seminars at the CS department of CMU well-run and extremely useful owing to the bright students and faculty who attended and provided the presenters with actionable feedback. This fabulous group includes Dave Andersen, Junchen Jiang, Serhat Kiyak, Hyeontaek Lim, Matt Mukerjee, David Naylor, Wolf Richter, Srini Seshan, Peter Steenkiste, Yuchen Wu, and Dong Zhou.

Some of my best learning experiences were at the times when I was pushed outside my comfort zone. First, my internship at Microsoft under the mentorship of Ratul Mahajan and George Varghese was a fun and sobering experience. In particular, Ratul taught me how to effectively communicate with network operators. My meetings with brilliant Microsoft Azure engineers opened a whole new window into my perception of real-world problems and solutions. Second, as part of my work on network verification, Sagar Chaki and Todd Millstein helped me learn quite a bit on formal methods and program analysis.

As a Ph.D. student, I spent most of my time at CMU CyLab. I would like to thank David Brumley and Virgil Gligor for doing a phenomenal job in turning CyLab into a remarkable cybersecurity research center. I also want to thank Toni Fox who worked diligently to make sure I had a meeting room whenever I needed one. My Ph.D. was financially supported by the NSF, a VMware Graduate Fellowship, and the CMU Bertucci Fellowship.

Abstract

Network operators must ensure their networks meet intended traversal policies (e.g., host A can talk to host B, or inbound traffic to host C goes through a firewall and then a NAT). Violations of the policies may result in revenue loss, reputation damage, and security breaches. Today checking whether the intended policies are enforced correctly is stymied by two fundamental sources of complexity: the *diversity* and *stateful* nature of the behaviors of real networks. First, we need to account for vast diversity in both the control plane (e.g., different routing protocols and their interactions) and the data plane (e.g., routers, firewalls, and proxies) of the network. Second, we need to reason about a very large space of stateful behaviors in both the control plane (e.g., the current state being characterized by the route advertisements the routers have seen so far) and the data plane (e.g., a firewall's current state with respect to a TCP session). Prior work on checking network policies is limited to a particular state of the network. Any attempt to reason about the behavior of the network across its state space is hindered by two fundamental challenges: *(i)* capturing the diversity of the control and data planes, and *(ii)* exploring the state space of the control and data planes in a scalable manner.

This thesis argues for the feasibility of checking the correctness of realistic network policies by addressing the above challenges via two key insights. First, to combat the challenge of diversity, we design unifying abstractions that glue together different routing protocols in the control plane and diverse network appliances (e.g., firewalls, proxies) in the data plane. Second, to explore the state space of the network in a scalable manner, we build tractable models of the control and data planes (e.g., by decomposing logically independent tasks) and design domain-specific optimizations (e.g., by narrowing down the scope of search given the intended policies). Taken together, these two ideas enable systematic reasoning about the correctness of stateful data and control planes. We show the utility and performance of these techniques across a range of realistic settings.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Network operators constantly strive to meet critical policy goals to ensure that networks are secure, provide high performance and availability, and meet external compliance requirements. To this end, they have to carefully configure routers and deploy and manage a wide range of network appliances or middleboxes. These middleboxes include WAN optimizers, proxies, intrusion detection and prevention systems, network- and application-level firewalls, and application-specific gateways [60]. For instance, the intended policy may mandate that HTTP traffic goes through a stateful firewall, intrusion detection system, and a web proxy [118, 136]. Policy violations could result in network outage, significant degradation in the network's performance, or serious security lapses.

Today ensuring that the network correctly implements a given policy is a primarily manual and error-prone task. For instance, a recent operator survey found that 35% of networks generate more than 100 problem tickets per month and one-fourth of these take multiple engineer-hours to resolve [55]. Checking even simple reachability (e.g., can A talk to B) policies is already hard. Recent advances such as software-defined networking (SDN) [98, 141] and network functions virtualization (NFV) [83] enable new capabilities to implement more sophisticated context-dependent policies (e.g., packets marked as suspicious may be subject on-demand to deeper inspection) [30,61,63,69,95,160]. These new paradigms also introduce additional sources of complexity (e.g., elastically scaling virtual network functions depending on the load [103, 153]) and also introduce new sources of error (e.g., problems traditionally associated with software implementations) [129].

In the rest of this section, first, we give an overview of the current approach to checking the correctness of networking policies (§1.1). We then discuss the challenges of providing the operators with the proper tools and techniques to check their intended policies as well as our contributions in addressing these challenges (§1.2). We outline the rest of the thesis in §1.3.

1.1 Current approach

As we will discuss in this sub-section, today network operators do not have proper tools and techniques to check the correctness of their intended policies.

1.1.1 Background

Before reviewing the related work on checking network policies, it is useful to take a step back and take a look at what a realistic network is expected to do (see Figure 1-1).

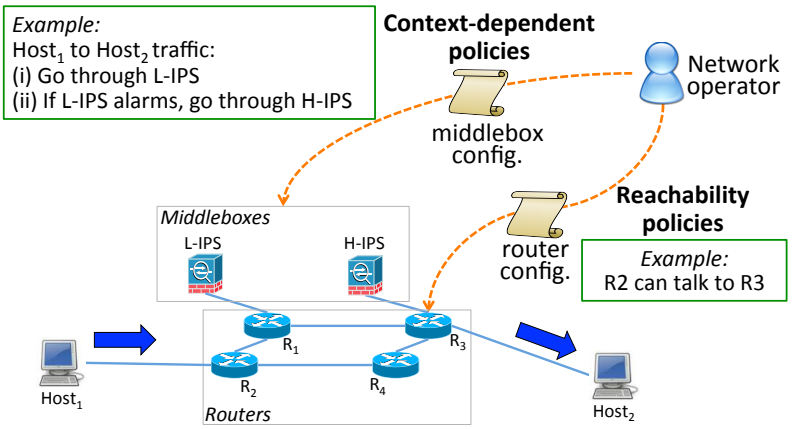


Figure 1-1: Network operators enforce reachability and context-dependent policies. Our goal is to enable them to determine whether they have done so correctly.

Enforcing reachability policies in the control plane: To enforce who can/cannot talk to whom in the network, the operator needs to configure the behavior of the *network control plane*. This is done by configuring individual routers with respect to routing protocols (e.g., BGP, OSPF, RIP). The goal here is to ensure certain end-points can or cannot talk to each other based on the organizational policies (e.g., the departments of a campus network can talk to each other, or traffic from the finance department in an enterprise network cannot reach the R&D department).

Enforcing context-dependent policies in the data plane: Over time networks have been expected to provide more than mere connectivity. Specifically, operators need to ensure their networks meet certain security and performance goals. To this end, operators extensively use specialized network appliances, known as middleboxes [167]. Firewalls, proxies, and intrusion detection/prevention systems (IDS/IPS) are just a few examples of middleboxes. Together with routers' forwarding tables, middleboxes form the *network data plane*, which is where the actual packets are processed.

We refer to policies that mandate a certain traversal of middleboxes as *context-dependent* policies. (These policies are also known as service chaining policies [118,136].) Context here refers to the processing context of traffic on middleboxes that is maintained as a state on the middlebox (e.g., "TCP connection established"). For example, the network operator may need to enforce the following policy: if traffic triggers an alarm on a light-weight IPS, then traffic should be sent to a heavy-weight IPS. In this example, "alarm" on the light IPS is context that is associated with a corresponding state on the light-weight IPS (e.g., the count of suspicious connections seen).

To see what is missing today in terms of reasoning about the behavior of the network, in the following sub-section, we will review representative related work.

1.1.2 Related work

Enforcing reachability and context-dependent policies has received significant attention. This involves efforts on policy enforcement using new networking paradigms such as SDN (e.g., SIMPLE [158], Slick [69], OpenNF [105], E2 [153], PGA [157], Kinetic [30]) and synthesizing correct-by-construction networks (e..g, Frenetic [101], Vericon [70], NetKat [67], Pyretic [144], Propane [73]).

Checking whether the intended policies are enforced correctly in the actual network has recently been an active area of research. The reason for this resurgent interest is the fact that if the intended policies are not enforced correctly, the outcome can be severe for the affected organizations in terms of lost revenue (e.g., service outage [22]), security breaches, and lost reputation (e.g., [46]).

A taxonomy for related work: Here we present a taxonomy to help organize the related

work and put our contributions in perspective. To check the correctness of the network, namely, to see whether the network behavior is compliant with the intended reachability and context-dependent, we naturally need to reason about the control and data planes, respectively. At a very high level, reasoning about any system requires two ingredients: A *model* of the system, and a mechanism to *explore* the model to see whether the behavior is compliant with the intended policy. Therefore, to reason about a network, we need a model of the network's control and data planes as well as mechanisms to explore these models.

Here we briefly discuss the related work on checking network policies using our taxonomy (see a summary in Table 1.1). As we will discuss shortly, prior work critically lacks the ability to model and explore the behavior of realistic networks, which involves stateful behaviors.

	Stateful modeling	State space exploration
Prior work on control plane analysis [97, 99]	No	No
Our work on control plane analysis (§4)	Yes	Yes
Prior work on data plane analysis [124, 125, 137]	No	No
Our work on data plane analysis (§2 and §3)	Yes	Yes

Table 1.1: Taxonomy of prior and our work on reasoning about a network.

Checking the control plane: To classify the prior work on reasoning about the control plane and understand its shortcoming, it is useful to first define what the network control plane is. The control plane maps the routers configuration and route advertisements sent to routers to the forwarding table (FIB or forwarding information base):

$$\textit{ControlPlane} : (\textit{Config} \times \textit{Adv}) \rightarrow \textit{FIB}$$

The FIB, in turn, will determine the behavior of the router with respect to forwarding individual packets. Let ϕ denote a predicate that captures the intended reachability policy (e.g., host A can talk to host B). In other words, if $\phi(ControlPlane)$ is true, the network is correct with respect to the reachability property.

Given this definition of the control plane, we can think of the state of a router's control plane as the set of route advertisements seen so far. Receiving any new route advertisement

can result in a state transition in the control plane.

To make sure a reachability policy is enforced correctly, the operator needs to check whether it is enforced correctly not only in the current FIB, but also in FIBs that may emerge due to interaction of the routers by sending/receiving route advertisements. As we will discuss in §4, many network outages occur due to *latent* routing bugs. These are bugs due to router misconfiguration that might be currently inactive and will be triggered only upon receiving a particular route advertisement. To see whether router configuration files involve any *latent* bugs, we need the ability to systematically explore the state space of the control plane.

There are three classes of prior work on checking the control plane, all of which fall short of finding latent bugs:

1. *Control plane testers*: Batfish [99] takes as input a configuration c and a fixed set of route advertisements adv and then searches for counterexamples to the formula *ControlPlane*$(config, adv)$ to see if there are any violating route advertisements. Because it uses a particular set of route advertisements as input, Batfish cannot identify latent bugs except by exhaustively enumerating different route advertisements, which is intractable.

2. *Configuration bug pattern detectors:* Tools like rcc [97] and ConfigAssure [149] model each bug pattern as a predicate on *configurations* φ : *Config* $\rightarrow$ *Boolean*. rcc then simply computes $\varphi(config)$ for each such bug pattern on a given network configuration $config$. This approach can identify some latent bugs but cannot guarantee that desired invariants will never be violated. Further, it cannot relate an identified bug to actual forwarding errors or identify route advertisements that trigger such errors.

3. *Protocol-specific tools:* There has been recent work on analyzing the control plane with respect to various route advertisements (as opposed to just one particular fixed set of route advertisements). However, it suffers from critical limitations. Some tools focus on a single routing protocol (e.g., BGP for Bagpipe [175]) or a limited set of routing protocol features (e.g., ARC [104]). They can thus not capture the behavior of the control plane that often uses multiple routing protocols and sophisticated features such as route

redistribution and route aggregation that are sources of many real-world bugs [106, 131, 140].

As we will discuss in §4, even one "bad" route advertisement sent to a router can cause a policy violation. This means what we really need is go beyond the prior work and create a fourth category, which we call *control plane checkers*, that expressively models stateful behaviors of the control plane and systematically explores the state space to identifies route advertisements that cause a policy violation (see rows 2 and 3 of Table 1.1).

While not directly related to reasoning about the control plane, there is another body of work that is worth mentioning here. Correct-by-design approaches like metarouting [110] and glue logic [133] provide a framework for defining new control planes such that configurations expressed in the framework meet specific properties (e.g., convergence). These properties hold over all possible route advertisements. Unfortunately, these approaches do not address practically desirable correctness properties (e.g., the absence of blackholes), and potentially unsafe but common protocols (e.g., BGP) and real-world router behaviors (e.g., route aggregation) do not fit within these frameworks.

Checking the data plane: Prior work on checking the data plane can be categorized into

1. *Static verification*, which uses network configuration files to check whether the network behavior complies with the intended policies assuming the data plane behaves correctly (e.g., HSA [124], Veriflow [125], and NOD [137].
2. *Active testing*, on the other hand, checks the behavior of the data plane by injecting test traffic into the network [184].

The key limitation of existing work, in both of the above categories, is that it assumes there are no middleboxes in the data plane. This assumption is widely in contradiction to the widespread use of middleboxes in real networks. For example, a recent survey shows that the number of middleboxes in a network can be on the order of the number of routers in the same network [167]. As we will see in § 2, unlike routers, middleboxes make checking the data plane difficult due to their stateful behaviors (e.g., TCP connection states on a stateful firewall).

How about verifying middleboxes' code? It is also worth mentioning that there has been work on verifying implementation code of middleboxes. Specifically, the work in [89] focuses on finding Click [127] code faults (e.g., crash) as opposed to verifying traffic processing policies (e.g., reachability). NICE combines model checking and symbolic execution to find bugs in control plane software [80]. While useful, this approach has two limitations. First, existing work has made strong assumptions about the size, language, and the structure of the data plane code. Second, even if the data plane code is correct, the data plane may still misbehave due to mis-configuration or "on-the-wire" problems (e.g., a link going down). Therefore, we favor the ability to check the behavior of the data plane with respect to processing actual traffic

Given the status of prior work on checking the data plane, what is critically missing is an approach that captures stateful behaviors of middleboxes and explores the corresponding state space to identify policy violations (see the last two rows of Table 1.1).

Our goal in the context of prior work: A Table 1.1 summarizes, our goal is to bridge the bap between the ability of current approaches to reasoning about the network on the one hand and the stateful natures of realistic networks on the other hand. This requires us to be able to *(i)* model stateful behaviors of the network (in both control and data planes), and *(ii)* systematically explore the state space of the control and data planes.

1.2 Thesis approach and contributions

Next we present our vision, discuss the challenges in realizing it, and outline our contributions.

1.2.1 Vision

The goal of this thesis is to enable network operators to proactively check the correctness of their intended reachability and context-dependent policies. We assume the operator has implemented her intended reachability and context-dependent policies via configuring the routers and middleboxes. Note that we are agnostic to how she has configured the network (i.e., manually or automatically). Our goal is to check whether the intended policies are enforced correctly.

Checking the control plane: Since the reachability behavior of a network is a manifestation of its control plane behavior, we argue that to check the reachability policies of a network, we need to reason about its control plane. This is in contrast to prior techniques that are too narrow in their reasoning (i.e., they reason only about a specific manifestation of the control plane [124,137]). Our goes here is to check whether the intended reachability policies will be satisfied given the router's configuration files through static checking. Our system provides the operator with the result of the verification, which is either success or failure. In the latter case, the system also provides a counter-example (i.e., an environment that results in a policy violation).

Checking the data plane: Context-dependent are realized by the stateful behaviors of middleboxes. While both approaches of static checking and active testing (see §1.1.2) are potentially useful, we focus on active testing for two reasons. First, data plane is where the actual traffic is processed; therefore, active testing of the data plane provides practical assurances as to whether things are working correctly on-the-wire. Second, dynamic actions of middleboxes (e.g., traffic modification, terminating/re-establishing sessions) make is hard to reason about the behavior of the data plane with a purely static approach (we will elaborate on such behaviors in § 3). As it is a typical practice in testing any system, our vision of testing the data plane is realized in the broad framework of model-based testing (MBT) [172]. In particular, we aim to generate test traffic that triggers policy-related states of the model of the data plane. When this traffic is injected into the actual network, we can check the correctness of its behavior via comparing it to the intended context-dependent policies.

1.2.2 Challenges

Realizing our vision is hindered by three key challenges:

- **Scalability**: Reasoning about any system (in our case the network's data and control planes) requires exploring the possible behaviors of the system with respect to the intended properties. To make the exploration more tractable, prior work on checking network policies has made simplifying assumptions to shrink the space of possible behaviors. Specifically, this is done by assuming a fixed set of route advertisements in the

control plane and assuming the lack of middleboxes in the data plane (see §1.1.2). Realizing our vision mandates relaxing these assumptions, which immediately leads to the challenge of scalability in exploring the network's behaviors. This is due to the notion of *state* in that we need to reason about a very large space of stateful network behaviors in both the control plane (e.g., the routing information the routers have obtained so far) and the data plane (e.g., a firewall's state with respect to a TCP session).

- **Diversity**: Real networks are diverse systems on two fronts. First, the control plane behavior is determined by a diverse set of routing protocols (e.g., BGP, RIP, OSPF) as well as interactions across individual protocols (e.g., route redistribution [133]). This is in contrast to the prior work that is protocol-specific [97]. Second, the data plane behavior is determined by various types of middleboxes that are very different in nature (e.g., a firewall vs. an IPS vs. a proxy). Prior work, in contrast, has only considered homogeneous data planes composed of routers/switches [124, 125].
- **Hidden context**: Even after addressing the scalability and diversity challenges, active testing of the data plane requires a binding between test traffic and its processing context (e.g,. cache hit/miss at a proxy, alarm/OK at an IPS) throughout the network. This, however, is challenging in the presence of middleboxes. Not only do not middleboxes make the processing context available, but also they modify opaquely modify traffic that makes it hard to determine the true source of the traffic. For instance, NATs and load balancers dynamically rewrite packet headers.

1.2.3 Contributions

This thesis argues that it is possible to enable network operators to proactively check the correctness of their intended policies by addressing the above challenges. First, to combat the challenge of diversity, we design unifying data abstractions that glue together different routing protocols in the control plane and diverse elements in the data plane. Second, to explore the state space of the network in a scalable manner, we build tractable models of the control and data planes (e.g., by decomposing logically independent tasks) and design domain-specific optimizations (e.g., by narrowing down the scope of search given the intended policies). Taken together, these two ideas enable systematic reasoning about the

correctness of stateful data and control planes. Further, we design an SDN-based network architecture to make the otherwise hidden context available via tagging traffic by middleboxes.

Since a network is composed of data and control planes, we tackle the challenges from §1.2.2 separately with respect to these planes. The specific contributions of this thesis are as follows:

- **Active testing of context-dependent policies in the data plane:** We address key expressiveness and scalability challenges of checking the data plane, we develop a principled testing framework called BUZZ [96]. We model a middlebox in a scalable-yet-expressive manner, we represent is as an ensemble of FSMs rather than a monolithic FSM. Further, to enable composing models of various middleboxes and model the entire data plane, we design a unifying traffic data unit that represents a sequence of packets. To generate test traffic given a model of the data plane, we develop an optimized symbolic execution (SE)-based workflow. We engineer domain-specific optimizations (e.g., reducing the number and scope of symbolic variables) to improve scalability. We develop custom translation mechanisms to convert the output of this step into concrete test traffic (§2);

- **Preserving the test traffic context via middlebox-assisted tagging:** To realize the above active testing framework, we address the challenge of hidden contexts by designing and building FlowTags [92]. FlowTags is an SDN-based network architecture that enables the middleboxes to embed the otherwise hidden context in their outgoing packets, which in turn, will be used by the next switches and middleboxes. The SDN controller orchestrates the network-wide tagging operations of the middleboxes. We show the feasibility of FlowTags in terms of minimal modifications we need to make to existing middleboxes to make them FlowTags-compliant as well as negligible performance overhead such modifications entail (§3);

- **Checking reachability policies via control plane analysis:** To address the key expressiveness and scalability challenges of checking the control plane, we have designed and built a tool, called ERA [94], to analyze the behavior of the control plane.

To build the tool, we have taken three key steps. First, we have designed a unifying and compact bit-vector model for the input/output (I/O) unit of the control plane that captures messages sent by the "environment" of a router (e.g., neighboring routers). Second, we have modeled the processing logic of the control plane as a fast pipeline of boolean operators. Finally, we have designed an algorithm to quickly find reachability between any two given router ports. To make this process fast and scalable, we have employed a range of techniques including the Karnaugh map, using equivalence classes of routes, and taking advantage of Intel AVX2 instruction set for computation of set intersection/union (§4).

We have also made BUZZ and ERA available as open source for the broader use of the research community [1, 2].

1.3 Outline

In §2, we present BUZZ, which is a test traffic generation system that enables the use of model-based testing (MBT) in the specific domain of testing stateful data planes. Note that BUZZ is built based on the assumption that the traffic context (e.g., a connection being suspicious) is preserved in the network, which is not true in the presence of different types of middleboxes that modify traffic (e.g., NATs, proxy). Therefore, taking advantage of SDN as a new opportunity, in §3 we present our architectural solution, called FlowTags, to make hidden context available by encoding it as tags inside packets. Then in §4, we present ERA, a tool for finding latent configuration bugs via control plane analysis. Finally, we present concluding remarks and outline directions for future work in §5.

Chapter 2

Reasoning about stateful data planes using BUZZ

Network operators extensively use specialized network appliances or middleboxes in enterprise networks to enhance the security and performance of their networks. Examples of middleboxes include firewalls, proxies, NATS, intrusion detection/prevention systems, to name a few. Middleboxes are highly popular today so that the number of middleboxes deployed in a typical enterprise network is comparable to the number of traditional routers/switches in the same network [167].

While an integral part of modern network infrastructures, middleboxes[1] have complex stateful behaviors. This means the packet processing behavior of the middleboxes, in addition to the input packet, depends on the current state of the middlebox. For example, a TCP-level firewall may take different actions on a given input packet (e.g., drop or allow) depending on the current state of the connection with which the packet is associated. At a high level, such stateful behaviors make reasoning about middleboxes challenging, as we need to explore a state space.

To be more specific, checking a context-dependent policy in a stateful network involves ensuring whether traffic goes through the intended sequence of NFs; e.g., if an intrusion

[1] In the rest of the thesis, we use the terms middlebox and stateful network function (NF) interchangeably. In general, an NF may be a switch/router or a middlebox (e.g., firewalls, load balancers, intrusion prevention systems, or proxies). It may be realized by a physical appliance or a virtual machine (VM).

detection system (IDS) flags host X for generating too many connections (i.e., if traffic context is "alarm"), then reroute subsequent flows to a deep packet inspection (DPI) filter [63]. Such rich policies and stateful data planes are quite common (e.g., the number of stateful NFs in a network may be comparable to the number of routers [167]).

As we saw in chapter §1, today we do not have a systematic way to check whether a stateful data plane correctly implements intended context-dependent policies. Existing approaches [99, 124, 125, 137, 184] face fundamental *expressiveness* and *scalability* challenges in this regard. First, current abstractions cannot capture stateful behaviors (e.g., how many connections host X has tried to establish) or express context-dependent policies (e.g., on-demand deep inspection). Second, trying to reason about stateful behaviors results in state-space explosion; e.g., a naive application of formal verification tools takes $>$ 20 hours even for a small network with 4-5 nodes (see §2.4).

In this chapter, we present BUZZ [96], a framework for testing context-dependent policies in stateful data planes. BUZZ takes in intended policies from the operator, and by exploring a model of the data plane, it finds abstract test traffic (i.e., an input that triggers policy-relevant states of a model of the data plane). It then translates the abstract test traffic into concrete test traffic and injects it into the actual data plane. Finally, it reports whether the observed behavior complies with the policies. As an active testing framework, BUZZ provides concrete assurances about the behavior "on-the-wire" and can help operators localize sources of violations [184] (§2.3).

In designing BUZZ, we make two key contributions:

- **Expressive-yet-scalable data plane models (§2.3.5):** We introduce a novel abstraction for network traffic called a BUZZ Data Unit (BDU). BDUs extend the notion of *located packets* (a located is a packet along with the router port at which it is currently located) from prior work [124] in three key ways: (1) it enables composition of diverse NFs spanning multiple protocol layers; (2) it simplifies models of NFs operating above L3 by aggregating a sequence of packets; and (3) it explicitly encodes traffic processing history to expose policy-relevant contexts. Second, we model individual NFs as FSMs that process BDUs and explicitly embed the relevant contexts into BDUs. A network then is simply a composition of individual NF models. To build tractable models, we decou-

ple logically independent tasks (e.g., client-side vs. server-side connections) or units of traffic (e.g., distinct TCP connections) within each NF to create an ensemble of FSMs representation rather than a monolithic FSM.

- **Scalable test traffic generation (§2.3.6):** To generate abstract test traffic to explore the behaviors of the data plane model, we develop an optimized symbolic execution (SE)-based workflow. To combat the challenge of state space explosion [77, 79], we engineer domain-specific optimizations (e.g., reducing the number and scope of symbolic variables). We also develop custom translation mechanisms to convert the output of this step into concrete test traffic.

We have implemented BUZZ as an application over `OpenDaylight` [37]. BUZZ provides both text-based and graphical interfaces for operators to input policies and receive test results through an automated workflow. We have written a library of models for several canonical NFs and implemented our SE optimizations using KLEE [78]. We have also developed simple monitoring and test resolution mechanisms (§2.3.6). BUZZ is open-source, and our code, models, and examples can be found at [1].

Our evaluation (§2.4) on a real testbed shows that BUZZ: (1) effectively helps detect both new and known policy violations within tens of seconds; (2) tests hundreds of policies in networks with hundreds of switches and stateful NFs within two minutes; (3) dramatically improves test scalability, providing nearly five orders of magnitude reduction in time for test traffic generation relative to strawman solutions (e.g., model checking).

2.1 Motivation

In this section, we use a few illustrative examples to discuss why it is challenging to check the correctness of context-dependent policies in stateful data planes.

Stateful firewalling: Today most firewalls capture TCP semantics. A common usage is reflexive ACLs [15] as shown in Figure 2-1, where incoming traffic is allowed depending on its *context*. In particular, the context-dependent policy here specifies that only traffic belonging to a TCP connection initiated by a host inside the department (i.e., if traffic context is "solicited") be allowed.

Prior work in network verification models each NF as a "transfer" function

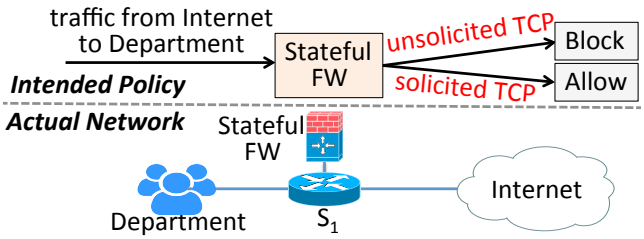


Figure 2-1: Is firewall allowing solicited and blocking unsolicited traffic?

$T(hdr, port)$ whose input/output is a located packet (i.e., a $header, port$ tuple) (e.g., [124, 125, 144]). Unfortunately, even the simple policy of Figure 2-1 cannot be captured by this stateless transfer function. In particular, it does not capture the policy-relevant *state* of the firewall (e.g., `SYN_SENT`) for a given connection.

Context-dependent traffic monitoring: In Figure 2-2, the operator uses a proxy to improve web performance. She also wants to restrict web access; i.e., H_2 (a host in the department) cannot have access to `XYZ.com`. Here the context-dependent policy specifies that both cache hits/misses for H_2 should be monitored. As noted elsewhere [92], there could be subtle policy violations where cached responses evade the monitor because (1) the proxy hides traffic provenance (i.e., true origin), and (2) the proxy's response (i.e., hit vs. miss) depends on the hidden policy-relevant *state* (i.e., the current cache contents).

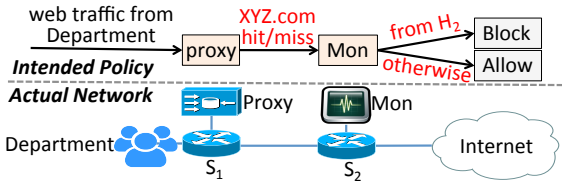


Figure 2-2: Are both cache hit/miss traffic monitored?

While there are mechanisms to fix this (e.g., [92]), operators need tools to check whether such mechanisms are implemented correctly. Again, a stateless transfer function [124, 125, 137] is insufficient, as it does not capture the state of the proxy.

Multi-stage triggers: Figure 2-3 uses a light-weight intrusion prevention system (L-IPS) for all traffic, and only subjects suspicious hosts (i.e., flagged by the L-IPS due to generating too many scans) to the expensive heavy-weight IPS (H-IPS) for payload signature matching. Such context-dependent multi-stage detection can minimize latency and reduce H-IPS load [95].

Again, we cannot check if such multi-stage policies are enforced correctly using ex-

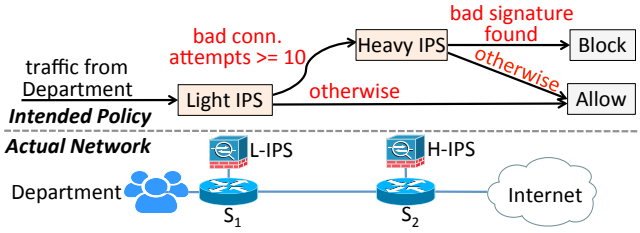


Figure 2-3: Is suspicious traffic sent to heavy IPS?

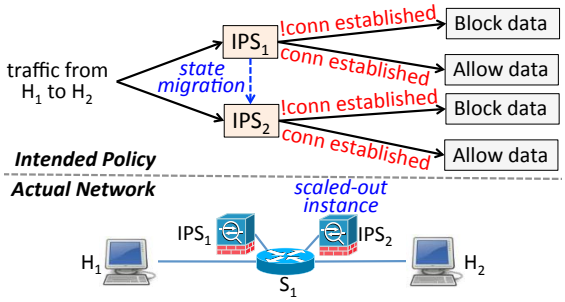


Figure 2-4: Does the scale-out mechanism honor the stateful semantics of migration?

isting mechanisms [99, 124, 125, 184] because they capture neither policy context (e.g., alarm/not alarm) nor data plane state (e.g., the count of bad connection attempts on L-IPS). This example also demonstrates that just capturing packet headers (e.g., [124, 125, 137]) is not sufficient, as the behavior of the H-IPS may depend on packet contents.

Dynamic NF deployments: NFV creates new opportunities for elastic scaling of NFs [83]. However, ensuring the correctness of policies in the presence of elastic scaling is not easy. For example, in Figure 2-4, suppose IPS_1 observes flow f_1 established between the two hosts; later f_1 is migrated to the newly launched IPS_2 for better load balancing [161]. Due to the stateful semantics of the IPS, IPS_2 needs to know that f_1 has already established a TCP connection; otherwise, IPS_2 may incorrectly block this flow. While recent efforts enable state migration [105, 161], we need ways to check whether they do so correctly.

Similarly, in dynamic NF failure recovery [83], if the main NF fails, the backup NF needs to be activated with the correct state so that traffic is uninterrupted (e.g., see [166]). Again, we lack the ability to check whether such mechanisms work as intended.

2.2 Related work

Network verification: There is a rich literature on checking reachability [90, 99, 123, 124, 137, 139, 177, 182]. The work closest to BUZZ is ATPG [184]. As discussed in the motivating example in §2.1, these approaches do not capture the stateful behaviors and context-dependent policies.

Code verification: The work in [89] focuses on finding Click [127] code faults (e.g., crash) as opposed to verifying traffic processing policies (e.g., reachability). NICE combines model checking and SE to find bugs in control plane software [80]. BUZZ is complementary to these efforts.

Modeling stateful networks: Joseph and Stoica formalized middlebox forwarding behaviors but do not model stateful behaviors [120]. The only work that also models stateful behaviors are FlowTest [93], Symnet [170], and the work by Panda et al [154]. FlowTest's [93] high-level models are not composable and the AI planning approaches do not scale beyond 4-5 node networks. Symnet [170] uses models written in Haskell to capture NAT semantics similar to our example; based on published work we do not have details on their models, verification procedures, or scalability. The work by Panda et al. is different from BUZZ in terms of both goals (only reachability policies) and techniques (static checking) [154].

Policy enforcement: There are several frameworks to facilitate policy enforcement [30, 92, 105, 153, 157, 158]. There are also efforts to generate correct-by-construction SDN programs [67, 70, 101]. Our work is complementary, as it checks whether the intended behavior manifests correctly in the actual data plane.

Simulation and shadow configurations: Simulation [34], emulation [18,32], and shadow configurations [65] are common methods to model/test networks. BUZZ is orthogonal in that it focuses on generating test traffic. While our current focus is on active testing, BUZZ applies to these platforms as well. We also posit that our techniques can be used to validate these efforts.

2.3 System design

2.3.1 Overview

Our goal is to enable network operators to check at human-interactive timescales whether their context-dependent policies are realized in stateful data planes. Next, we present a high-level view of BUZZ to meet this goal and summarize key challenges in realizing it.

To put our work in perspective, we note that there are two complementary approaches: *(1) Static verification* uses network configuration files to check whether the network behavior complies with the intended policies assuming the data plane behaves correctly (e.g., HSA [124], Veriflow [125], NOD [137], Batfish [99]); *(2) Active testing*, on the other hand, checks the behavior of the data plane by injecting test traffic into the network [184]. While both are useful, we adopt an active testing approach for two reasons. First, it provides practical assurances that things are actually working correctly "on-the-wire". Second, network behaviors in certain scenarios such as dynamic NF deployment (Figure 2-4) are hard to capture with a purely static approach.

Due to context-dependent policies and complex stateful behaviors, naive attempts to generate test traffic, either manually or via fuzzing [107, 142], are ineffective. For example, in Figure 2-3, in order to trigger the policy context "L-IPS alarm" and check if traffic will actually go to H-IPS, we need to carefully craft a sequence of packets that drive the count of bad connections on L-IPS to ≥ 10; achieving this via randomly generated packets is unlikely. Our goal is to automate this process.

To bridge the gap between policies and the actual data plane, we adopt model-based testing (MBT) [172], which is useful when the blackbox behavior of a system needs to be actively tested. The high-level idea is to (1) use a *model* (or *specification*) of the system under test and a *search* mechanism to systematically find *test inputs* that trigger certain behaviors of the model, and then (2) compare the behavior of the system under test to the behavior of the model for each input [172].

Figure 2-5 shows the high-level workflow of BUZZ:

1. *Model Instantiation*: BUZZ instantiates a model of the data plane using the intended

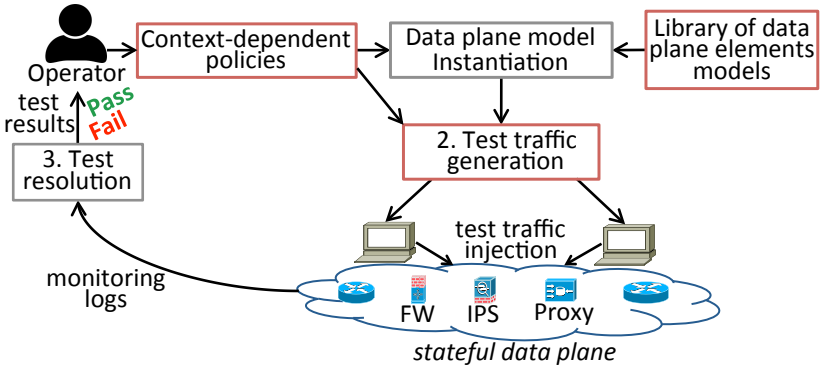


Figure 2-5: High-level workflow of BUZZ.

policies (the only input by the operator) and a library of NF models;

2. *Test Traffic Generation*: BUZZ generates abstract test traffic to trigger policy-relevant behaviors of the data plane model. BUZZ then translates it into concrete test traffic, which is then injected into the actual data plane;

3. *Test Resolution*: BUZZ monitors the actual data plane and compares the observed behavior to the intended policies. The result (i.e., success/violation) is reported to the operator.

There are two challenges in realizing this workflow:

- *Expressive-yet-scalable data plane models:* To see why this is challenging, let us consider some seemingly natural candidates. A natural starting point would be the transfer function abstraction [124, 144]; however, it is not expressive, as it offers no stateful semantics and no binding to the relevant context. On the other hand, using an NF's implementation code as its model is not tractable (e.g., Squid [49] has $\geq$ 200K lines of code) and may suffer from other practical limitations (e.g., code may not be available, or implementation bugs may affect test traffic).
- *Scalable test traffic generation:* Exploring data plane's behaviors is challenging even for simple reachability policies in stateless data planes [184]. Our setting is worse, as reasoning about stateful behaviors requires addressing the challenge of state-space explosion. Off-the-shelf mechanisms (e.g., model checking) struggle beyond a few hundred lines of code (see §2.3.6 and §2.4).

We address these two challenges in §2.3.5 and §2.3.6, respectively. Before doing so, in the next section (§2.3.2), we first formalize our problem to shed light on the key require-

Listing 2.1: An abstract stateful NF.

```
1 //Input: packet inPkt on port inPort
2 ⟨outPkt, state⟩ ← process(inPkt, state)
3 context ← stateToContextMap(state)
4 outPort ← applyPolicy(outPkt, context)
5 dispatch(outPkt, outPort)
```

ments of modeling the data plane and generating test traffic.

2.3.2 Problem formulation

In this section, we formalize our model-based testing framework to see what a data plane model should capture and what test traffic needs to do. These inform our approach to modeling (§2.3.5) and test traffic generation (§2.3.6).

2.3.3 Intuition behind model and test traffic

What should the data plane model capture: First, we give the intuition behind what an NF model needs to capture. As we saw in §2.1, data planes are stateful (e.g., the bad connection attempts count in Figure 2-3). However, being stateful is not sufficient for a data plane model to be expressive. Specifically, to test context-dependent policies, the model needs to explicitly map each state to a context. For example, if we want to trigger an alarm on L-IPS in Figure 2-3 (e.g., to check if the traffic will actually go to H-IPS), we need to capture the mapping from the bad connection attempts count (e.g., ≥ 10 or < 10) to the context (e.g., alarm or not alarm).

To understand what an NF model should capture, we consider the abstract NF shown in Listing 2.1 that shows the NF model as running three logical steps: (1) It processes an input packet and updates some relevant state (e.g., an IPS updating `bad_conn_attempts_count`) (Line 2); (2) It extracts the relevant *context* for the processed packet (e.g., alarm on an IPS based on `bad_conn_attempts_count`) (Line 3); (3) It applies the corresponding policy (e.g., drop, forward) via function $applyPolicy(.)$ and then dispatches the packet to the policy-mandated port (Lines 4-5).

What should test traffic do? At a high level, test traffic for a given policy needs to drive the data plane to a state corresponding to the context. In Listing 2.1, this means we need to find a sequence of packets that drives the NF to a state (Line 2) that maps to the intended

context (Line 3). If the NF is policy-compliant, the traffic at this point will be sent to a policy-mandated port (Lines 4-5). For example, to exercise the context of "L-IPS alarm" in Figure 2-3, test traffic needs to make `bad_conn_attempts_count` to exceed 10; then, we check whether traffic at this point actually goes to H-IPS.

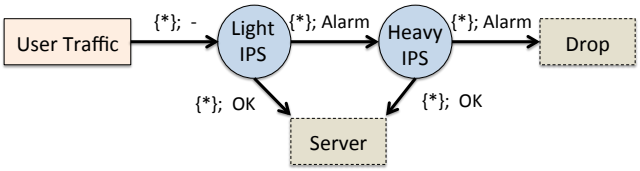
2.3.4 Formal framework

Having seen the intuition behind state, context, and test traffic, we formalize these to inform our design.

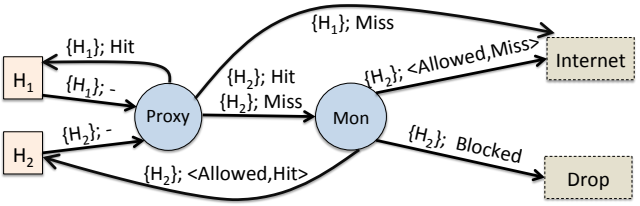
Context-dependent policies: Before formally defining context-dependent policies, it is useful to intuitively see what they are used for. Most prior work on middlebox policy focuses on a *static policy graph* that maps a given *traffic class* (e.g., as defined by network locations and flow header fields) to a *chain* of middleboxes [103,121,158]. For instance, the administrator may specify that all outgoing web traffic from location A to location B must go through a firewall, an IDS, and a proxy in order. However, this static abstraction [103, 121, 158] fails to explicitly capture the origin and processing context of traffic. Thus, we proposes the *context-dependent policy graph* (or $CDPG$) abstraction to specify intended policies.

A CDPG is a directed graph with two types of nodes: (1) In and Out nodes, and (2) *logical middlebox* nodes. In and Out nodes represent network ingresses and egresses (including "drop" nodes). Each logical middlebox represents a type of middlebox function, such as "firewall." (For clarity, we restrict our discussion to "atomic" middlebox functions; i.e., a multi-function box will be represented using multiple nodes.) Each logical middlebox node is given a configuration that governs its processing behavior for each traffic class (e.g., firewall rulesets or IDS signatures). As discussed earlier, administrators specify middlebox configurations in terms of the unmodified traffic entering the CDPG without worrying about intermediate transformations.

Each edge in the CDPG is annotated with the *condition* $m \rightarrow m'$ under which a packet needs to be steered from node m to node m'. This condition is defined in terms of (1) the traffic class, and (2) the *processing context* of node m, if applicable. Figure 2-6 shows two DPG snippets:



(a) Dynamic policy routing



(b) Middlebox context

Figure 2-6: CDPGs for the examples in Figures 3-3 and 3-4. Rectangles with solid lines denote "Ingress" nodes and with dotted lines denote "Egress" nodes. Circles denote logical middlebox functions. Each edge is annotated with a $\{Class\}; Context$ denoting the traffic class and the processing context(s). All traffic is initialized with a null/"-" context.

- Let us revisit the example in Figure 2-3. In Figure 2-6a, we want all traffic to be first processed by the light IPS. If the light IPS flags a packet as suspicious, then this should be sent to the heavy IPS. In the case, the edge connecting the light IPS to the heavy IPS is labeled as "*, *Alarm*"; * denotes the class (any traffic in this case) and *Alarm* provides the relevant processing history from the light IPS.
- Next, we revisit the example in Figure 2-2. For the two hosts H_1 and H_2 in Figure 2-6b, we want to use the monitoring device to apply an ACL on host H_2's web requests. For correct policy enforcement, the ACL must be applied to both cached and uncached responses. Thus, both "H_2, *Hit*" and "H_2, *Miss*" need to be on the Proxy-to-ACL edge. (For ease of visualization, we do not show the policies applied to the responses coming from the Internet.)

Next we formally define a context-dependent policy. Let $context^{pkt}_{NF_i}$ denote the processing context corresponding to packet pkt at NF_i (Line 3 of Listing 2.1). Then, the *context sequence* of the packet is the sequence of contexts along the NFs it has traversed; i.e., if pkt has traversed NF_1, ..., NF_i, its context sequence is $ContextSeq^{pkt} = \langle context^{pkt}_{NF_1}, \ldots, context^{pkt}_{NF_i} \rangle$.

Context-dependent policies are expressed as a set of rules of the form:

$$Policy : TrafficSpec \times ContextSeq \mapsto PortSeq$$

Here, $TrafficSpec$ is a predicate on the IP 5-tuple (e.g., source IP and transport protocol), $ContextSeq$ is a context sequence, and $PortSeq$ is a sequence of network ports $Ports$ (interfaces).[2] For example, in Figure 2-3, the policy that mandates "if traffic triggers an alarm on L-IPS, it must be sent to H-IPS" is specified as:

$$\langle \texttt{srcIP=Dept} \rangle, \langle alarm_{L-IPS} \rangle \mapsto$$
$$\langle L{-}IPS \rightarrow S_1, S_1 \rightarrow S_2, S_2 \rightarrow H{-}IPS \rangle$$

(Policies for dynamic NF deployments, such as Figure 2-4, are defined slightly differently—see §2.3.6.)

Stateful data planes: Contexts are convenient "shorthands" to define policies. In reality, however, the data plane operates in terms of the related but (possibly) lower-level notion of state.

As we saw in Listing 2.1, a stateful NF takes an input packet on one of its ports, processes it, goes to a new state, and outputs a packet on one of its ports. A stateful NF can be naturally expressed as a finite-state machine (FSM) of the form $NF_i = (S_i, I_i, Ports_i, T_i)$, where S_i is the set of NF_i states, I_i is the initial state of NF_i, $Ports_i$ is the set of ports of NF_i (where $Ports_i \in Ports$), and $T_i : Pkts \times Ports_i \times S_i \mapsto Pkts \times Ports_i \times S_i$ is the stateful (as opposed to stateless, e.g., [124]) transfer function of NF_i. We model intended packet drops as sending packets to a virtual "drop port" on the NF. To model the entire data plane, the topology function $\tau : Ports \mapsto Ports$ captures the physical interconnection of NFs. Finally, we define the state of the data plane, S_{DP}, as the conjunction of the states of its individual NFs.

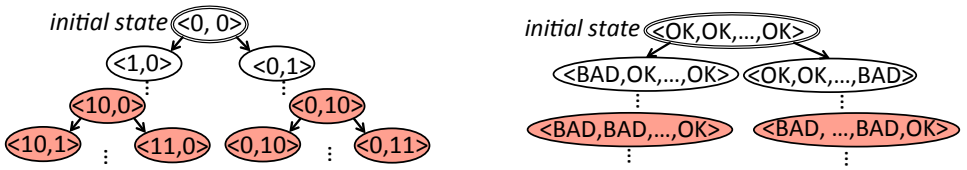
[2]Without loss of generality, we assume policies are in terms of physical NF instances as opposed to logical types of NFs. This is more precise because the semantics of stateful NFs (e.g., NATs) requires that both directions of a flow pass the same NF instance.

There are many levels of abstraction to write such an FSM on, from low-level code variables to high-level logical states (e.g., proxy cache state). Irrespective of this granularity, to be expressive for testing the model needs to provide a mapping from the states to the corresponding traffic specification and context:

$$stateToContextMap_i : 2^{S_i} \mapsto TrafficSpec \times C_i$$

where C_i denotes the set of all contexts of NF_i.

To illustrate this, let us revisit Figure 2-3. Figure 2-7 shows two possible ways of modeling L-IPS as an FSM. In both Figures 2-7a and 2-7b, each of the red states maps to $\langle$`srcIP=Dept`$\rangle, \langle alarm_{L-IPS} \rangle$—these mappings make the models expressive. (In §2.3.5, we will discuss other requirements of an FSM-based NF model in addition to expressiveness.)



(a) Each state is of the form $\langle badAttmpCnt_{H_1}, badAttmpCnt_{H_2} \rangle$

(b) Each state is of the form $\langle connStatus_{f_1}, \ldots, connStatus_{f_{20}} \rangle$

Figure 2-7: Two example FSM models of L-IPS of Figure 2-3 assuming a world with 2 hosts and 20 flows. The states corresponding to alarm (i.e., at least 10 bad connection attempts) are highlighted in red.

Test traffic: Test traffic needs to trigger the policy context by driving the data plane to a state that corresponds the context (e.g., a red state in Figure 2-7). Thus, $trace = \langle pkt_1, \ldots, pkt_m, \ldots, pkt_r \rangle$ is a test trace for $intent : trafficSpec \times contextSeq \mapsto portSeq$ iff:

1. Each packet $pkt \in trace$ satisfies $trafficSpec$, and
2. S_{DP} does not correspond to $contextSeq$ after injection of each of packets $\langle pkt_1, \ldots, pkt_{m-1} \rangle$, and
3. S_{DP} corresponds to $contextSeq$ after injection and processing each of packets $\langle pkt_m, \ldots, pkt_r \rangle$.

After $trace$ is injected into the actual data plane, test resolution involves checking whether packets $\langle pkt_m, \ldots, pkt_r \rangle$ actually traverse ports $portSeq$.

Takeaways: This framework suggests two key design implications: (1) While an FSM is a natural starting point to model a stateful NF, an expressive model should bridge the gap between its states and policy-mandated traffic specification and context (§2.3.5); and (2) Test traffic should satisfy the traffic specification and drive the data plane to a state that corresponds to the policy context (§2.3.6).

2.3.5 Data plane model instantiation

In this section, we discuss how to instantiate a model of the data plane. Recall from §2.3 that this stage takes as input a library of NF models and the policy. The challenge in building such a library is to model each type of NF (e.g., stateful firewall, web proxy) such that these models are (1) *composable*, despite diverse types of NFs operating at different network layers; (2) *expressive*, despite stateful behaviors and hidden context; and (3) *scalable* to explore. After presenting our high-level approach, we introduce a new abstract data unit for modeling input-output of NFs and describe how we create scalable NF models via an ensemble-of-FSMs representation. Finally, we describe how we construct the network-wide model composing individual models of NFs.

High-level idea: A natural starting point to model an NF that is composable is the *transfer function* from prior work [124, 144]. Each NF is modeled as: $lp \leftarrow T(lp)$. Here, the input/output is a located packet $lp = (pkt, port)$, an IP packet (header) along with its location in the network. However, as we saw in §2.1, this is not expressive on several fronts w.r.t. state and context. To see how we can make it expressive, let us revisit our abstract NF from Listing 2.1 and contrast it with the transfer function. This highlights two key missing elements: (1) there is no notion of state, and (2) the located packet has no binding to the relevant context.

Our formalism from §2.3.2 suggests two extensions: (1) Instead of the (stateless) transfer function, we need to move to an FSM-like abstraction that captures state and the state-to-context mappings; and (2) We need some way to logically bind a packet to its relevant context. To this end, we extend the located packet abstraction so that it carries the relevant

Listing 2.2: BDU is the I/O unit of an NF model.

```
struct BDU{
 // IP fields
 int srcIP, dstIP, proto;
 // transport
 int srcPort, dstPort;
 // TCP specific
 int tcpSYN, tcpACK, tcpFIN, tcpRST;
 // HTTP specific
 int httpGetObj, httpRespObj;
 // BUZZ-specific
 int dropped, networkPort, BDUid;
 // Each NF updates traffic context
 int c-Tag[C_TAG_MAX]; //context tags
 int p-Tag; //provenance tag
 ...};
```

context history as it traverses the data plane model. Then, we can consider an NF as an FSM that processes this extended located packet and explicitly includes the policy-relevant context in the outgoing packet. In a nutshell, this summarizes our basic insight to create an expressive model.

Next, we discuss how we translate this insight into a concrete realization. We also address the scalability requirement of NF models, as a naive FSM model will have too many states to explore.

The BUZZ Data Unit (BDU): We start by presenting our approach to modeling the extended located packet idea described above and explain how it enables composability, expressiveness, and scalability. Concretely, a BDU is a *struct* as shown in Listing 2.2 that extends a located packet [124, 144] in three key ways:

1. *Multi-layer abstraction with IP as the common denominator:* Unlike a located packet, a BDU can explicitly encode higher-layer semantics (e.g., HTTP GET or responses). The key to achieving model composability while enabling higher-layer semantics is simple. Borrowing from the design of IP, we pick the network layer as the narrow waist across diverse NFs. Each NF model processes only relevant fields of an input BDU (e.g., an L2 switch ignores HTTP fields).

2. *Tag fields for context and provenance:* First, to ensure a BDU carries its context as it goes through the network, we introduce the context tag, or `c-Tag`, field, which explicitly binds the BDU to its context (e.g., 1 bit for cache hit/miss, 1 bit for alarm/no-alarm).

When the NF model receives an input BDU, it generates an output BDU with the updated `c-Tag` (e.g., a proxy may set the cache hit bit). Second, a BDU preserves its provenance via its `p-Tag` field. This field encodes the BDU's original 5-tuple indicating its $TrafficSpec$. This binding is needed because certain NFs (e.g., NATs, proxies) rewrite the original IP 5-tuple of a BDU. We ensure the provenance field `p-Tag` is left unchanged by NF models the BDU traverses.

3. *Aggregation for scalability:* Each BDU can represent a sequence of packets associated with higher-layer NF operations. This aggregation helps shrink the search space for finding test traffic (§2.3.6). For example, all packets of an HTTP reply are captured by a single BDU with the `httpRespObj` field indicating the retrieved object id; a proxy's state (e.g., cache contents) gets updated after receiving this BDU.

To design a BDU struct in practice, we need to identify the protocols that affect any context mentioned in the policies. The struct's fields are simply the union of the policy-related headers of these protocols. For example, if our policy involves a stateful firewall, then TCP `SYN` and `ACK` should be part of the fields, as these are the fields that denote connection establishment semantics. Since each NF model processes only relevant fields of an incoming BDU, our BDU abstraction is future-proof. For example, if we later need to add an ICMP field to the BDU of Listing 2.2, existing NF models will remain unchanged, as they simply ignore this new field.

Ensemble of FSMs representation: While there are many ways to expressively model a stateful NF, not all models may be scalable. To see why, consider modeling the state-space as the concatenation of state variables we have identified (e.g., in a proxy this concatenation may have three variables: per-host and per-server connection states and per-object cache state). Taking this approach means with var variables each with val possible values, such a monolithic FSM has val^{var} states (i.e., an exponential growth with the number of values). While it may be tempting to reduce the state space by moving to a layer-specific abstraction (e.g., a proxy model that ignores TCP and purely works at the HTTP layer), this is not viable, as the models of diverse NFs will not be composable.

To build a scalable FSM without compromising composability, we borrow insights from

the design of actual NFs. NF programs in practice are not monolithic; rather, they independently track "active" connections, and different functional components of an NF are segmented; e.g., client- vs. server-side handling in a proxy are separate. This naturally suggests two opportunities:

1. *Decoupling independent traffic units:* Consider a stateful firewall. If modeled as a monolithic FSM, each state of the model involves states of individual connections. While this is expressive, it is not scalable as the number of connections grow. By decoupling per-connection states, we model the NF as an ensemble of FSMs. In general, this insight cuts the number of states from $|state|^{|conn|}$ to $|conn| \times |state|$, where $|conn|$ and $|state|$ denote the number of connections and states per connection, respectively.

2. *Decoupling independent tasks:* To illustrate this, consider a proxy. The code of a real proxy (e.g., `Squid` [49]) typically has three logical modules in charge of managing client-side and server-side connections and the cache. We decouple such logically independent tasks in the model so that instead of a monolithic FSM model with each state being of the "cross-product" form $\langle client\_TCP\_state, server\_TCP\_state, cache\_content \rangle$, we use an ***ensemble*** of three smaller FSMs, i.e., $\langle client\_TCP\_state \rangle$, $\langle server\_TCP\_state \rangle$, and $\langle cache\_content \rangle$. In general, if an NF has $|T|$ independent tasks with task i having S_i states, this idea cuts the number of states from $\prod_{i=1}^{|T|} |S_i|$ to $\sum_{i=1}^{|T|} |S_i|$.

Putting it together: Taken together, our BDU abstraction as the traffic I/O unit and FSM ensembles as NF models satisfy the three modeling requirements of composability, expressiveness, and scalability. As an illustration, Listing 2.3 shows a code snippet of a proxy model focusing on the actions when a client requests a non-cached HTTP object and while the proxy has not established a TCP connection with the server. Each NF instance is identified by a unique `id` that allows us to index into the relevant variables. Since the traffic I/O of the model (Line 1) is a BDU, the model is composable with other NF models. Second, instead of a monolithic FSM, it is partitioned into these three dimensions (i.e., client-, server-side connections and cache) making the model scalable. The state variables of different proxy instances are naturally partitioned per NF instance (not shown) and

Listing 2.3: Proxy as an ensemble of FSMs.

```
1  BDU Proxy(NFId id, BDU inBDU){
2     ...
3     if ((frmClnt(inBDU)) && (isHttpRq(inBDU))){
4       if (!cached(id, inBDU)){
5            if(srvConnEstablished(id, inBDU))
6              outBDU=rqstFrmSrv(id, outBDU);
7            else
8              outBDU=tcpSYNtoSrv(id, inBDU); }}
9     //set c-Tags based on context (e.g., hit/miss)
10    outBDU.c-Tags = ...
11    ...
12    return outBDU;}
```

help track the relevant NF states, and are updated by the NF-specific functions such as `srvConnEstablished`. The choice of passing *id*s and modeling state in per-*id* global variables is not fundamental but an artifact of using C/KLEE. If the input `inBDU` is an HTTP request (Line 3) and the requested object is not cached (Line 4), the proxy checks the status of the server TCP connection. If it has already been established (Line 5), the output BDU is an HTTP request (Line 6). Otherwise, the proxy initiates a TCP connection with the server (Line 8). Finally, note that the proxy updates `c-Tags` of the output BDU before sending it out.

Composing the data plane model: Next we discuss how to instantiate a model of the data plane given the models of individual NFs. Listing 2.4 illustrates this for the network of Figure 2-2. BUZZ uses the policy to automatically concretize the relevant model parameters (e.g., lines 3–4 specify which content/host to watch). Lines 8–10 model the stateless switch, where we model a switch as a static data store lookup [124]. Note that a BDU captures its current location in the network via its `networkPort` field, which gets updated as it traverses the network. Function `lookUp()` takes an input BDU, looks up its forwarding table, and creates a new `outBDU` with its port value set based on the forwarding table.

Similar to prior work [124, 184], our network model processes one-packet-per-NF at a time, without modeling (a) batching or queuing inside the network, (b) parallel processing inside NFs, or (c) simultaneous processing of different packets across NFs. As a result, the data plane model is a simple loop (Line 26); in each iteration, a BDU is processed (Line 27) in two steps: (1) it is forwarded to the other end of the current link (Line 28), (2) it is then passed as an argument to the NF connected at this end (e.g., a switch or firewall)

Listing 2.4: Data plane pseudocode for Figure 2-2.

```
1  // Symbolic BDUs to be instantiated (see §2.3.6).
2  BDU A[20];
3  int objToWatch = XYZ.com;
4  int hostToWatch = H2;
5  // Global state variables
6  bool Cache[2][100]; // 2 proxies, 100 objects
7  // Model of a switch
8  BDU Switch(NFId id, BDU inBDU){
9     outBDU=lookUp(id, inBDU);
10    return outBDU;}
11 // Model of a monitoring NF
12 BDU Mon(NFId id, BDU inBDU){
13     ...
14     outBDU = inBDU;
15     if (isHttp(id, inBDU)){
16       takeMonAction(id, inBDU);/* if inBDU
17       contains objToWatch destined to
18       hostToWatch, set outBDU.dropped to 1.*/}
19     ...
20    return outBDU;}
21 // Model of a proxy NF; See Listing 2.3
22 BDU Proxy(NFId id, BDU inBDU){...}
23 main(){
24   // Model of the data plane
25   initializeProvenanceTags(A[]);
26   for each injected A[i]
27    while (!DONE(A[i])){
28     Forward A[i] on current link;{
29     A[i] = Next_NF(A[i]);{
30     assert(
31     (!(A[i].p-Tag==hostId[H2]))
32     ||(!(A[i].c-Tags[cacheContext]==objToWatch));
33 }}}}
```

(Line 29). The output BDU is then processed in the next iteration. The loop is executed until the BDU is "DONE"; i.e., it either reaches its destination or is dropped by an NF. (NFs may be time-triggered (e.g., TCP time-out), so we capture time using a BDU field. These "time BDUs" are injected by the network model periodically to update time-related states.) Based on the policy, wee identify the `Next_NF` in line 29. (As an optimization, our implementation pre-populates switches' `lookup()` and `Next_NF()` based on shortest-path routing between policy-relevant NFs.) The role of the `assert` statement will become clear in §2.3.6, where we discuss test traffic generation.

2.3.6 Test traffic generation

In this section, we discuss how to generate test traffic given the policies and the data plane model. First, we highlight why we choose symbolic execution (SE) as a starting mechanism to explore the data plane model. Then we present our domain-specific optimizations to

scale SE to generate abstract test traffic consisting of BDUs. Then, we show how to convert this abstract test traffic into concrete test traffic. Finally, we present an extension to test dynamic NF scenarios.

Why symbolic execution (SE)?: For BUZZ to be usable by operators at human interactive timescales, it should generate test traffic within seconds to a few minutes even for large networks. This is challenging on two fronts:

- **Traffic space explosion:** Unlike prior work where an IP packet header is an independent unit of test (hence mandating a search only over the header space [123, 125, 184, 185]), we need to search over a very large *traffic space* of all possible sequences of traffic units. While BDUs, as compared to IP packets, improve scalability via aggregation (§2.3.5), we still have to search over the space of possible BDU value assignments.
- **State space explosion:** Even though using the FSM ensembles abstraction significantly reduces the number of states, it does not address *state space explosion* due to composition of NFs; e.g., if the models of NF_1 and NF_2 can reach K_1 and K_2 states, respectively, their composition will have $K_1 \times K_2$ states.

Unfortunately, several canonical search solutions (e.g., model checking [13, 85] and AI planning tools [20]) do not scale beyond 5-10 stateful NFs; e.g., model checking took 25 hours for policy involving only two contexts.

As the first measure to address the search scalability challenge, we choose symbolic execution (SE), which is a well-known approach to tackle state-space explosion [77]. At a high level, an SE engine explores possible behaviors of a program (in our case, the data plane model) by assigning different values to its *symbolic variables* [79]. In our implementation, we use KLEE [78], a popular SE engine.

Generating abstract test traffic: BUZZ employs SE as follows. For each $intent : trafficSpec \times contextSeq \mapsto portSeq$, we constrain the symbolic BDUs to satisfy the $TrafficSpec$. Then, to drive the SE engine to generate test traffic that satisfies $contextSeq = \langle context_{NF_1}, \ldots, context_{NF_N} \rangle$, we introduce the logical negation of $contextSeq$ as an *assertion* in the network model code. In practice, if $contextSeq$ involves contexts of N NFs $context_1, \ldots, context_N$, BUZZ instruments the network model with

Listing 2.5: Assertion pseudocode for Figure 2-3 to trigger alarms at both IPSes.

```
1  // Global state variables
2  int L_IPS_Alarm[noOfHosts];//alarm per host
3  int H_IPS_Alarm[noOfHosts];//alarm per host
4  ...
5  //A[] is an array of symbolic BDUs
6  ...
7     assert((!(A[i].c-Tags[L_IPS_Alarm]==1)) ||
8           (!(A[i].c-Tags[H_IPS_Alarm]==1)));
```

an assertion of the form $\neg(context_1 \wedge \cdots \wedge context_N)$, where each term is expressed in terms of BDUs' `c-Tag` sub-fields. The assertion guides the SE engine toward finding a "violation" of the assertion by assigning concrete values to symbolic BDUs.[3] In effect, SE generates *abstract test traffic* by concretizing a sequence of symbolic BDUs. The abstract test traffic will be then translated into concrete test traffic, which in turn, will be injected into the actual data plane. The injected concrete test traffic must traverse the sequence of ports specified in $portSeq$; otherwise, the actual data plane violates $intent$.

To illustrate this, let us revisit Listing 2.4, where we want a test trace to check cached responses from the proxy to host H_2. Lines 30-32 show the assertion to get a sequence of i BDUs that change the state of the data plane such that the ith BDU in the abstract traffic trace: (1) is from host H_2 (Line 31), and (2) corresponds to a cached response (Line 32). For example, the SE engine may generate 6 BDUs: three BDUs between a host other than H_2 in the $Dept$ and the proxy to establish a TCP connection (the 3-way handshake) where the third BDU has `httpGetObj = httpObjId` (this effectively makes the proxy cache the object), followed by another 3 BDUs, this time from H_2 with the field `httpGetObj` set to `httpObjId` to induce a cached response. Similarly, Listing 2.5 shows an assertion in Lines 7-8 to trigger alarms at both L-IPS and H-IPS of the example from Figure 2-3.

While SE is significantly faster than other candidates, it is not sufficient for interactive use. Even after a broad sweep of configuration parameters to customize KLEE, it took several hours for a small network (§2.4). To scale to large topologies, we implement two optimizations:

[3]Note that an assertion of the form $\neg(A_1 \wedge \cdots \wedge A_n)$, or equivalently $(\neg A_1 \vee \cdots \vee \neg A_n)$, is violated only if each term A_i is evaluated to `true`.

- **Minimizing number of symbolic variables:** Making an entire BDU structure (Listing 2.2) symbolic forces KLEE to find values for every field. Instead, BUZZ identifies the policy-related subset of BDU fields and only makes these symbolic and concretizes the rest. For instance; when BUZZ is testing a data plane with a stateful firewall but no proxies, it makes the HTTP-relevant fields concrete (i.e., non-symbolic) by assigning a don't care value $*$ (represented by -1 in our implementation) to them.
- **Scoping values of symbolic variables:** The $trafficSpec$ scopes the range of values a BDU may take. BUZZ further narrows this range using the policy and protocols semantics. For example, even though the `tcpSYN` field is an integer, BUZZ constrain it to be either 0 or 1.

Test coverage: Ideally, test traffic should cover the space of all possible traffic, including (1) packet traces of all possible lengths (in terms of number of packets in the trace), and (2) enumerating all possible values of the fields of each packet. However, this is impractical with respect to both test traffic generation and injection overheads. That is why even in case of simple reachability policies and stateless data planes in prior work [184], only one sample packet out of an equivalence class of packets (i.e., the set of all packets that experience the same forwarding behavior) is selected as the test packet. Similarly, we define our test coverage goal as obtaining one test trace to exercise each policy. In §2.4, we will show that BUZZ (1) successfully satisfies this goal, and (2) can be used to satisfy alternative coverage goals.

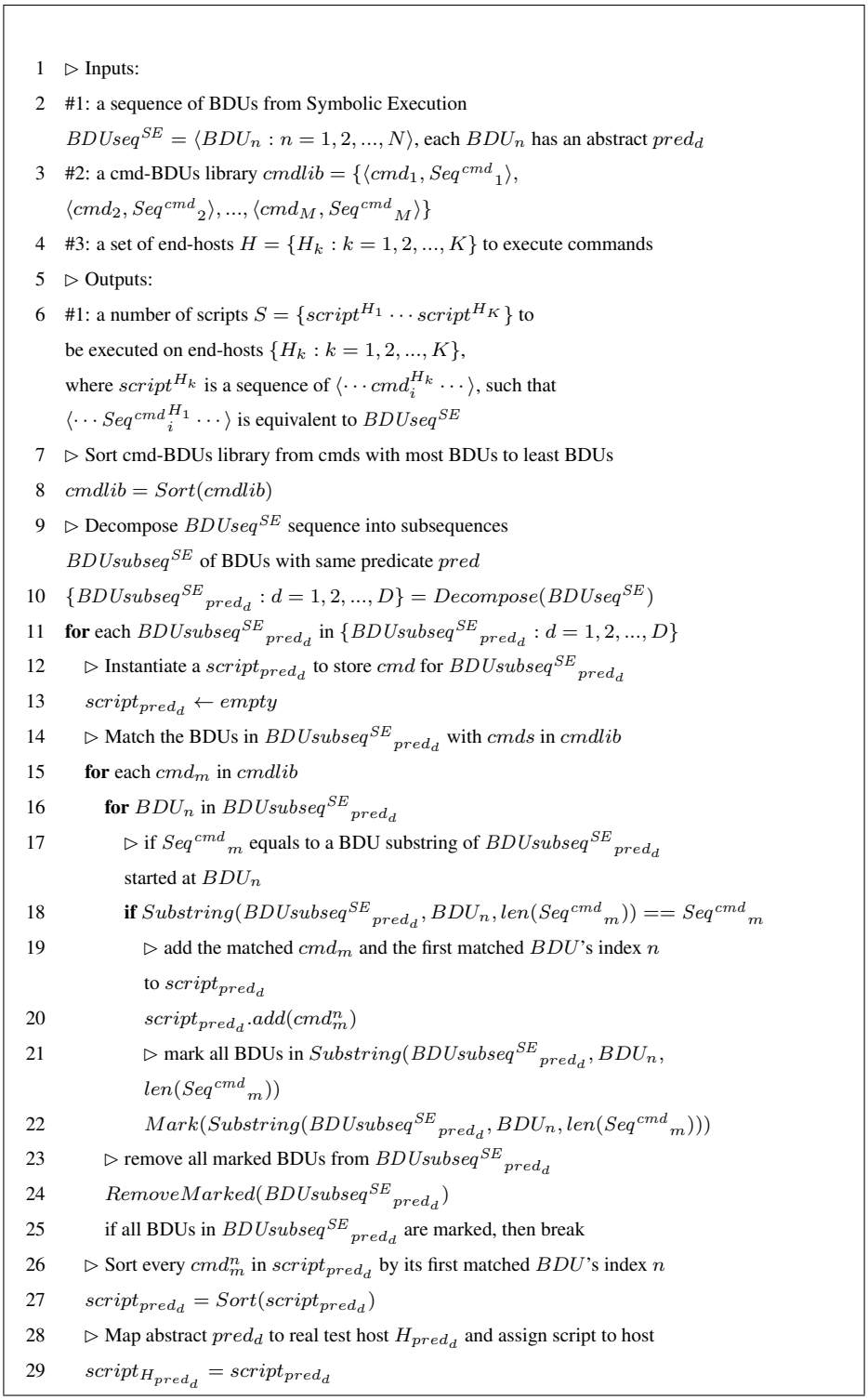
Generating concrete test traffic: The output of the SE step is a sequence of BDUs $BDUSeq^{SE}$. Since BDUs are abstract, we cannot directly inject them into the actual data plane. Moreover, we cannot simply do a one-to-one translation between BDUs and raw packets and do a trace replay [10,184] because we need to honor session semantics (e.g., for TCP or FTP) of the policies—several parameters of such sessions (e.g., TCP seq. numbers) are outside of our control and are chosen by the OS of the end hosts at run time.

To this end, we translate abstract test traffic into *test traffic injection scripts* that are run on end hosts to inject concrete test traffic. The translation algorithm uses a library of traffic injection commands that maps a known $BDUSeq_l$ into a script. For example, if a BDUSeq consists of 3 BDUs for TCP connection establishment and a web request, we map

this into a `wget` with the required parameters (e.g., server IP and object URL). In the most basic case, the script will be an IP packet. Using our domain knowledge, we populated this library with commands (e.g., `getHTTP(.)`, `sendIPPacket(.)`) that support IP, TCP, UDP, FTP, and HTTP.

For completeness, its pseudocode is presented in Figure 2-8. Here we give the intuition behind our translation algorithm. We partition the $BDUSeq^{SE}$ based on srcIP-dstIP pairs (i.e., communication end-points) of BDUs; i.e., $BDUSeq^{SE} = \bigcup_l BDUSeq_l$. Then for each partition $BDUSeq_l$, we do a longest-specific match (i.e., match on a protocol at the highest possible layer of the network stack) in our test script library, retrieve the corresponding command for each subsequence, and then concatenate these commands to form a traffic injection script.

Figure 2-8 shows the pseudocode for the translation mechanism.



▷ Inputs:
#1: a sequence of BDUs from Symbolic Execution
$BDUseq^{SE} = \langle BDU_n : n = 1, 2, ..., N \rangle$, each BDU_n has an abstract $pred_d$
#2: a cmd-BDUs library $cmdlib = \{\langle cmd_1, {Seq^{cmd}}_1 \rangle,$
$\langle cmd_2, {Seq^{cmd}}_2 \rangle, ..., \langle cmd_M, {Seq^{cmd}}_M \rangle\}$
#3: a set of end-hosts $H = \{H_k : k = 1, 2, ..., K\}$ to execute commands
▷ Outputs:
#1: a number of scripts $S = \{script^{H_1} \cdots script^{H_K}\}$ to
be executed on end-hosts $\{H_k : k = 1, 2, ..., K\}$,
where $script^{H_k}$ is a sequence of $\langle \cdots cmd_i^{H_k} \cdots \rangle$, such that
$\langle \cdots {Seq^{cmd}}_i^{H_1} \cdots \rangle$ is equivalent to $BDUseq^{SE}$
▷ Sort cmd-BDUs library from cmds with most BDUs to least BDUs
$cmdlib = Sort(cmdlib)$
▷ Decompose $BDUseq^{SE}$ sequence into subsequences
$BDUsubseq^{SE}$ of BDUs with same predicate $pred$
$\{{BDUsubseq^{SE}}_{pred_d} : d = 1, 2, ..., D\} = Decompose(BDUseq^{SE})$
for each ${BDUsubseq^{SE}}_{pred_d}$ in $\{{BDUsubseq^{SE}}_{pred_d} : d = 1, 2, ..., D\}$
  ▷ Instantiate a $script_{pred_d}$ to store cmd for ${BDUsubseq^{SE}}_{pred_d}$
  $script_{pred_d} \leftarrow empty$
  ▷ Match the BDUs in ${BDUsubseq^{SE}}_{pred_d}$ with $cmds$ in $cmdlib$
  for each cmd_m in $cmdlib$
    for BDU_n in ${BDUsubseq^{SE}}_{pred_d}$
      ▷ if ${Seq^{cmd}}_m$ equals to a BDU substring of ${BDUsubseq^{SE}}_{pred_d}$
      started at BDU_n
      if $Substring({BDUsubseq^{SE}}_{pred_d}, BDU_n, len({Seq^{cmd}}_m)) == {Seq^{cmd}}_m$
        ▷ add the matched cmd_m and the first matched BDU's index n
        to $script_{pred_d}$
        $script_{pred_d}.add(cmd_m^n)$
        ▷ mark all BDUs in $Substring({BDUsubseq^{SE}}_{pred_d}, BDU_n,$
        $len({Seq^{cmd}}_m))$
        $Mark(Substring({BDUsubseq^{SE}}_{pred_d}, BDU_n, len({Seq^{cmd}}_m)))$
    ▷ remove all marked BDUs from ${BDUsubseq^{SE}}_{pred_d}$
    $RemoveMarked({BDUsubseq^{SE}}_{pred_d})$
    if all BDUs in ${BDUsubseq^{SE}}_{pred_d}$ are marked, then break
  ▷ Sort every cmd_m^n in $script_{pred_d}$ by its first matched BDU's index n
  $script_{pred_d} = Sort(script_{pred_d})$
  ▷ Map abstract $pred_d$ to real test host H_{pred_d} and assign script to host
  $script_{H_{pred_d}} = script_{pred_d}$

Figure 2-8: Translating abstract test traffic into test traffic injection scripts.

Testing dynamic NF deployments: Next we describe the extensions needed to handle dynamic NF deployment scenarios. Intuitively, the goal in these scenarios is to ensure

the change is transparent with respect to stateful semantics of traffic. To be concrete, let $Policy_{before}$ and $Policy_{after}$ denote the policies that the operator intends to enforce before and after the "change" occurs, where the change is captured by $changeCond$ (e.g., an NF's scale-out, or failure). We define the correct enforcement of a dynamic NF deployment policy as follows: For each data plane state $s \in S_{DP}$, if $changeCond$ is triggered while the data plane is in s, then $Policy_{after}$ is enforced correctly.

In Figure 2-4, $Policy_{before}$ is the top part of the policy graph (i.e., involving IPS_1), $Policy_{after}$ is the bottom part of the policy graph (i.e., involving IPS_2), and $changeCond$ is IPS_1's scale-out. Irrespective of the state in which IPS_1 scales out, IPS_2 must start processing traffic with the same state at which IPS_1 has scaled out.

Abstract test traffic generation for dynamic NF deployment scenarios is slightly different from what we described in §2.3.6. At a high-level, for every data plane state $s \in S_{DP}$, BUZZ (1) generates test traffic to drive the data plane to s, (2) triggers $changeCond$ (e.g., by scaling-out an NF), and (3) test if the data plane is compliant with $Policy_{after}$. For completeness, we describe the full procedure in Figure 2-9.

```
▷ Inputs:
#1: Policy_1:pred_1(5−tuple)×C_1↦Ports_1 before migrate/rollback
#2: Policy_2:pred_2(5−tuple)×C_2↦Ports_2 after migrate/rollback
▷ Outputs:
#1: a sequence of BDUseq^SE = ⟨BDU_n : n = 1, 2, ..., N⟩ with two substrings,
BDUseq^SE_before and BDUseq^SE_after, which should satisfy:
BDUseq^SE_before exploits all possible context context in C_1 before
migration/rollback happens.
BDUseq^SE_after test all possible context in C_2 after the migration/rollback.
▷ Init BDU sequence
BDUseq^SE = ⟨BDU_n : n = 1, 2, ..., N⟩
▷ note the values in BDU_n for calculation by Symbolic Execution
makesymbolic(BDU_n)
▷ exploits all possible context in C_1
▷ BDUs processed sequentially by Policy_1
for each BDU_i in BDUseq^SE
  if BDU_i is in BDUseq^SE_before
    ▷ process BDU_i by Policy_1 and update C_1
    C_1 = Policy_1(pred_1(BDU_i), C_1, Ports_1)
▷ do migrate/rollback and change service chain from Policy_1 to Policy_2
▷ map ports
Ports_2 = g(Ports_1)
▷ migrate/rollback context
C_2 = C_1
▷ test all possible context in C_2
▷ BDUs processed sequentially by Policy_2
for each BDU_j in BDUseq^SE
  if BDU_j is in BDUseq^SE_after
    ▷ process BDU_i by Policy_2 and update C_2
    C_2 = Policy_2(pred_2(BDU_i), C_2, Ports_2)
▷ generate BDU sequence with values assigned by Symbolic Execution
symbolicoutput = ⟨BDU_n : n = 1, 2, ..., N⟩
```

Figure 2-9: Pseudocode for abstract test traffic generation for change management policies.

2.3.7 Implementation

BUZZ comprises ≈ 10,000 lines of code, including NF models, code for test traffic generation, test resolution, extensions to KLEE, and the operator interfaces. The entire workflow of BUZZ is implemented atop `OpenDayLight` [37]. The source code is available at [1].

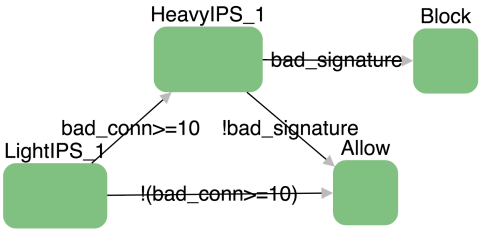


Figure 2-11: Graphical interface to input policies (e.g., multistage-triggers policy in Figure 2-3).

Operator interface: Operators can enter policies using either a text-based or a graphical interface (example screenshots are shown in Figures 2-10 and 2-11). BUZZ then performs a set of sanity checks on the policies and warns the operator of any mistakes (e.g., an overlap between $TrafficSpec$ of two policies). This I/O is the only effort that BUZZ needs from the operator. Once policies are entered, the workflow of BUZZ (Figure 2-5) is entirely automated.

Choose File In_L_H_IPS.js

```
#Traffic
10.1.0.1 10.2.0.1
#Enforcement
LightIPS_1 bad_conn>=Threshold HeavyIPS_1
LightIPS_1 !(bad_conn>=Threshold) Allow
HeavyIPS_1 bad_signature Block
HeavyIPS_1 !bad_signature Allow
#Customize
LightIPS_1:Threshold=10
```

Figure 2-10: Text-based interface to input policies (e.g., multistage-triggers policy in Figure 2-3).

NF models: We have written C models for switches, ACL devices, stateful firewalls, NATs, L4 load balancers, HTTP and FTP proxies, passive monitoring, and simple intrusion prevention systems (e.g., counting failed connection attempts and matching payload signatures). Our models are between 10 (for a switch) to 100 lines (for a proxy cache) of C code. We reuse common templates across NFs; e.g., TCP connection sequence used in both the firewall and proxy models. Note that modeling NFs is a one-time offline task and can be augmented with community efforts [33]. We validated models by inspecting call graphs visualization [29, 56] on extensive manually generated input traffic to ensure the models are correct.

Test traffic generation and injection: We use KLEE with the optimizations discussed in §2.3.6 to generate BDU-level test traffic (i.e., abstract test traffic), and then translate it to test scripts that run at the injection points.

Test traffic monitoring and test resolution: We use offline monitoring via `tcpdump` (with suitable filters). BUZZ uses the monitoring logs to determine the test result. For completeness, we have provided the monitoring and test resolution pseudocode in Figure 2-12. Here we give the intuition behind this process. From the input policy, BUZZ inspects the monitoring logs to check whether traffic has traversed the policy-mandated ports. If so, the test concludes with success. Otherwise, a policy violation along with the first violating port on which traffic appeared is declared.

```
1  ▷ Inputs:
2  #1: packet traces pkttrace_{port_i} dumped at each port_i in Ports
3  #2: policy Policy:pred(5−tuple)×C↦Ports, where C includes all possible
   contexts
4  ▷ Outputs:
5  #1: The resolution result of each context context_i in C in terms of pass/fail
6  #2: The port of the NF that causes the failure

7  ▷ perform resolution scheme for each context context_i in C
8  for each context_i in C
9     ▷ Trace = ⟨pkt_m, ..., pkt_r⟩ is the test packets for this context
10    for testpkt in ⟨pkt_m, ..., pkt_r⟩
11       ▷ calculate the logically correct ports testpkt should reach
12       Ports^{logical}_{testpkt} = Policy(pred(testpkt)), context_i)
13       ▷ find the real ports testpkt has reached
14       Ports^{reality}_{testpkt} = search testpkt in each pkttrace_{port_i}
15       if Ports^{reality}_{testpkt} == Ports^{logical}_{testpkt}
16          context_i test pass
17       else
18          context_i test fail
19          ▷ Compare the port of Ports^{reality}_{testpkt} and Ports^{logical}_{testpkt} and find the first
            different port, which is the NF that causes the failure.
20          FailedNFPort = FirstDiffPort(Ports^{reality}_{testpkt}, Ports^{logical}_{testpkt})
21       mark context_i as tested
```

Figure 2-12: Pseudocode for BUZZ test resolution.

2.4 Evaluation

In this section, we show that:

1. BUZZ can help detect a broad spectrum of both new and known policy violations

(§2.4.1);

2. BUZZ works in close-to-interactive time scales (i.e., within two minutes) even for large topologies with 100s of switches and stateful NFs (§2.4.2); and

3. BUZZ's design is critical for its scalability (§2.4.3).

Testbed and topologies: We use a testbed of 13 server-grade machines (20-core 2.8GHz servers with 128GB RAM) connected via direct 1GbE links and a 10GbE Pica8 OpenFlow switch. On each server, with KVM installed, we run injectors and software NFs as separate VMs, connected via `Open vSwitch`. The specific stateful NFs are iptables [26] as a NAT and a stateful firewall, Squid [49] as a proxy, Snort [48] and Bro [156] as IPS/IDS, Balance [8], and PRADS [41].

In addition to the example scenarios from §2.1, we use 8 randomly selected recent topologies from the Internet Topology Zoo [53] with 6–196 nodes. We also use two larger topologies (400 and 600 nodes) by extending these topologies. These serve as switch-level topologies; we extend them with different NFs to enforce policies. For the scalability experiments, we augment each switch-level topology with stateful NFs (§2.4.2) by connecting each stateful NF to a randomly selected switch. As a concrete policy enforcement scheme, we used prior work to handle dynamic middleboxes [92]. (We reiterate designing this scheme is not the goal of BUZZ; we simply needed *some* concrete solution.)

2.4.1 BUZZ end-to-end use cases

First, we demonstrate the effectiveness of BUZZ in finding both new and known policy violations.

Finding new violations: Using BUZZ, we uncovered several policy violations in recent systems, a few of which we present here:

- **Violations due to reactive control in Kinetic [30]**: We set up a simple policy composed of an IDS followed by a Kinetic dynamic firewall. By generating malicious traffic, BUZZ found that the first few malicious packets are wrongly let through. The root cause of this violation is the delay between (1) the IDS's detection of malicious traffic and sending an "infected" event to the controller, and (2) the controller's reconfiguration of the data

plane to block malicious traffic.

- **Incorrect state migration using OpenNF [105]**: We used the OpenNF-enhanced PRADS [41, 105] to enforce the following policy: if a host spawns more than $Thresh$ TCP connections, its traffic should be sent to a rate limiter. BUZZ revealed a violation due to the incorrect state migration when we elastically scale a PRADS instance. Specifically, BUZZ made a host establish n_1 and n_2 sessions with a server before and after migration, respectively, such that: n_1, $n_2 < Thresh$, but $n_1 + n_2 > Thresh$. BUZZ then found that traffic did not go to the rate limiter. This is because OpenNF does not migrate the session count (i.e., n_1) from $PRADS_1$ to $PRADS_2$.
- **Faulty policy composition using PGA [157]**: We used PGA[4] to compose two policies on traffic from H_1 to H_2: it should pass a load balancer and a stateful firewall ($policy_1$), and if it is found suspicious, it then should go to an IPS ($policy_2$). After enforcing the composition of the two policies, BUZZ found that the test traffic exercising $policy_1$ did not go through the firewall. This is because the SDN switch rules corresponding to $policy_1$ took precedence over the switch rules for $policy_2$, rendering $policy_2$ ineffective.
- **Incorrect tagging using FlowTags [92]:** BUZZ helped us identify a bug in our FlowTags implementation in OpenDaylight [37]. In the scenario of Figure 2-2 from §2.1, the controller code in charge of decoding tags (e.g., to distinguish hosts behind the proxy) would assign the same tag value to traffic from different hosts. Our test traffic showed that the proxy's cache hit replies bypass the monitoring device. BUZZ's traffic trace indicated that the tag values of cache miss/hit are identical; this gave us a hint as to focus on the controller code in charge of configuring the tagging behavior of the proxy.

Finding known violations: We used a "red team–blue team" exercise, to evaluate the utility of BUZZ in finding known policy violations. In each scenario, the red team (Student 1) secretly picks one of the policies (at random) from the set of policies that is known to both teams, and creates a failure that causes the network to violate this intent; e.g., misconfiguring L-IPS count threshold. The blue team (Student 2) uses BUZZ to identify a violation and localize the source of the policy violation.

[4]We used our implementation of PGA, as its code was unavailable.

"Red Team" scenario	**BUZZ test trace**	**Violating NF**
Cascaded NATs using Click IPRewriter [127] ; NAT_2 incorrectly rewrites srcIP triggering "assertion failure" on NAT_1 [88]	H_1 attempts to access to the server	NAT_2
Multi-stage triggers (Fig. 2-3); L-IPS miscounts by summing three hosts	H_1 makes 9 scan attempts followed by 9 scans by H_2	L-IPS
Conn. limit.; Login counter resets	H_1 makes 3 continuous log in attempts with a wrong password	Login counter
Conn. limit.; S_1 missing switch forwarding rules from AuthServer to the protected server	H_2 makes a log in attempt with the correct password	S_1
Conflicting firewall rules: Rule 1, if internal connect to external IP, allow IP to access any internal port; Rule 2, block external access to internal port 443	A TCP connection from internal C_1 to external S_1 followed by an access from S_1 to $C_1 : port443$	Firewall
Asymmetric routing; Client-to-server TCP traffic goes through Bro, but the response bypasses Bro. Since Bro does not see the SYN_ACK packet, it (mistakenly) blocks the connection.	a TCP connection followed by TCP data packets	switch close to destination

Table 2.1: Example red-blue team scenarios.

Table 2.1 highlights the results for a subset of these scenarios and the specific traces that BUZZ generated. Three of the scenarios use the motivating examples from §2.1. In the Conn. limit. scenario, two hosts are connected to a server through an authentication server to prevent brute-force password guessing attacks. The authentication server is expected to halt a host's access after 3 consecutive failed log in attempts. Finally, in the asymmetric routing scenario, upstream and downstream traffic traverse different paths [130]. In all scenarios, the blue-team successfully localized the failure (i.e., which NF or link is the root cause) within 10 seconds.

It is useful at this time to reiterate that these types of violations could not be exposed

by existing debugging tools such as ATPG [184], ping, or traceroute, as they do not capture violations w.r.t. stateful/context-dependent aspects. We also tried using fuzzing to generate test traffic, using both Scapy [47] and a custom fuzzer. Across all scenarios, fuzzing did not find any test trace within 48 hours. This is because we need targeted search to trigger specific data plane states, which fuzzing is not suited for.

2.4.2 Scalability

Recall that we envision operators using BUZZ in an interactive fashion; i.e., the time for test generation should be within 1-2 minutes even for large networks with hundreds of switches and stateful NFs.

We evaluate how BUZZ scales with topology size and policy complexity. We define policy complexity as the number of stateful NFs whose contexts appear in the policy. We consider a baseline policy that has 3 stateful NFs (a NAT, followed by a proxy, followed by a stateful firewall). The firewall is expected to block access from a fixed subset of origin hosts to certain web content. To create more complex policies, we linearly "chain" together repetitions of the baseline policy.

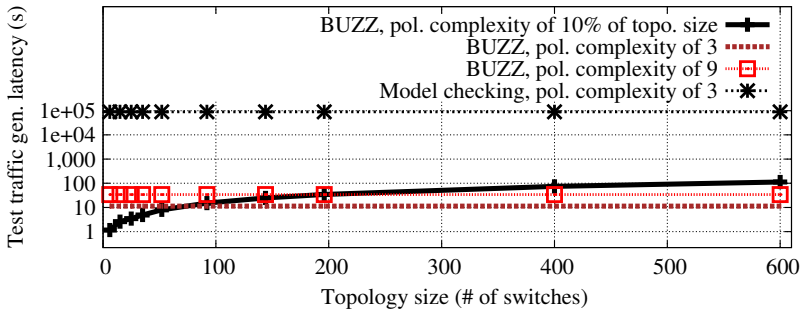


Figure 2-13: Test generation latency of BUZZ.

Figure 2-13 shows the average test traffic generation latency for various topology sizes and policy complexities. There are two takeaways. First, BUZZ generates test traffic in human-interactive time scales; even in the largest topology with 600 switches and the most complex policy it takes only 113 seconds. Second, BUZZ's test traffic generation latency only depends on the policy complexity: if we increase the topology size without increase the policy complexity, this will not add to the test traffic generation latency. This is expected, as test traffic generation involves a search over the data plane state space, which

naturally is a function of stateful NFs.

To put the traffic generation latency of BUZZ in perspective, Figure 2-13 also shows the traffic generation latency of a strawman solution of using the model checker CMBC [13]. Even on a small 9-node topology (6 switches and 3 stateful NFs), it took 25 hours; i.e., on a 90$\times$ larger topology, BUZZ is *at least* five orders of faster.

Test coverage: We have evaluated the test coverage of BUZZ, and here, we discuss the three takeaways. First, across all scenarios we have discussed in this section, we explicitly enumerated all contexts, and observed that BUZZ provided full coverage with respect to the coverage goal of generating one test case to trigger each context. Second, we extended BUZZ to satisfy an alternative coverage goal of generating > 1 test trace per context. We enabled this through an iterative test generation process, where in each iteration, we obtain a new test case by using assertions such that a previously generated test case will not be generated again. Finally, while, in general, using multiple test cases per context may reveal new violations, in our experiments, we did not find new violations by doing so.

2.4.3 Effect of BUZZ design choices

Next, we do a component-wise analysis to demonstrate the effect of our key design choices and optimizations.

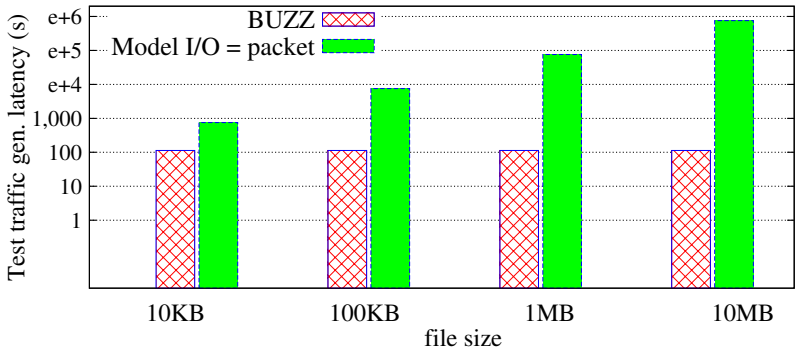


Figure 2-14: BDUs vs. packets for various request sizes.

BDUs vs. packets: To see how aggregating a sequence of packets as a BDU helps with scalability, we use BUZZ to generate test traffic to test the proxy-monitor policy (Figure 2-2), first in terms of BDUs and then in terms of raw MTU-sized packets, on varying sizes of files to retrieve from the web. Figure 2-14 shows that on the topology with 600 switches

and 300 stateful NFs, in case of packet-level test traffic generation, test traffic generation latency increases linearly with the file size. On the other hand, since the number of test packets is dominated by the number of object retrieval packets, aggregating all file retrieval packets as one BDU significantly cuts the latency. (The results, not shown, are consistent across topologies as well as using FTP instead of HTTP.)

(2) scoping the values yields a further $> 9\times$ reduction.

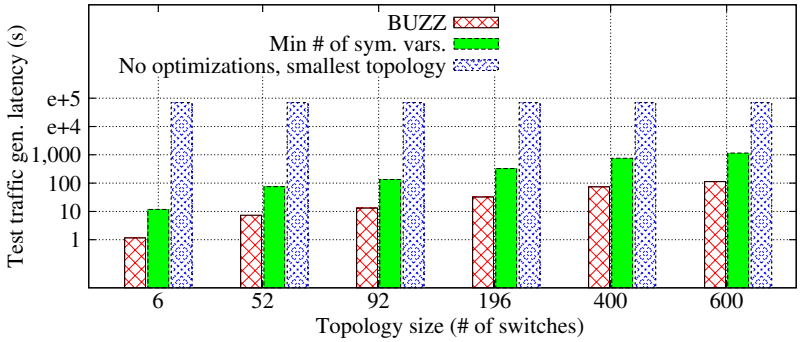


Figure 2-15: Improvements due to SE optimizations.

Impact of SE optimizations: We examine the effect of the SE-specific optimizations (§2.3.6) in Figure 2-15. To put these numbers in context, using KLEE without the optimizations on a network of six switches and a policy chain with three stateful NFs takes $\geq$ 19 hours. We see that (1) minimizing the number of symbolic variables cuts the test generation latency by three orders of magnitude, and

2.5 Summary

BUZZ tackles a key missing piece in network verification—checking *context-dependent policies* in *stateful data planes* introduces fundamental expressiveness and scalability challenges. We make two key contributions to address these challenges: *(1)* Developing expressive and scalable network models; and *(2)* An optimized application of symbolic execution to tackle state-space explosion. We demonstrate that BUZZ is scalable and it can help diagnose policy violations.

We see the following as natural directions for future work:

Model synthesis: BUZZ uses hand-generated models of NFs. A natural direction for future work is to use program analysis to automatically synthesize NF models from mid-

dlebox code (e.g., [84]) or logs (e.g., [74]).

Soundness vs. completeness: We found BUZZ to be empirically sound (i.e., every bug it found was a real bug) For "infinite-state" systems, it is not possible to simultaneously achieve both guarantees [116]. BUZZ's design favors soundness (i.e., if we report a violation, then the data plane actually has that behavior) over completeness (i.e., if we do not find a violation, then there are no bugs). In our setting, this is a worthwhile trade-off as we can repeat tests for greater coverage [116, 184].

New use cases: Looking forward, we believe BUZZ can be extended to systematically check interoperability of new protocols with middleboxes [114]. As preliminary evidence, we were able to replicate a known problem with a middlebox-cooperative TCP extension called HICCUPS [86], where the protocol fails in the presence of middleboxes that modify certain headers or if there are multiple middleboxes on the path.

Chapter 3

Exposing hidden traffic context using FlowTags

As we discussed in the beginning of §2, many network management goals (e.g., high performance and security) are implemented using custom middleboxes, such as firewalls, NATs, proxies, intrusion detection and prevention systems, and application-level gateways [165, 167]. In this section we present our solution to the challenge of "hidden processing context" (see 1.2.2), which is introduced by middleboxes and hinders reasoning about the network behavior.

As we saw in §2, a necessary part of a test traffic generating system is to capture traffic processing context (§2.3.5). Unfortunately, processing context is typically hidden in realistic deployments due to the actions of middleboxes (e.g., firewalls, NATS, and proxies) that modify traffic. In this chapter, we present FlowTags [92], which is our solution to the challenge of hidden middlebox context. Middleboxes such as NATs, proxies, and load balancers, hide the true context of traffic as part of their inherent operations. This is a stumbling block to testing context-dependent policies because in order to test and diagnose network data planes, we need the ability to determine the true origin and processing context of test traffic at any point in the network.

In particular, to be able to reason about a stateful data plane as we presented in the preview section, we need two types of information to be associated with each packet at any location in the network:

1. ORIGINBINDING: There should be an explicit binding between a packet and its "origin" (i.e., the network entity that has originally created the packet);
2. CONTEXTBINDING: There should be an explicit binding between a packet and its processing context with respect to the middleboxes the packet has gone through (e.g., "alarm" at an IPS, "connection established" at a firewall);

For instance, NATs and load balancers dynamically rewrite packet headers, thus violating ORIGINBINDING. Similarly, dynamic middlebox actions, such as responses served from a proxy's cache, may violate CONTEXTBINDING. (We elaborate on these examples in §3.1.)

To address the problem of hidden context, at a high level, we observe the central visibility into the traffic processing behavior of the network in Software-Defined Networking (SDN) as an opportunity. Specifically, we have designed FlowTags, an SDN-based network architecture, to systematically preserve traffic provenance in the presence of middlebox actions. We take a pragmatic stance that we should attempt to integrate middleboxes into SDN as "cleanly" as possible. Thus, our focus here is to systematically revive the ORIGINBINDING and CONTEXTBINDING, even in the presence of dynamic middlebox actions. We identify *flow tracking* as the key to policy enforcement. (We use the term "flow" in a general sense, not necessarily to refer to IP 5-tuple.) That is, we need to reliably associate additional contextual information with a traffic flow as it traverses the network, even if packet headers and contents are modified. For instance, this helps determine the packet's true endpoints rather than rewritten versions (e.g., as with load balancers), or provide hints about the packet's provenance (e.g., a cached response).

Based on this insight, we extend the SDN paradigm with the *FlowTags* architecture. Because middleboxes are in the best (and possibly the only) position to provide the relevant contextual information, FlowTags envisions simple extensions to middleboxes to add *tags*, carried in packet headers. SDN switches use the tags as part of their flow matching logic for their forwarding operations. Downstream middleboxes use the tags as part of their packet processing workflows. We retain existing SDN switch interfaces and explicitly decouple middleboxes and switches, allowing the respective vendors to be able to innovate

independently.

Deploying FlowTags thus has two prerequisites: (P1) adequate header bits with SDN switch support to match on tags and (P2) extensions to middlebox software. We argue that (P1) is possible in IPv4; quite straightforward in IPv6; and will become easier with recent OpenFlow standards that allow flexible matching [38] and new switch hardware roadmaps [75]. As we show in §3.3.6, while identifying where in the middlebox codebase to modify to add support for FlowTags is hard, (P2) requires minor code changes to middlebox software.

Contributions and roadmap: Our specific contributions in this chapter are:

- We describe controller–middlebox interfaces to configure tagging capabilities (§3.3.3) and design new controller policy abstractions and rule generation mechanisms to systematically configure the tagging logic (§3.3.4).
- We show that it is possible to extend five software middleboxes to support FlowTags, each requiring less than 75 lines of custom code in addition to a common 250-line library. (To put these numbers in context, the middleboxes we have modified have between 2K to over 300K lines of code.) (§3.3.6).
- We show that FlowTags adds little overhead over SDN mechanisms and that the controller is scalable (§3.4).
- We present a use case of FlowTags in enabling scalable and elastic DDoS defense via in-data plane tagging (§3.5).

3.1 Motivation

As we saw in the previous chapter, testing context-dependent policies in a stateful data plane requires visibility into traffic processing context. To see why this is a challenge in practice, we present a few examples that highlight how middlebox actions violate making it difficult to enforce and reason about network-wide policies. We also discuss why some seemingly natural strawman solutions fail to address our requirements.

Attribution problems: Figure 3-1 shows two middleboxes: a NAT that translates private IPs to public IPs and a firewall configured to block hosts H_1 and H_3 from accessing specific

public IPs. Ideally, we want administrators to configure firewall policies in terms of the original source IPs. Unfortunately, we do not know the private-public IP mappings that the NAT chooses dynamically; i.e., the ORIGINBINDING tenet is violated. If only traffic from H_1 and H_3 should be directed to the firewall and the rest are allowed to pass through, an SDN controller cannot install the correct forwarding rules at switches S_1/S_2 because the NAT change the packet headers; i.e., CONTEXTBINDING no longer holds.

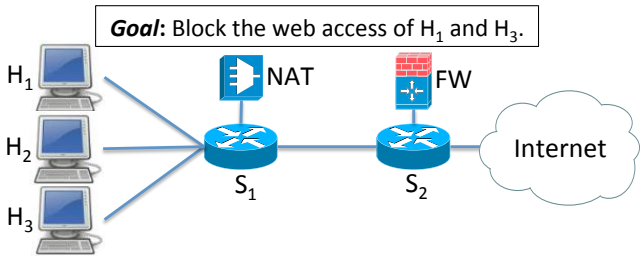


Figure 3-1: Applying the blocking policy is challenging, as the NAT hides the true packet sources.

Network diagnosis: In Figure 3-2, suppose the users of hosts H_1 and H_3 complain about high network latency. In order to debug and resolve this problem (e.g., determine if the middleboxes need to be scaled up [103]), the network administrator may use a combination of host-level (e.g., XTrace [100]) and network-level (e.g., [45]) logs to break down the delay for each request into per-segment components as shown. Because ORIGINBINDING does not hold, it is difficult to correlated the logs to track flows [159, 176].

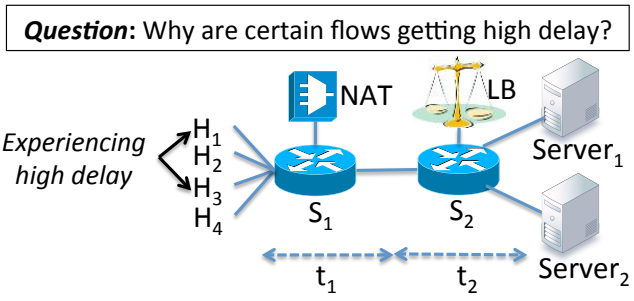


Figure 3-2: Middlebox modifications make it difficult to consistently correlate network logs for diagnosis.

Data-dependent policies: In Figure 3-3, the light IPS checks simple features (e.g., headers) and we want to route suspicious packets to the heavy IPS, which runs deeper analysis to determine if the packet is malicious. Such a triggered architecture is quite common; e.g., rerouting suspicious packets to dedicated packet scrubbers [42]. The problem here is that

ensuring CONTEXTBINDING depends on the *processing history*; i.e., did the light IPS flag a packet as suspicious? However, each switch and middlebox can only make processing or forwarding decisions on a link-local view.

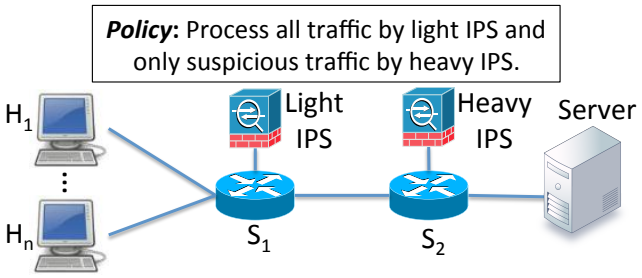


Figure 3-3: S_2 cannot decide if an incoming packet should be sent to the heavy IPS or the server.

Policy violations due to middlebox actions: Figure 3-4 shows a proxy used in conjunction with an access control device (ACL). Suppose we want to block H_2's access to `xyz.com`. However, that H_2 may bypass the policy by accessing cached versions of `xyz.com`, thus evading the ACL. The problem, therefore, is that middlebox actions may violate CONTEXTBINDING by introducing unforeseen paths. In this case, we may need to explicitly route the cached responses to the ACL device as well.

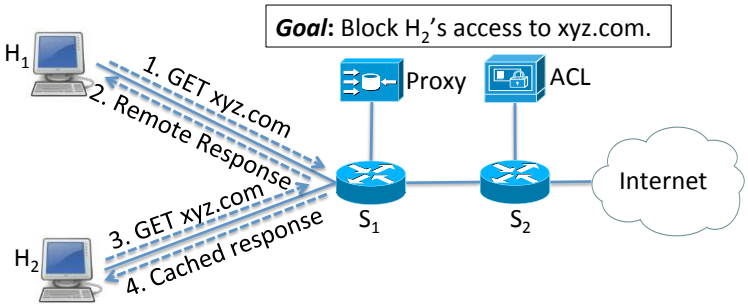


Figure 3-4: Lack of visibility into the middlebox context (i.e., cache hit/miss in this example) makes policy enforcement challenging.

3.2 Related work

Next, we highlight why some seemingly natural strawman solutions fail to address the above problems. Due to space constraints, we discuss only a few salient candidates and summarize their effectiveness in the previously presented examples in Table 3.1.

Placement constraints: One way to ensure ORIGINBINDING/CONTEXTBINDING is to "hardwire" the policy into the topology. In Figure 3-1, we could place the firewall before

Strawman solution	Attribution (Figure 3-1)	Diagnosis (Figure 3-2)	Data-dependent policy (Figure 3-3)	Policy violations (Figure 3-4)
Placement	Yes, if we alter policy chains	No	If both IPSes are on S_1 & Light IPS has 2 ports	Yes
Tunneling (e.g, [117, 121])	No	No	Need IPS support	No
Consolidation (e.g., [164])	Not with separate modules	No	Maybe, if shim is aware	
Correlation (e.g., [158])	Not accurate and high overhead			

Table 3.1: Analyzing strawman solutions vs. the motivating examples.

the NAT. Similarly, for Figure 3-3 we could connect the light IPS and the heavy IPS to S_1 and configure the light IPS to emit legitimate/suspicious packets on different output ports. S_1 can then use the incoming port to determine if the packet should be sent to the heavy IPS. This coupling between policy and topology, however, violates the SDN philosophy of decoupling the control logic from the data plane. Furthermore, this restricts flexibility to reroute under failures, load balance across middleboxes, or customize policies for different workloads [159].

Tunneling: Another option to ensure CONTEXTBINDING is to set up tunneling rules, for example, using MPLS or virtual circuit identifiers (VCI). For instance, we could tunnel packets from the "suspicious" output of the light IPS to the heavy IPS. (Note that this additionally requires middleboxes to support tunnels.) Such topology/tunneling solutions may work for simple examples but they quickly break for more complex policies; e.g., if there are more outputs from the light IPS. Note, that even by combining placement+tunneling, we cannot solve the diagnosis problem in Figure 3-2, as it does not provide ORIGINBINDING.

Middlebox consolidation: At first glance, it may seem that we can ensure CONTEXTBINDING by running *all* middlebox functions on a consolidated platform [68, 164]. While consolidation provides other benefits (e.g., reduced hardware costs), it has several limitations. First, this requires a significant network infrastructure change. Second, this merely shifts the burden of CONTEXTBINDING to the internal routing "shim" that routes packets between the modules. Finally, if the individual modules are provided by different vendors, diagnosis and attribution is hard, as this shim cannot ensure ORIGINBINDING.

Flow correlation: Prior work attempts to heuristically correlate the payloads of the traffic entering and leaving middleboxes to correlate flows [158]. However, this approach can too often result in missed/false matches to be useful for security applications [158]. Also, such "reverse engineering" approaches fundamentally lack ground truth. Finally, this process has high overhead, as multiple packets per flow need to be processed at the controller in a stateful manner (e.g., when reassembling packet payloads).

As Table 3.1 shows, none of these strawman solutions can address all of the motivating scenarios. In some sense, each approach partially addresses some *symptoms* of the violations of ORIGINBINDING and CONTEXTBINDING, but does not address the *cause* of the problem. Thus, despite the complexity they entail in terms of topology hacks, routing, and middlebox and controller upgrades, they have limited applicability and have fundamental correctness limitations.

Beyond the above strawman solutions, for completeness, we also discuss focus on other classes of related work.

Middlebox policy routing: Prior work focuses on orthogonal aspects such as middlebox load balancing (e.g., [135, 158]) or compact data plane strategies (e.g,. [91]). While these are candidates for translating the $CDPG$ to a $CDPGImpl$ (§3.3.4), they do not provide reliable mechanisms to address dynamic middlebox actions.

Middlebox-SDN integration: OpenNF [105] focuses on exposing the internal state (e.g., cache contents and connection state) of middleboxes to enable (virtual) middlebox migration and recovery. This requires significantly more instrumentation and vendor support compared to FlowTags, which only requires externally relevant mappings. Stratos [103] and Slick [69] focus on using SDN to dynamically instantiate new middlebox modules in response to workload changes. The functionality these provide is orthogonal to FlowTags.

Tag-based solutions: Tagging is widely used to implement Layer2/3 functions, such as MPLS labels or virtual circuit identifiers (VCI). In the SDN context, tags have been used to avoid loops [158], reduce FlowTable sizes [91], or provide virtualized network views [145]. Tags in FlowTags capture higher-layer semantics to address ORIGINBINDING and CONTEXTBINDING. Unlike these Layer2/3 mechanisms where switches are generators and consumers of tags, FlowTags middleboxes generate and consume tags and switches are

consumers.

Tracing and provenance: The idea of flow tracking has parallels in the systems (e.g., tracing [100]), databases (e.g., provenance [186]), and security (e.g., taint tracking [147, 152]) literature. Our specific contribution is to use flow tracking for integrating middleboxes in SDN-capable networks.

3.3 System design

3.3.1 FlowTags overview

As we saw in the previous section, violating the ORIGINBINDING and CONTEXTBINDING tenets makes it difficult to correctly implement several network management tasks. To address this problem, we propose the FlowTags architecture. In this section, we highlight the main intuition behind FlowTags, and then we show how FlowTags extends the SDN paradigm.

FlowTags takes a first-principles approach to ensure that ORIGINBINDING and CONTEXTBINDING hold even in the presence of middlebox actions. Since the middleboxes are in the best (and sometimes the only) position to provide the relevant context (e.g., a proxy's cache hit/miss state or a NAT's public-private IP mappings), we argue that middleboxes need to be extended in order to be integrated into SDN frameworks.

Conceptually, middleboxes add tags to outgoing packets. These tags provide the missing bindings to ensure ORIGINBINDING and the necessary processing context to ensure CONTEXTBINDING. The tags are then used in the data plane configuration of OpenFlow switches and other downstream middleboxes.

To explain this high-level idea, let us revisit the example in Figure 3-1 and extend it with the relevant tags and actions as shown in Figure 3-5. We have three hosts $H_1 - H_3$ in an RFC1918 private address space; the administrator wants to block the Internet access for H_1, H_3, and allow H_2's packets to pass through without going to the firewall. The controller (not shown) configures the NAT to associate outgoing packets from H_1, H_2, and H_3 with the tags 1, 2, and 3, respectively, and adds these to pre-specified header fields. (See §3.3.5). The controller configures the firewall so that it can *decode* the tags to map the observed

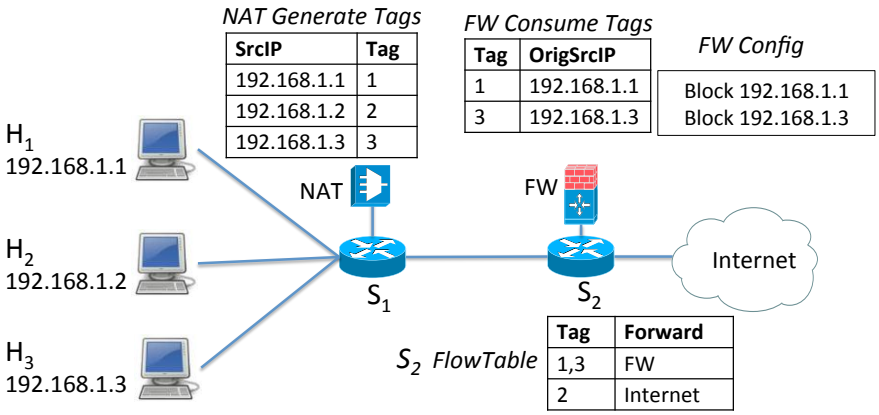


Figure 3-5: Figure 3-1 augmented to illustrate how tags can solve the attribution problem.

IP addresses (i.e., in "public" address space using RFC1918 terminology) to the original hosts, thus meeting the ORIGINBINDING requirement. Similarly, the controller configures the switches to allow packets with tag 2 to pass through without going to the firewall, thus meeting the CONTEXTBINDING requirement. As an added benefit, the administrator can configure firewall rules w.r.t. the original host IP addresses, without needing to worry about the NAT-induced modifications.

This example highlights three key aspects of the FlowTags approach. First, middleboxes (e.g., the NAT) are *generators* of tags (as instructed by the controller). The packet-processing actions of a FlowTags-enhanced middlebox will now entail adding the relevant tags into the packet header. This is crucial for both ORIGINBINDING and CONTEXTBINDING, depending on the middlebox. Second, other middleboxes (e.g., the firewall) are *consumers* of tags, and their processing actions need to decode the tags. This is necessary for ORIGINBINDING. Third, SDN-capable switches in the network use the tags as part of their forwarding actions, in order to route packets according to their intended policy, ensuring CONTEXTBINDING holds. In this simple example, the middlebox exclusively generate/-consume tags. In general, however, a given middlebox will both consume and generate tags.

Note that the FlowTags semantics apply in the context of a *single administrative domain*. In the simple case, we set tag bits to NULL on packets exiting the domain.[1] This

[1]More generally, if we have a domain hierarchy (e.g., "CS dept" and "Physics dept" and "Univ" at higher level), each sub-domain's egress can rewrite the tag to only capture higher-level semantics (e.g, "CS" rather

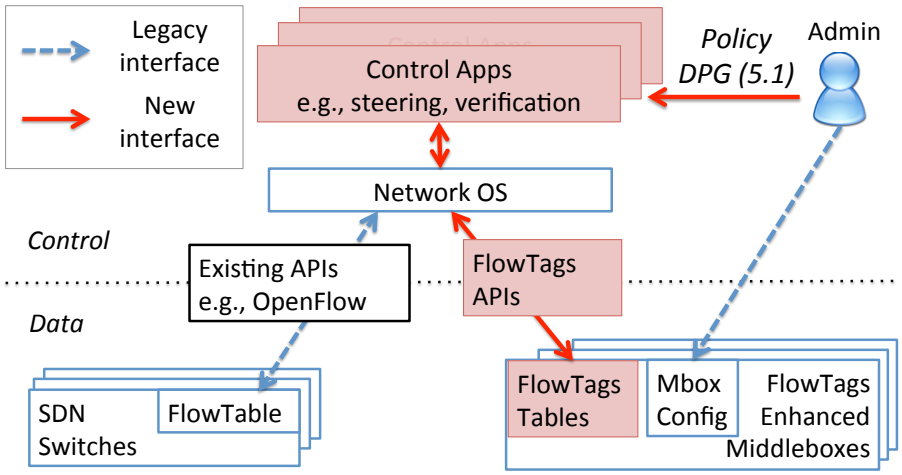


Figure 3-6: Interfaces between different components in the FlowTags architecture.

alleviates concerns that the tag bits may accidentally leak proprietary topology or policy information. When packets arrive on the external interface, the gateway sets the tag bits to the appropriate value (e.g., to ensure stateful middlebox traversal) before forwarding the packet inside the domain.

3.3.2 Architecture and interfaces

Given this high-level intuition, next we describe the interfaces between the controller, middleboxes, switches, and the network administrator in a FlowTags-enhanced SDN architecture.

Current SDN standards (e.g., OpenFlow [141]) define the APIs between the controller and switches. As shown in Figure 4-5, FlowTags extends today's SDN approach along three key dimensions:

1. **FlowTags APIs** between the controller and FlowTags-enhanced middleboxes to programmatically configure the tag generation and consumption logic (§3.3.3).

2. **FlowTags control modules** that configure the tagging-related generation/consumption behavior of the middleboxes and tag-related forwarding actions of SDN switches (§3.3.4).

3. **FlowTags-enhanced middleboxes** consume an incoming packet's tags when processing the packet and generate new tags based on the context (§3.3.6).

---

than "CS host A"), without revealing internal details.

FlowTags requires neither new capabilities from SDN switches, nor any direct interactions between middleboxes and switches. Switches continue to use traditional SDN APIs such as OpenFlow. The only interaction between switches and middleboxes is indirect, via tags embedded inside the packet headers. We take this approach for two reasons: (1) to allow switch and middlebox designs and their APIs to innovate independently; and (2) to retain compatibility with existing SDN standards (e.g., OpenFlow). Embedding tags in the headers avoids the need for each switch and middlebox to communicate with the controller on every packet when making their forwarding and processing decisions.

We retain existing configuration interfaces for customizing middlebox actions; e.g., vendor-specific languages or APIs to configure firewall/IDS rules. The advantage of FlowTags is that administrators can configure these rules without having to worry about the impact of intermediate middleboxes. For example, FlowTags allows the operator to specify firewall rules with respect to the original source IPs in the first scenario of §3.1. This provides a cleaner mechanism, as the administrator does not have to reason about the space of possible header values a middlebox may observe.[2]

3.3.3 FlowTags APIs and operation

Next, we walk through how a packet is processed in a FlowTags-enhanced network, and describe the main FlowTags APIs. For ease of presentation, we assume each middlebox is connected to the rest of the network via a switch. (FlowTags, as is, works in a topology that middleboxes are chained together.) We restrict our description of FlowTags to a *reactive* controller that responds to incoming packets.

In the interest of brevity, we only discuss the APIs pertaining to packet processing. Analogous to the OpenFlow configuration APIs, we envision functions to obtain and set FlowTags capabilities in middleboxes; e.g., which fields in the header are used to encode the tag values (§3.3.5).

In general, the same middlebox can be both a *generator* and the *consumer* of the tags. For clarity for describing our APIs, we focus on these two roles separately starting with

[2]Going forward, we want to configure the middlebox rules to ensure the HIGHLEVELNAMES as well [81].

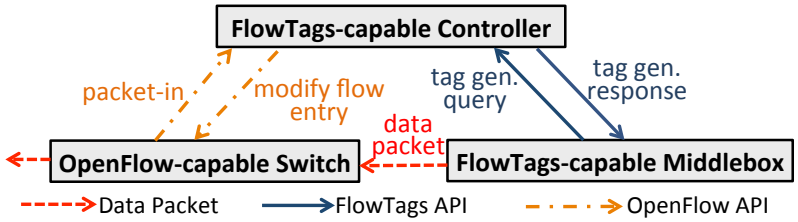


Figure 3-7: Packet processing walkthrough for tag generation with the FlowTags APIs.

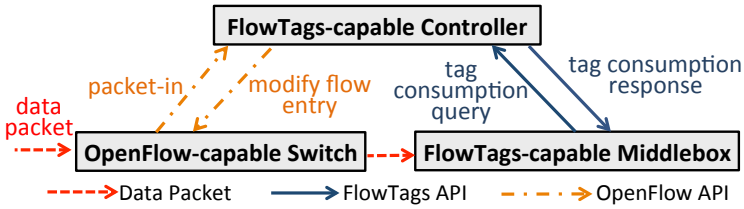


Figure 3-8: Packet processing walkthrough for tag consumption with the FlowTags APIs

generation, assuming that a packet starts with a NULL tag before it reaches any middlebox.

Middlebox tag generation, Figure 3-7: Before the middlebox outputs a processed (and possibly modified) packet, it sends the `FT_GENERATE_QRY` message to the controller requesting a tag value to be added to the packet (Step 1). As part of this query the middlebox provides the relevant packet processing context; e.g., a proxy tells the controller if this is a cached response or an IPS provides the processing verdict. The controller provides a tag value via the `FT_GENERATE_RSP` response (Step 2). (We defer tag semantics to the next section.)

Middlebox tag consumption, Figure 3-8: When a middlebox receives a tag-carrying packet, it needs to "decode" this tag; e.g., an IDS needs the original IP 5-tuple for scan detection. The middlebox sends the `FT_CONSUME_QRY` message (Step 5) to the controller, which provides the necessary decoding rule for mapping the tag via the `FT_CONSUME_RSP` message (Step 6).

Switch actions: In Figure 3-7, when the switch receives a packet from the middlebox with a tag (Step 3), it queries the controller with the `OFPT_PACKET_IN` message (Step 4), and the controller provides a new flow table entry (Step 5). This determines the forwarding action; e.g., whether this packet should be routed toward the heavy IPS in Figure 3-3. Similarly, when the switch receives a packet in Figure 3-8 (Step 1), it requests a forwarding entry and the controller uses the tag to decide if this packet needs to be forwarded to the

middlebox.

Most types of middleboxes operate at a IP flow or session granularity and their dynamic modifications typically use a consistent header mapping for all packets in the same flow. Thus, analogous to OpenFlow, we need to run the `FT_CONSUME_QRY` and the `FT_GENERATE_QRY` only *once per flow* at a middlebox. The middlebox stores the per-flow tag rules locally and subsequent packets in the same flow can reuse the cached tag rules.

3.3.4 FlowTags controller

Here we discuss how a FlowTags-enhanced SDN controller can assign tags and tags-related "rules" to middleboxes and switches. We begin with a policy abstraction that informs the semantics that tags need to express. Then, we discuss techniques to translate this solution into practical encodings. Finally, we outline the controller's implementation.

From CDPG to Tag Semantics: The input to the FlowTags controller is the context-dependent*policy* that the administrator wants to enforce w.r.t. middlebox actions (Figure 4-5) as a *context-dependent policy graph* (or $CDPG$) from §2.3.4. We assume that the administrator creates the CDPG based on domain knowledge. The CDPG representation helps us reason about the semantics we need to capture via tags to ensure ORIGINBINDING and CONTEXTBINDING.

Restoring ORIGINBINDING: We can ensure ORIGINBINDING if we are always able to map a packet to its original IP 5-tuple $OrigHdr$ as it traverses a CDPG. Note that having $OrigHdr$ is a *sufficient* condition for ORIGINBINDING; i.e., given the $OrigHdr$ any downstream middlebox or switch can conceptually implement the action intended by a CDPG. In some cases such as per-flow diagnosis (Figure 3-2), mapping a packet to the $OrigHdr$ is necessary. In other examples, a coarser identifier may be enough; e.g., just srcIP in Figure 3-1.

Restoring CONTEXTBINDING: To ensure CONTEXTBINDING, we essentially need to capture the edge condition $m \rightarrow m'$. Recall that this condition depends on (1) the traffic class and (2) the middlebox context from logical middlebox m (and possibly previous logical middleboxes) denoted by NC. Given that the $OrigHdr$ for ORIGINBINDING provides

the necessary context to determine the traffic class, the only additional required information the middlebox context in $m \rightarrow m'$.

Conceptually, and assuming no constraints on the tag identifier space, we can think of the controller assigning a globally unique tag T to each "located packet"; i.e., a packet along with the edge on the CDPG [162]. The controller maps the tag of each located packet to the information necessary for ORIGINBINDING and CONTEXTBINDING: $T \rightarrow \langle OrigHdr, NC \rangle$. Here, the $OrigHdr$ represents the original IP 5-tuple of this located packet when it first enters the network (i.e., before any middlebox modifications) and NC captures the processing context of this located packet.

In the context of tag consumption from §3.3.3, `FT_CONSUME_QRY` and `FT_CONSUME_RSP` essentially "dereference" tag T to obtain the $OrigHdr$. The middlebox can apply its processing logic based on the $OrigHdr$; i.e., satisfying ORIGINBINDING.

For tag generation at logical middlebox m, `FT_GENERATE_QRY` provides as input to the controller: (1) the necessary middlebox context to determine which NC will apply, and (2) the tag T of the incoming packet that triggered this new packet to be generated. The controller creates a new tag T' entry for this new located packet and populates the entry $T' \rightarrow \langle OrigHdr', NC \rangle$ for this new tag as follows. First, it uses $OrigHdr$ (i.e., for the input tag T) to determine the value $OrigHdr'$ for T'. In many cases (e.g., NAT), this is a simple copy. In some cases (e.g., proxy response), the association has to reverse the src/dst mappings in $OrigHdr$. Second, it associates the new tag T' with the new NC provided by the middlebox. The controller instructs the middlebox via `FT_GENERATE_RSP` to add T' to the packet header. Because T' is mapped to NC, it helps enforce CONTEXTBINDING.

To summarize, the DPG captures the necessary semantics to successfully restore ORIGINBINDING and CONTEXTBINDING.

3.3.5 Encoding tags in headers

In practice, we need to embed the tag value in a finite number of packet-header bits. IPv6 has a 20-bit *Flow Label* field, which seems ideal for this use (thus answering the question "how should we use the flow-label field?" [66]). For our current IPv4 prototype and testbed, we used the 6-bit DS field (part of the 8-bit ToS), which sufficed for our scenarios. To

deploy FlowTags on large-scale IPv4 networks, we would need to borrow bits from fields that are not otherwise used. For example, if VLANs are not used, we can use the 12-bit VLAN Identifier field. Or, if all traffic sets the DF (Don't Fragment) IP Flag, which is typical because of Path MTU Discovery, the 16-bit IP_ID field is available.[3]

Next, we discuss how to use these bits as efficiently as possible; §3.4 reports on some analysis of how many bits might be needed in practice.

As discussed earlier, tags restore ORIGINBINDING and CONTEXTBINDING. Conceptually, we need to be able to distinguish every located packet—i.e., the combination of all flows and all possible paths in the DPG. Thus, a simple upper bound on the number of bits in each packet to distinguish between $|Flows|$ flows and $|DPGPaths|$ processing paths is: $\log_2 |Flows| + \log_2 |DPGPaths|$, where $Flows$ is the set of IP flows (for ORIGINBINDING), and $DPGPaths$ is the set of possible paths a packet could traverse in DPG (for CONTEXTBINDING). However, this grows log-linearly in the number of flows over time and the number of paths (which could be exponential w.r.t. the graph size). This motivates optimizations to reduce the number of header bits necessary:

- **Coarser tags:** For many middlebox management tasks, it may suffice to use a tag to identify the logical traffic class (e.g., "CS Dept User") and the local middlebox context (e.g., 1 bit for cache hit or miss or 1 bit for "suspicious"), rather than individual IP flows.
- **Temporal reuse:** We reuse the tag assigned to a flow after the flow expires; we detect expiration via explicit flow termination, or via timeouts [141]. The controller tracks active tags and finds an unused value for a new tag.
- **Spatial reuse:** To address ORIGINBINDING, we only need to ensure that the new tag does not conflict with tags already assigned to currently active flows at the middlebox to which this packet is destined. For CONTEXTBINDING, we only need: (1) capture the most recent edge on the CDPG rather than the entire path (i.e., reducing from $|DPGPaths|$ to the node degree); and (2) ensure that the switches on the path have no ambiguity in the forwarding decision w.r.t. other active flows.

Putting it Together: Our design is a *reactive* controller that responds to

[3]IP_ID isn't part of the current OpenFlow spec; but it can be supported with support for flexible match options [38, 75].

OFPT_PACKET_IN, FT_CONSUME_QRY, and FT_GENERATE_QRY events from the switches and the middleboxes.

Initialization: Given an input CDPG, we generate a data plane realization $CDPGImpl$; i.e., for each logical middlebox m we need to identify candidate *physical middlebox instances* and for each edge in CDPG we find a switch-level path between corresponding physical middleboxes. This translation should also take into account considerations such as load balancing across middleboxes and resource constraints (e.g., switch TCAM, link capacity). While FlowTags is agnostic to the specific realization, we currently use SIMPLE [158], mostly because of our familiarity with the system. (This procedure only needs to run when the $CDPG$ itself changes or in case of network topology change, not per flow arrival.)

Middlebox event handlers: For each physical middlebox instance PM_i, the controller maintains two FlowTags tables: $CtrlInTagsTable_i$ and the $CtrlOutTagsTable_i$. The $CtrlInTagsTable_i$ maintains the tags corresponding to all active flows *incoming* into this middlebox and maintains a table of entries $\{T \rightarrow OrigHdr\}$. The $CtrlOutTagsTable_i$ tracks the tags that need to be assigned to *outgoing* flows and maintains a table of entries $\{\langle T, NC\rangle \rightarrow T'\}$, where T is the tag for the incoming packet, NC captures the relevant middlebox context for this flow (e.g., cache hit/miss), and T' is the output tag to be added. At bootstrap time, these structures are initialized to be empty.

The HANDLE_FT_CONSUME_QRY looks up the entry for tag T in the $CtrlInTagsTable_i$ and sends the mapping to PM_i. As we will see in the next section, middleboxes keep these entries in a FlowTable-like structure to avoid lookups for subsequent packets. The HANDLE_FT_GENERATE_QRY is slightly more involved, as it needs the relevant middlebox context NC. Given this context, the $CDPG$, and the $CDPGImpl$, the controller identifies the next hop physical middlebox $PM_{i'}$ for this packet. It also determines a non-conflicting T' using the logic from §3.3.5.

Switch and flow expiry handlers: The handlers for OFPT_PACKET_IN are similar to traditional OpenFlow handlers; the only exception is that we use the incoming tag to determine the forwarding entry. When a flow expires, we trace the path this flow took and, for each PM_i, delete the entries in $CtrlInTagsTable_i$ and $CtrlOutTagsTable_i$, so that these

tags can be repurposed.

3.3.6 FlowTags-enhanced middleboxes

As discussed in the previous sections, FlowTags requires middlebox support. We begin by discussing two candidate design choices for extending a middlebox to support FlowTags. Then, we describe the conceptual operation of a *FlowTags-enhanced* middlebox. We conclude this section by summarizing our experiences in extending five software middleboxes.

Extending Middleboxes: We consider two possible ways to extend middlebox software to support FlowTags:

- **Module modification:** The first option is to modify specific internal functions of the middlebox to consume and generate the tags. For instance, consider an IDS with the scan detection module. Module modification entails patching this scan detection logic with hooks to translate the incoming packet headers+tag to the $OrigHdr$ and to rewrite the scan detection logic to use $OrigHdr$. Similarly, for generation, we modify the output modules to provide the relevant context as part of the `FT_GENERATE_QRY`.
- **Packet rewriting:** A second option is to add a lightweight shim module that *interposes* on the incoming and outgoing packets to rewrite the packet headers. For consumption, this means we modify the packet headers so that the middlebox only sees a packet with the true $OrigHdr$. For generation, this means that the middlebox proceeds as-is and then the shim adds the tag before the packet is sent out.

In both cases, the administrator sets up the middlebox configuration (e.g., IDS rules) as if there were no packet modifications induced by the upstream middleboxes because FlowTags preserves the binding between the packet's modified header and the $OrigHdr$.

For consumption, we prefer packet rewriting because it generalizes to the case where each middlebox has multiple "consumer" modules; e.g., an IDS may apply scan detection and signature-based rules. For generation, however, packet rewriting may not be sufficient, as the shim may not have the necessary visibility into the middlebox context; e.g., in the proxy cache hit/miss case. Thus, we use module modification in this case.

End-to-end view: Figure 3-9 shows a simplified view of a FlowTags-enhanced middlebox. In general, *consumption* precedes *generation*. The reason is that the packet's current tag

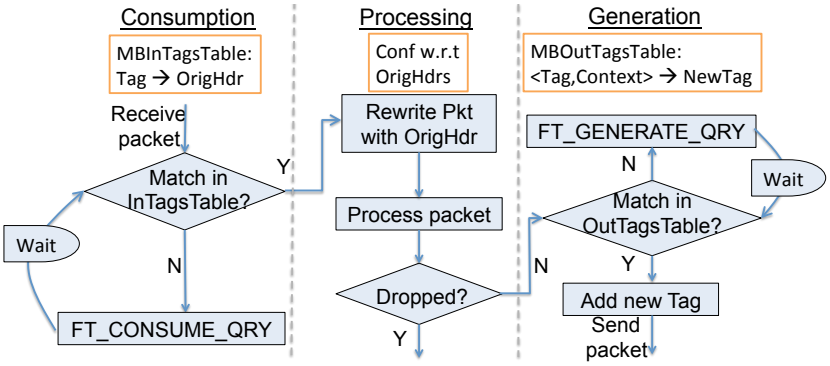


Figure 3-9: We choose a hybrid design where the "consumption" side uses the packet rewriting and the "generation" uses the module modification option.

can affect the specific middlebox code paths, and thus impacts the eventual outgoing tags.

Mirroring the controller's $CtrlInTagsTable_i$ and $CtrlOutTagsTable_i$, each physical middlebox i maintains the tag rules in the $MBInTagsTable_i$ and $MBOutTagsTable_i$. When a packet arrives, it first checks if the tag value in the packet already matches an existing tag-mapping rule in $MBInTagsTable_i$. If there is a match, we rewrite packet headers (see above) so that the processing modules act as if they were operating on $OrigHdr$. If there is a $MBInTagsTable_i$ miss, the middlebox sends a `FT_CONSUME_QRY`, buffers the packet locally, and waits for the controller's response.

Note that the tags are logically propagated through the processing contexts (not shown for clarity). For example, most middleboxes follow a connection-oriented model with a data structure maintaining per-flow or per-connection state; we augment this structure to propagate the tag value. Thus, we can causally relate an outgoing packet (e.g., a NAT-ed packet or a proxy cached response) to an incoming packet.

When a specific middlebox function or module is about to send a packet forward, it checks the $MBOutTagsTable_i$ to add the outgoing tag value. If there is a miss then it sends the `FT_GENERATE_QRY`, providing the necessary module-specific context and the tag (from the connection data structure) for the incoming packet that caused this outgoing packet to be generated.

Experiences in extending middleboxes: Given this high-level view, next we describe our experiences in modifying five software middleboxes that span a broad spectrum of management functions. (Our choice was admittedly constrained by the availability of the middlebox source code.) Table 3.2 summarizes the middleboxes and the modifications

Name, Role	Modified, Total LOC	Key Modules	Data Structures
Squid [49], Proxy	75, 216K	Client and Server Side Connection, Forward, Cache Lookup	Request Table
Snort [48], IDS/IPS	45, 336K	Decode, Detect, Encode	Verdict
Balance [8], Load Balancer	60, 2K	Client and Server Connections	n/a
PRADS [41], Monitoring	25, 15K	Decode	n/a
iptables [26], NAT	55, 42K	PREROUTING, POSTROUTING	Conn Map

Table 3.2: Summary of the middleboxes we have added FlowTags support to with the number of lines of code and the main modules to be updated. We use a common library ($\approx$ 250 lines) that implements routines for communicating to the controller.

necessary.

Our current approach to extend middleboxes is semi-manual and involved a combination of call graph analysis [29, 56] and traffic injection and logging techniques [10, 23, 25, 51]. Based on these heuristics, we identify the suitable "chokepoints" to add the FlowTags logic. Developing techniques to automatically extend middleboxes is an interesting direction for future work.

- **Squid:** Squid [49] is a popular proxy/cache. We modified the functions in charge of communicating with the client, remote server, and those handling cache lookup. We used the packet modification shim for incoming packets and apply module modification to handle the possible packet output cases based on cache hit and miss events.
- **Snort:** Snort [48] is an IDS/IPS that provides many functions—logging, packet inspection, packet filtering, and scan detection. Similar to Squid, we applied the packet rewriting step for tag consumption and module modification for tag generation as follows. When a packet is processed and a "verdict" (e.g., OK vs. alarm) is issued, the tag value is generated based on the type of the event (e.g., outcome of a matched alert rule).
- **Balance:** Balance [8] is a TCP-level load balancer that distributes incoming TCP connections over a given a set of destinations (i.e., servers). In this case, we simply read/write the tag bits in the header fields.
- **PRADS:** PRADS [41] is passive monitor that gathers traffic information and infers what hosts and services exist in the network. Since this is a passive device, we only need the packet rewriting step to restore the (modified) packet's $OrigHdr$.
- **NAT via iptables:** We have registered appropriate tagging functions with iptables [26]

hook points while configured as a source NAT such that it maintains 5-tuple visibility via tagging. We added hooks for tag consumption and tag generation into the PREROUTING and the POSTROUTING chains, which are, respectively, the input and output checkpoints.

3.3.7 Implementation

We implement the FlowTags controller as a `POX` module [40]. The $CtrlInTagsTable_i$ and $CtrlOutTagsTable_i$ are implemented as hash-maps. For memory efficiency and fast look up of available tags, we maintain an auxiliary bitvector of the active tags for each middlebox and switch interface; e.g., if we have 16-bit tags, we maintain a 2^{16} bit vector and choose the first available bit, using a log-time algorithm [71]. We also implement simple optimizations to precompute shortest paths for every pair of physical middleboxes.

3.4 Evaluation

We address the following questions regarding the performance and scalability of FlowTags:

- Q1: What overhead does supporting FlowTags add to middlebox processing?
- Q2: Is the FlowTags controller fast and scalable?
- Q3: What is the overhead over traditional SDN in a FlowTags-enhanced network?
- Q4: How many tag bits do we need in practice?

Setup: For the microbenchmarks (Q1 and Q2), we run each middlebox and POX controller in isolation on a single core in a 32-core 2.6 Ghz Xeon server with 64 GB RAM. For the end-to-end experiments (Q3), we use Mininet [32] on the same server configured to use 24 cores and 32 GB RAM to model the network switches and hosts. We augment Mininet with middleboxes running as external virtual appliances. Each middlebox runs as a VM configured with 2GB memory on one CPU core. (The number of middlebox instances is limited to 28 due to the maximum number of PCI interfaces that can be plugged-in using KVM [31]). We emulate the example topologies from §3.1 and larger PoP-level ISP topologies from RocketFuel [169]. Our default DPG has an average path length of 3.

Q1 Middlebox overhead: We configure each middlebox to run with the default configuration. We vary the offered load (up to 100 Mbps) and measure the per-packet processing

Topology (*#nodes*)	Baseline (*ms*)	Optimized (*ms*)
Abilene (*11*)	0.037	0.024
Geant (*22*)	0.066	0.025
Telstra (*44*)	0.137	0.026
Sprint (*52*)	0.161	0.027
Verizon (*70*)	0.212	0.028
ATT (*115*)	0.325	0.028

Table 3.3: Time to run HANDLE_FT_GENERATE_QRY**.**

latency. Overall, the overhead was low ($<1\%$) and independent of the offered load (not shown). We also analyzed the additional memory and CPU usage using `atop` and it was $< 0.5\%$ across all experiments (not shown).

Q2 Controller scalability: Table 3.3 shows the running time for the HANDLE_FT_GENERATE_QRY. (This is the most complex FlowTags processing step; other functions take negligible time.) The time is linear as a function of topology size with the baseline algorithms but almost constant with the optimization to pre-compute reachability information. This implies that a single-thread POX controller can handle $\frac{1}{0.028ms} \approx 35K$ middlebox queries per second (more than three times larger than the peak number of flows per second reported in [81]).

We also varied the DPG complexity along three axes: number of nodes, node degrees, and distance between adjacent DPG nodes in terms of number of switches. With route precomputation, the controller processing time is independent of the DPG complexity (not shown).

Q3 End-to-end overhead: Figure 3-10 shows the breakdown of different components of the flow setup time in a FlowTags-enhanced network (i.e., mirroring the steps in Figure 3-7) for different Rocketfuel topologies. Since our goal is to compare the FlowTags vs. SDN operations, we do not show round-trip times to the controller here, as it is deployment-specific [112]. (FlowTags adds 1 more RTT per-middlebox, but this can be avoided by pre-fetching rules for the switches and middleboxes.) Since the values are close to the average, we do not show error bars. We can see that the FlowTags operations add negligible overhead. In fact, the middlebox tag processing is so small that it might be hard to see in the figure.

We also measure the reduction in TCP throughput a flow experiences in a FlowTags-

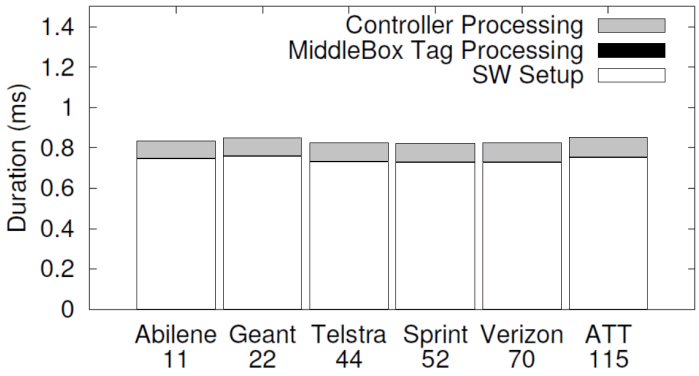


Figure 3-10: Breakdown of flow processing time in different topologies (annotated with #nodes).

Flow size (pkts)	Reduction in throughput %		
	1ms RTT	10ms RTT	20ms RTT
2	12	16.2	22.7
8	2.1	2.8	3.8
32	1.6	2.3	3.0
64	1.5	2.1	2.9

Table 3.4: Reduction in TCP throughput with FlowTags relative to a pure SDN network.

enhanced network compared to a traditional SDN network with middleboxes (but without FlowTags). We vary two parameters: (1) controller RTT and (2) the number of packets per flow. As we can see in Table 3.4, except for very small flows (2 pkts), the throughput reduction is $<4\%$.

Q4 Number of tag bits: To analyze the benefits of the spatial and temporal reuse, we consider the worst-case where we want to diagnose each IP flow. We use packet traces from CAIDA (Chicago and San Jose traces, 2013 [12]) and a flow-level enterprise trace [57]. We simulate the traces across the RocketFuel topologies, using a gravity model to map flows to ingress/egress nodes [169].

Table 3.5 shows the number of bits necessary with different reuse strategies on the AT&T topology.[4] The results are similar across other topologies (not shown). We see that temporal reuse offers the most reduction. Spatial reuse helps only a little; this is because with gravity model workload, there is typically a "hotspot" with many concurrent flows.

[4]Even though the #flows is different across traces, the #bits is identical, as the values of $ceil$ of $\log_2$ of the #flows is the same.

Configuration (spatial, temporal)	Number of bits	
	CAIDA trace	Enterprise trace
No spatial; 30 sec	22	22
Spatial; 30 sec	20	20
Spatial; 10 sec	18	18
Spatial; 5 sec	17	17
Spatial; 1 sec	14	14

Table 3.5: Effect of spatial and temporal reuse of tags.

To put this in the context of §3.3.5, using the <Spatial, 1 sec> configuration, tags can fit in the IPv6 FlowLabel, and would fit in the IPv4 IP_ID field.

3.5 Case study: FlowTags as an enabler for flexible and elastic DDoS defense

While in the context of this thesis FlowTags is used to expose hidden traffic processing context to enable testing context-dependent policies, the ability to explicitly capture context turns out to be an enabler for building more flexible policy enforcement frameworks. To demonstrate this, here we briefly present how we have employed FlowTags to build a flexible and elastic DDoS defense system called Bohatei [95]. (In addition to Bohatei, we have also used FlowTags to build a scalable enterprise network security framework called PSI [181].)

In spite of extensive industrial and academic efforts (e.g., [4, 143, 146]), distributed denial-of-service (DDoS) attacks continue to plague the Internet. Over the last few years, we have observed a dramatic escalation in the number, scale, and diversity of DDoS attacks. For instance, recent estimates suggest that over 20,000 DDoS attacks occur per day [151], with peak volumes of 0.5 Tbps [35,87]. At the same time, new vectors [122,171] and variations of known attacks [163] are constantly emerging. The damage that these DDoS attacks cause to organizations is well-known and include both monetary losses (e.g., $40,000 per hour [24]) and loss of customer trust.

DDoS defense today is implemented using expensive and proprietary hardware appliances (deployed in-house or in the cloud [16, 42]) that are *fixed* in terms of placement, functionality, and capacity. First, they are typically deployed at fixed network aggregation points (e.g., a peering edge link of an ISP). Second, they provide fixed functionality with

respect to the types of DDoS attacks they can handle. Third, they have a fixed capacity with respect to the maximum volume of traffic they can process. This fixed nature of today's approach leaves network operators with two unpleasant options: (1) to overprovision by deploying defense appliances that can handle a high (but pre-defined) volume of every known attack type at each of the aggregation points, or (2) to deploy a smaller number of defense appliances at a central location (e.g., a scrubbing center) and reroute traffic to this location. While option (2) might be more cost-effective, it raises two other challenges. First, operators run the risk of underprovisioning. Second, traffic needs to be explicitly routed through a fixed central location, which introduces additional traffic latency and requires complex routing hacks (e.g., [173]). Either way, handling larger volumes or new types of attacks typically mandates purchasing and deploying new hardware appliances.

Ideally, a DDoS defense architecture should provide the *flexibility* to seamlessly place defense mechanisms where they are needed and the *elasticity* to launch defenses as needed depending on the type and scale of the attack. We observe that similar problems in other areas of network management have been tackled by taking advantage of two new paradigms: software-defined networking (SDN) [108, 141] and network functions virtualization (NFV) [150]. SDN simplifies routing by decoupling the control plane (i.e., routing policy) from the data plane (i.e., switches). In parallel, the use of virtualized network functions via NFV reduces cost and enables elastic scaling and reduced time-to-deploy akin to cloud computing [150]. These potential benefits have led major industry players (e.g., Verizon, AT&T) to embrace SDN and NFV [5, 7, 36, 58].[5]

Next, we briefly highlight new opportunities that SDN and NFV can enable for DDoS defense.

Lower capital costs: Current DDoS defense is based on specialized hardware appliances (e.g., [4, 44]). Network operators either deploy them on-premises, or outsource DDoS defense to a remote packet scrubbing site (e.g., [16]). In either case, DDoS defense is expensive. For instance, based on public estimates from the General Services Administration (GSA) Schedule, a 10 Gbps DDoS defense appliance costs ≈$128,000 [21]. To put

[5]To quote the SEVP of AT&T: "To say that we are both feet in [on SDN] would be an understatement. We are literally all in [5]."

this in context, a commodity server with a 10 Gbps Network Interface Card (NIC) costs about $3,000 [17]. This suggests roughly 1-2 orders of magnitude potential reduction in capital expenses (ignoring software and development costs) by moving from specialized appliances to commodity hardware.[6]

Time to market: As new and larger attacks emerge, enterprises today need to frequently purchase more capable hardware appliances and integrate them into the network infrastructure. This is an expensive and tedious process [150]. In contrast, launching a VM customized for a new type of attack, or launching more VMs to handle larger-scale attacks, is trivial using SDN and NFV.

Elasticity with respect to attack volume: Today, DDoS defense appliances deployed at network chokepoints need to be provisioned to handle a predefined maximum attack volume. As an illustrative example, consider an enterprise network where a DDoS scrubber appliance is deployed at each ingress point. Suppose the projected resource footprint (i.e., defense resource usage over time) to defend against a SYN flood attack at times t_1, t_2, and t_3 is 40, 80, and 10 Gbps, respectively.[7] The total resource footprint over this entire time period is $3 \times max\{40, 80, 10\} = 240$ Gbps, as we need to provision for the worst case. However, if we could elastically scale the defense capacity, we would only introduce a resource footprint of $40 + 80 + 10 = 130$ Gbps—a 45% reduction in defense resource footprint. This reduced hardware footprint can yield energy savings and allow ISPs to repurpose the hardware for other services.

Flexibility with respect to attack types: Building on the above example, suppose in addition to the SYN flood attack, the projected resource footprint for a DNS amplification attack in time intervals t_1, t_2, and t_3 is 20, 40, and 80 Gbps, respectively. Launching only the required types of defense VMs as opposed to using monolithic appliances (which handle both attacks), drops the hardware footprint by 40%; i.e., from $3 \times (max\{40, 80, 10\} + max\{20, 40, 80\}) = 480$ to 270.

Flexibility with respect to vendors: Today, network operators are locked-in to the defense

[6]Operational expenses are harder to compare due to the lack of publicly available data.

[7]For brevity, we use the traffic volume as a proxy for the memory consumption and CPU cycles required to handle the traffic.

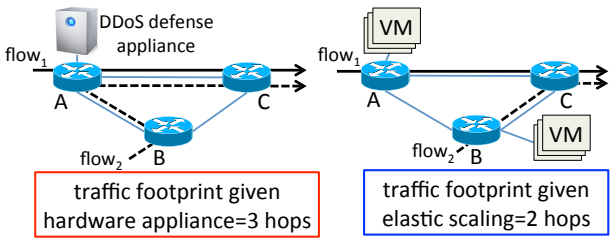


Figure 3-11: DDoS defense routing efficiency enabled by SDN and NFV.

capabilities offered by specific vendors. In contrast, with SDN and NFV, they can launch appropriate best-of-breed defenses. For example, suppose vendor 1 is better for SYN flood defense, but vendor 2 is better for DNS flood defense. The physical constraints today may force an ISP to pick only one hardware appliance. With SDN/NFV we can avoid the undesirable situation of picking only one vendor and rather have a deployment with both types of VMs each for a certain type of attack. Looking even further, we also envision that network operators can mix and match capabilities from different vendors; e.g., if vendor 1 has better detection capabilities but vendor 2's blocking algorithm is more effective, then we can flexibly combine these two to create a more powerful defense platform.

Simplified and efficient routing: Network operators today need to employ complex routing hacks (e.g., [173]) to steer traffic through a fixed-location DDoS hardware appliance (deployed either on-premises or in a remote site). As Figure 3-11 illustrates, this causes additional latency. Consider two end-to-end flows $flow_1$ and $flow_2$. Way-pointing $flow_2$ through the appliance (the left hand side of the figure) makes the total path lengths 3 hops. But if we could launch VMs where they are needed (the right hand side of the figure), we could drop the total path lengths to 2 hops—a 33% decrease in traffic footprint. Using NFV we can launch defense VMs on the closest location to where they are currently needed, and using SDN we can flexibly route traffic through them.

In summary, we observe new opportunities to build a flexible and elastic DDoS defense mechanism via SDN/NFV. In the next section, we highlight the challenges in realizing these benefits.

Deployment scenario: For concreteness, we focus on an ISP-centric deployment model, where an ISP offers DDoS-defense-as-a-service to its customers. Note that several ISPs already have such commercial offerings (e.g., [6]). We envision different monetization

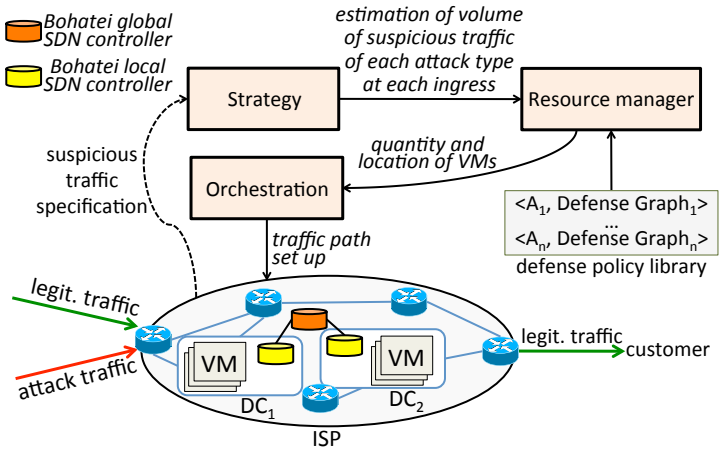


Figure 3-12: Bohatei system overview and workflow. Scalable orchestration is enabled by FlowTags in-data plane tag-based traffic forwarding.

avenues. For example, an ISP can offer a value-added security service to its customers that can replace the customers' in-house DDoS defense hardware. Alternatively, the ISP can allow its customers to use Bohatei as a cloudbursting option when the attack exceeds the customers' on-premise hardware. While we describe our work in an ISP setting, our ideas are general and can be applied to other deployment models; e.g., CDN-based DDoS defense or deployments inside cloud providers [42].

In addition to traditional backbone routers and interconnecting links, we envision the ISP has deployed multiple datacenters as shown in Figure 3-12. Note that this is not a new requirement; ISPs already have several in-network datacenters and are planning additional rollouts in the near future [36, 58]. Each datacenter has commodity hardware servers and can run standard virtualized network functions [155].

Threat model: We focus on a general DDoS threat against the victim, who is a customer of the ISP. The adversary's aim is to exhaust the network bandwidth of the victim. The adversary can flexibly choose from a *set of candidate attacks* $AttackSet = \{A_a\}_a$. As a concrete starting point, we consider the following types of DDoS attacks: TCP SYN flood, UDP flood, DNS amplification, and elephant flow. We assume the adversary controls a large number of bots, but the total *budget* in terms of the maximum volume of attack traffic it can launch at any given time is fixed. Given the budget, the adversary has a complete control over the choice of (1) type and mix of attacks from the $AttackSet$ (e.g., 60% SYN

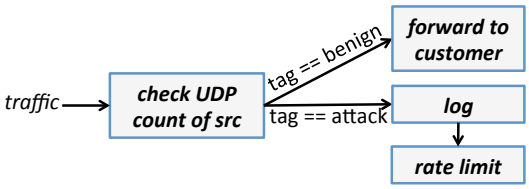


Figure 3-13: A sample defense against UDP flood.

and 40% DNS) and (2) the set of ISP ingress locations at which the attack traffic enters the ISP. For instance, a simple adversary may launch a single fixed attack A_a arriving at a single ingress, while an advanced adversary may choose a mix of various attack types and multiple ingresses. For clarity, we restrict our presentation to focus on a single customer noting that it is straightforward to extend our design to support multiple customers.

Defenses: We assume the ISP has a pre-defined *library of defenses* specifying a defense strategy for each attack type. For each attack type A_a, the defense strategy is specified as a directed acyclic graph DAG_a representing a typical multi-stage attack analysis and mitigation procedure. Each node of the graph represents a logical module and the edges are tagged with the result of the previous nodes processing (e.g., "benign" or "attack" or "analyze further"). Each logical node will be realized by one (or more) *virtual appliance(s)* depending on the attack volume. Figure 3-13 shows an example strategy graph with 4 modules used for defending against a UDP flood attack. Here, the first module tracks the number of UDP packets each source sends and performs a simple threshold-based check to decide whether the source needs to be let through or throttled.

Our goal here is not to develop new defense algorithms but to develop the system orchestration capabilities to enable flexible and elastic defense. As such, we assume the DAGs have been provided by domain experts, DDoS defense vendors, or by consulting best practices.

3.5.1 System design

The workflow of Bohatei has four steps (see Figure 3-12):

1. *Attack detection:* We assume the ISP uses some out-of-band anomaly detection technique to flag whether a customer is under a DDoS attack [72]. The design of this detection algorithm is outside the scope of Bohatei. The detection algorithm gives a

coarse-grained specification of the suspicious traffic, indicating the customer under attack and some coarse identifications of the type and sources of the attack; e.g., "srcprefix=*,dstprefix=cust,type=SYN".

2. *Attack estimation:* Once suspicious traffic is detected, the strategy module estimates the volume of suspicious traffic of each attack type arriving at each ingress.

3. *Resource management:* The resource manager then uses these estimates as well as the library of defenses to determine the type, number, and the location of defense VMs that need to be instantiated. The goal of the resource manager is to efficiently assign available network resources to the defense while minimizing user-perceived latency and network congestion.

4. *Network orchestration:* Finally, the network orchestration module sets up the required network forwarding rules to steer suspicious traffic to the defense VMs as mandated by the resource manager.

Given this workflow, we highlight the three challenges we need to address to realize our vision:

C1. Responsive resource management: We need an efficient way of assigning the ISP's available compute and network resources to DDoS defense. Specifically, we need to decide how many VMs of each type to run on each server of each datacenter location so that attack traffic is handled properly while minimizing the latency experienced by legitimate traffic. Doing so in a *responsive* manner (e.g., within tens of seconds), however, is challenging. Specifically, this entails solving a large NP-hard optimization problem, which can take several hours to solve even with state-of-the-art solvers.

C2. Scalable network orchestration: The canonical view in SDN is to set up switch forwarding rules in a *per-flow* and *reactive* manner [141]. That is, every time a switch receives a flow for which it does not have a forwarding entry, the switch queries the SDN controller to get the forwarding rule. Unfortunately, this per-flow and reactive paradigm is fundamentally unsuitable for DDoS defense. First, an adversary can easily saturate the control plane bandwidth as well as the controller compute resources [168]. Second, installing per-flow rules on the switches will quickly exhaust the limited rule space ($\approx$4K TCAM rules). Note

that unlike traffic engineering applications of SDN [115], coarse-grained IP prefix-based forwarding policies would not suffice in the context of DDoS defense, as we cannot predict the IP prefixes of future attack traffic.

C3. Dynamic adversaries: Consider a dynamic adversary who can rapidly change the attack mix (i.e., attack type, volume, and ingress point). This behavior can make the ISP choose between two undesirable choices: (1) wasting compute resources by overprovisioning for attack scenarios that may not ever arrive, (2) not instantiating the required defenses (to save resources), which will let attack traffic reach the customer.

Here we only highlight our key idea to address C2, as the solution to that is directly enabled by FlowTags (to see the ideas to address the other challenges, refer to [95]).

3.5.2 Scalable network orchestration using FlowTags

To address C2, we design a scalable orchestration mechanism using two key ideas. First, switch forwarding rules are based on per-VM tags rather than per-flow to dramatically reduce the size of the forwarding tables. Second, we proactively configure the switches to eliminate frequent interactions between the switches and the control plane [141].

Given the outputs of the resource manager module (i.e., assignment of datacenters to incoming suspicious traffic and assignment of servers to defense VMs), the role of the network orchestration module is to configure the network to implement these decisions. This includes setting up forwarding rules in the ISP backbone and inside the datacenters. The main requirement is scalability in the presence of attack traffic. In this section, we present our *tag-based* and *proactive* forwarding approach to address the limitations of the per-flow and reactive SDN approach.

The canonical SDN view of setting up switch forwarding rules in a per-flow and reactive manner is not suitable in the presence of DDoS attacks. Furthermore, there are practical scalability and deployability concerns with using SDN in ISP backbones [54, 82]. There are two main ideas in our approach to address these limitations:

- Following the hierarchical decomposition in resource management, we also decompose the network orchestration problem into two-sub-problems: (1) Wide-area routing to get traffic to datacenters, and (2) Intra-datacenter routing to get traffic to the right VM in-

stances. This decomposition allows us to use different network-layer techniques; e.g., SDN is more suitable inside the datacenter while traditional MPLS-style routing is better suited for wide-area routing.

- Instead of the controller reacting to each flow arrival, we *proactively* install forwarding rules before traffic arrives. Since we do not know the specific IP-level suspicious flows that will arrive in the future, we use logical *tag-based* forwarding rules with per-VM tags instead of per-flow rules.

Wide-area orchestration: The Bohatei global controller sets up forwarding rules on backbone routers so that traffic detected as suspicious is steered from edge PoPs to datacenters (the specific type and amount of traffic to transfer from each edge PoP to each datacenter is determined by a resource management algorithm that we do not discuss here— see [95] for details).

To avoid a forklift upgrade of the ISP backbone and enable an immediate adoption of Bohatei, we use traditional tunneling mechanisms in the backbone (e.g., MPLS or IP tunneling). We proactively set up static tunnels from each edge PoP to each datacenter. Once the global controller has solved the DSP problem, the controller configures backbone routers to split the traffic according to the $f_{e,a,d}$ values. While our design is not tied to any specific tunneling scheme, the widespread use of MPLS and IP tunneling make them natural candidates [115].

Intra-datacenter orchestration: Inside each datacenter, the traffic needs to be steered through the intended sequence of VMs. There are two main considerations here:

1. The next VM a packet needs to be sent to depends on the *context* of the current VM. For example, the node *check UDP count of src* in the graph shown in Figure 3-13 may send traffic to either *forward to customer* or *log* depending on its analysis outcome.

2. With elastic scaling, we may instantiate several physical VMs for each logical node depending on the demand. Conceptually, we need a "load balancer" at every level of our annotated graph to distribute traffic across different VM instances of a given logical node.

Note that we can trivially address both requirements using a per-flow and reactive so-

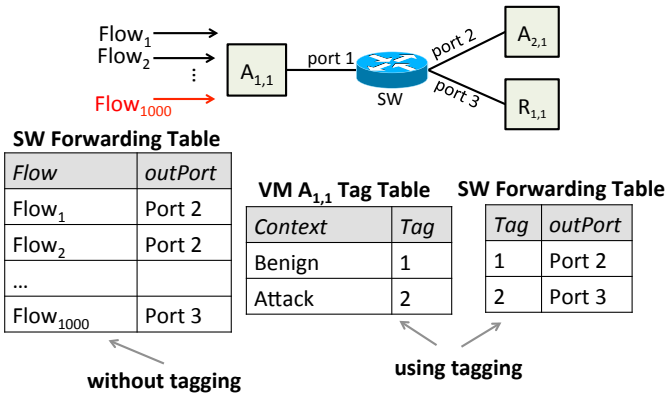


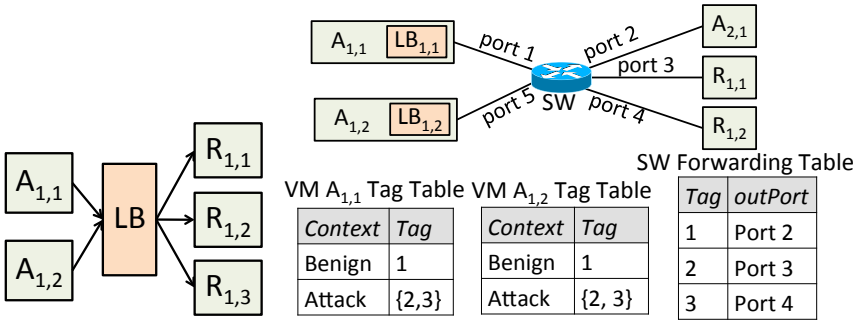
Figure 3-14: Context-dependent forwarding using tags.

lution. Specifically, a local controller can track a packet as it traverses the physical graph, obtain the relevant context information from each VM, and determine the next VM to route the traffic to. However, this approach is clearly not scalable and can introduce avenues for new attacks. The challenge here is to meet these requirements without incurring the overhead of this per-flow and reactive approach.

Encoding processing context: Instead of having the controller track the context, our high-level idea is to encode the necessary context as *tags* inside packet headers [92]. Consider the example shown in Figure 3-14 composed of VMs $A_{1,1}$, $A_{2,1}$, and $R_{1,1}$. $A_{1,1}$ encodes the processing context of outgoing traffic as tag values embedded in its outgoing packets (i.e., tag values 1 and 2 denote benign and attack traffic, respectively). The switch then uses this tag value to forward each packet to the correct next VM.

Tag-based forwarding addresses the control channel bottleneck and switch rule explosion. First, the tag generation and tag-based forwarding behavior of each VM and switch is configured proactively once the local controller has solved the SSP. We proactively assign a tag for each VM and populate forwarding rules before flows arrive; e.g., in Figure 3-14, the tag table of $A_{1,1}$ and the forwarding table of the router have been already populated as shown. Second, this reduces router forwarding rules as illustrated in Figure 3-14. Without tagging, there will be one rule for each of the 1000 flows. Using tag-based forwarding, we achieve the same forwarding behavior using only two forwarding rules.

Scale-out load balancing: One could interconnect VMs of the same physical graph as shown in Figure 3-15a using a dedicated load balancer (load balancer). However, such a load balancer may itself become a bottleneck, as it is on the path of every packet from any



(a) A naive load balancer design.

(b) A distributed load balancer design.

Figure 3-15: Different load balancer design points.

VM in the set $\{A_{1,1}, A_{1,2}\}$ to any VM in the set $\{R_{1,1}, R_{1,2}, , R_{1,3}\}$. To circumvent this problem, we implement the distribution strategy *inside each VM* so that the load balancer capability scales proportional to the current number of VMs. Consider the example shown in Figure 3-15b where due to an increase in attack traffic volume we have added one more VM of type A_1 (denoted by $A_{1,2}$) and one more VM of type R_1 (denoted by $R_{1,2}$). To load balance traffic between the two VMs of type R_1, the load balancer of A_1 VMs (shown as $LB_{1,1}$ and $LB_{1,2}$ in the figure) pick a tag value from a *tag pool* (shown by {2,3} in the figure) based on the processing context of the outgoing packet and the intended load balancing scheme (e.g., uniformly at random to distribute load equally). Note that this tag pool is pre-populated by the local controller (given the defense library and the output of the resource manager module). This scheme, thus, satisfies the load balancing requirement in a scalable manner.

Other issues: There are two remaining practical issues:

- *Number of tag bits:* We give a simple upper bound on the required number of bits to encode tags. First, to support context-dependent forwarding out of a VM with k relevant contexts, we need k distinct tag values. Second. to support load balancing among l VMs of the same logical type, each VM needs to be populated with a tag pool including l tags. Thus, at each VM we need at most $k \times l$ distinct tag values. Therefore, an upper bound on the total number of unique tag values is $k_{max} \times l_{max} \times \sum_a \mid V_a^{annotated} \mid$, where k_{max} and l_{max} are the maximum number of contexts and VMs of the same type in a graph, and $V_a^{annotated}$ is the set of vertices of annotated graph for attack type a. To

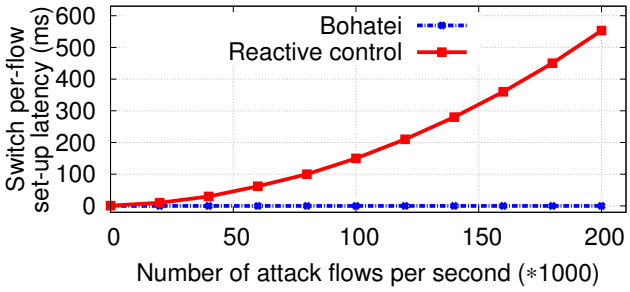


Figure 3-16: Bohatei control plane scalability.

make this concrete, in our experiments (described in [95]) the maximum value required tags was 800, that can be encoded in $\lceil log_2(800) \rceil = 10$ bits. In practice, this tag space requirement of Bohatei can be easily satisfied given that datacenter grade networking platforms already have extensible header fields [128].

- *Bidirectional processing:* Some logical modules may have bidirectional semantics. For example, in case of a DNS amplification attack, request and response traffic must be processed by the same VM. (In other cases, such as the UDP flood attack, bidirectionality is not required.). To enforce bidirectionality, ISP edge switches use tag values of outgoing traffic so that when the corresponding incoming traffic comes back, edge switches sends it to the datacenter within which the VM that processed the outgoing traffic is located. Within the datacenter, using this tag value, the traffic is steered to the VM.

Next we evaluate Bohatei to show the scalability benefit of using FlowTags as the orchestration mechanism (to see details of the evaluation set up and more results, please refer to [95]).

Control plane responsiveness: Figure 3-16 shows the per-flow setup latency comparing Bohatei to the SDN per-flow and reactive paradigm as the number of attack flows in a DNS amplification attack increases. (The results are consistent for other types of attacks and are not shown for brevity.) In both cases, we have a dedicated machine for the controller with 8 2.8GHz cores and 64 GB RAM. To put the number of flows in context, 200K flows roughly corresponds to a 1 Gbps attack. Note that a typical upper bound for switch flow set-up time is on the order of a few milliseconds [180]. We see that Bohatei incurs zero rule setup latency, while the reactive approach deteriorates rapidly as the attack volume increases.

Number of forwarding rules: Figure 3-17 shows the maximum number of rules required

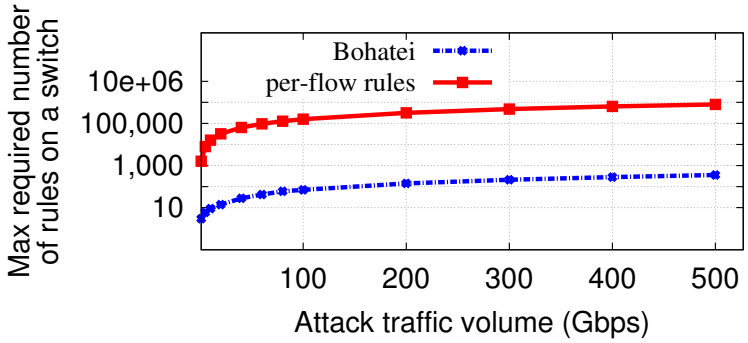


Figure 3-17: Number of switch forwarding rules in Bohatei vs. today's flow-based forwarding.

on a switch across different topologies for the SYN flood attack. Using today's flow-based forwarding, each new flow will require a rule. Using tag-based forwarding, the number of rules depends on the number of VM instances, which reduces the switch rule space by four orders of magnitude. For other attack types, we observed consistent results (not shown). To put this in context, the typical capacity of an SDN switch is 3K-4K rules (shared across various network management tasks). This means that per-flow rules will not suffice for attacks beyond 10Gbps. In contrast, Bohatei can handle hundreds of Gbps of attack traffic; e.g., a 1 Tbps attack will require $<$ 1K rules on a switch.

Benefit of scale-out load balancing: We measured the resources that would be consumed by a dedicated load balancing solution. Across different types of attacks with a fixed rate of 10Gbps, we observed that a dedicated load balancer design requires between 220–300 VMs for load balancing alone. By delegating the load balancing task to the VMs, our design obviates the need for these extra load balancers (not shown).

3.6 Summary

The dynamic, traffic-dependent, and hidden actions of middleboxes make it hard to systematically reason about network policies. We are not alone in recognizing the significance of this problem—others, including the recent IETF network service chaining working group, mirror several of our concerns [119, 138, 159].

The key insight in FlowTags is that the crux of these problems lies in the fact that middlebox actions hide the true source as well as the processing context of packets as they traverse the network. We argue that middleboxes are in the best (and possibly the

only) vantage point to restore these tenets, and make a case for extending middleboxes to provide the necessary context via tags embedded inside packet headers. We design new SDN APIs and controller modules to configure this tag-related behavior. We showed a scalable proof-of-concept controller and the viability of adding FlowTags support with minimal changes to five canonical middleboxes. We also demonstrated that the overhead of FlowTags is comparable to traditional SDN mechanisms. We further demonstrated the utility of FlowTags in enabling scalable policy enforcement in the specific use case of DDoS defense.

We believe that there are three natural directions of future work: first, automating DPG generation via model refinement techniques (e.g., [84]); second, automating middlebox extension using appropriate programming languages techniques [64]; finally, performing holistic testing of the network while accounting for switches and middleboxes. (We will discuss more directions for future work in 5.)

Chapter 4

Reasoning about stateful control planes using ERA

As we saw in §1.1.1 a network is composed of the data and control planes. After discussing our approach to reasoning about the correctness of stateful data planes in the previous chapters, in this chapter, we present our solution for checking the correctness of the network control plane.

The network control plane is in charge of IP routing. Routing involves exchange of routing messages (known as route advertisements or announcements) between routers. Every router maintains a data structure known as the Forwarding Information Base of (FIB) which instructs the router how to forward incoming packets. The router may update its FIB upon receiving a route advertisement from a neighboring router (e.g., to incorporate a better route to a destination IP prefix).

Since a FIB is a dynamically changing data structure, to make sure a reachability policy is enforced correctly, the operator needs to check whether it is enforced correctly not only in the current FIB, but also in the FIBs that may emerge due to interaction of the routers by sending/receiving route advertisements. Many network outages occur due to *latent* routing bugs. These are bugs due to router misconfiguration that might be currently inactive and will be triggered only upon receiving a particular route advertisement. We have seen several recent high-profile routing misconfigurations wreak havoc on the security and performance of critical network services [46, 59]. Going back in history, routing misconfigurations have

created global convergence and reachability issues (e.g., [3, 22]). To see whether router configuration files involve any *latent* bugs, we need the ability to systematically explore the state space of the control plane.

Put another way, reasoning about the control plane is hard, as a real network is in a perpetual churn: route advertisements arrive, links fail, and routers need to be taken offline for maintenance. Nonetheless, an operator needs assurances on the network behaviors because a policy violation may be latent and occur only in a certain future *incarnation* or *state* of the control plane (e.g., a specific route advertisement from a peering network may cause disconnection between A and B [19,46]). Unfortunately, today operators do not have proper tools for efficient reasoning about the network in different environments.

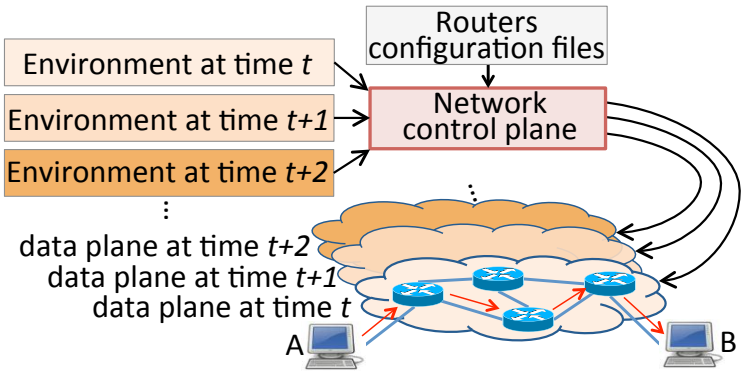


Figure 4-1: Reachability behavior of a network (e.g., A can talk to B) is determined by its data plane, which, in turn, is the current incarnation of the control plane.

To highlight this challenge, it is useful to consider prior work on network verification. A network is composed of a control plane, which configures the behavior of the data plane, which in turn, is in charge of forwarding actual packets (see Figure 4-1). The control plane, therefore, can be thought of as a program that takes configuration files and the current network environment (i.e., route advertisements) and generates a data plane. The data plane is conceptually a program that takes a packet and its location (i.e., a router port) as input and outputs a packet at a different location. As a result, if we rest our analysis on the data plane (e.g., Veriflow [125], HSA [124], NOD [137]) and verify its behavior over its inputs (i.e., packets), we are inherently able to reason about only the current incarnation of the control plane (i.e., the current data plane), and cannot say anything about the network behavior under a different environment.

While there is prior work on bug-finding and verification for the control plane, it suffers from critical limitations. Some tools focus on a single routing protocol (e.g., BGP for Bagpipe [175] and rcc [97]) or a limited set of routing protocol features (e.g., ARC [104]). They can thus not capture the behavior of the entire control plane that often uses multiple routing protocols and sophisticated features [106,131,140]. On the other hand, Batfish [99] analyzes the entire control plane in the context of a given environment, but it does so by simulating the behavior of individual routing protocols to compute the resulting data plane. This simulation is expensive (see §4.4.2), which makes it prohibitive to iteratively use Batfish to analyze the impact of many environments.

What is critically missing today is the ability to efficiently find network reachability bugs across multiple possible environments. (§4.1 motivates this need using real-world examples.) Doing so requires reasoning about network reachability directly at the control plane level, without explicitly computing the data plane that manifests in each environment. Such reasoning is challenging due to the complexity of the control plane, which involves various routing protocols (e.g., BGP, OSPF, RIP) each with its own intricacies (e.g., selecting best route to a destination prefix is different for BGP and OSPF) and cross-protocol interactions (e.g., route redistribution [132]).

We address these challenges in a tool called ERA (Efficient Reachability Analysis) by employing several synergistic ideas [94]. First, we design a unified control plane model that succinctly captures the key behaviors of various routing protocols. In this model, a router is viewed as a function that accepts a route announcement as input and produces a set of route announcements for its neighbors. Second, we use binary decision diagrams (BDDs) [126] to compactly represent the route announcements that constitute a user-specified environment. Third, we shrink the BDD representation of route announcements by identifying equivalence classes of announcements that are treated identically by the given network [178]. Each equivalence class is given an integer index, and the reachability analysis is transformed to arithmetic operations directly on sets of these indices. Consequently, we take advantage of vectorized instruction sets on commodity CPUs for fast computation of these set operations (§4.3.3).

ERA can be used to identify bugs in reachability policies of the form "A can talk to B"

as well as a wide range of common policies that are expressible in terms of reachability relationships, such as valley-free routing and blackhole-freeness (§4.3.4). Our implementation of ERA is available as an open source and extensible toolkit to which new kinds of analysis can be added (§4.3.5).

We evaluate the utility of ERA in a range of real and synthetic scenarios (§4.4.1). Across all scenarios, it successfully finds both new and known reachability violations, which were otherwise hard to find using the state of the art techniques. We also evaluate the scalability of ERA and find that it can handle a network with over 1,600 routers in 6 seconds. Our evaluations show that our control plane modeling and exploration techniques yield significant speedup.

4.1 Motivation

We motivate reasoning about multiple network incarnations using real reachability bugs encountered in a large cloud provider's network. These bugs were latent and manifested only under certain environments.

Maintenance-triggered: Some bugs stem from unexpected interactions of different routing protocols and configuration directives. In this example (Figure 4-2), the interactions are between static routing and BGP. For redundancy, the operator's goal was to have two paths between the DCN (datacenter network) and the WAN (wide area network), one through R_1 and the other through R_2. One day, the operator decided to temporarily bring down R_2 for maintenance, which she thought was safe because of the assumed redundancy. However, as soon as R_2 was brought down, the entire DCN was disconnected from the WAN (and the rest of the Internet).

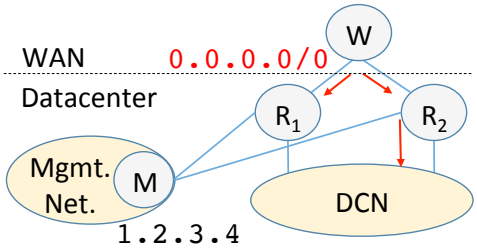


Figure 4-2: A bug triggered by maintenance.

Manual investigation revealed that R_1 contained a static default route

`ip route 0.0.0.0/0 1.2.3.4` (here `1.2.3.4` is the next-hop of the static route, which is the address of the management network). Static routes to a prefix supersede dynamic routes [14, 28]. Thus R_1 preferred the static route over the default BGP route advertised by the WAN (shown in red). Since static routes are typically not propagated to neighbors, R_1 did not advertise the default route to the DCN. Thus, the DCN was entirely dependent on R_2 for external connectivity.

The bug in R_1's configuration was that the operator had forgotten to type keywords to indicate that the static route belonged to the management network, not data network. (These keywords were present in R_2's configuration.) The bug was latent as long as R_2 was up, but was triggered when R_2 was brought down.

Announcement-triggered: In Figure 4-3, DC_A had several services hosted inside the subprefixes of 10.10.0.0/16. Instead of announcing the individual subprefixes, R_1 was announcing this aggregate prefix. DC_B could reach the services inside DC_A through the WAN. As soon as a new service with prefix 10.10.1.160/28 was launched inside DC_A, all other services inside the /16 prefix became unreachable from DC_B.

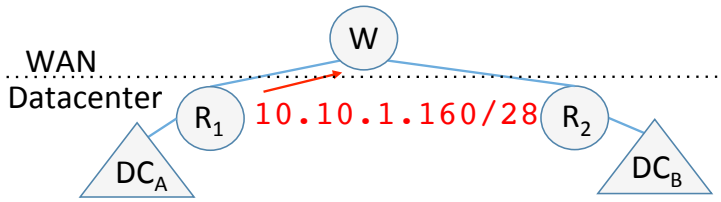


Figure 4-3: A bug triggered by a BGP announcement.

Investigation revealed two latent configuration bugs that combined to create this outage: *(1)* R_1 was not configured to filter 10.10.1.160/28 in its announcements to the WAN; and *(2)* R_2 was configured with an aggregate route to 10.10.0.0/16 with DC_B as the next hop. The result of the first bug was that the /28 announcement reached R_2 through the WAN. Then, as a result of the second bug, the /16 aggregate route was activated at R_2. This aggregate route, as a local route to router R_2, took precedence over the /16 being announced through the WAN. When the aggregate route was activated, R_2 started dropping all traffic to the /16 except for traffic to the /28. These drops are due to the *sinkhole* semantics of route aggregation— the aggregating router drops packets for subprefixes for which it does not

have an active route to prevent routing loops [134].[1] Proper connectivity existed prior to the /28 announcement because the /16 announcement from the WAN did not activate the aggregate route at R_2.

Failure-triggered: In Figure 4-4, R_1 and R_2 were configured to announce prefix 10.10.0.0/16 that aggregated the subprefixes announced by leaf routers (A_1, A_2, A_3). After link A_2—B_2 failed, WAN traffic destined to A_2's prefix (10.10.2.0/24) started getting blackholed (i.e., dropped) at R_1 even though A_2 had connectivity via B_3 and R_2.

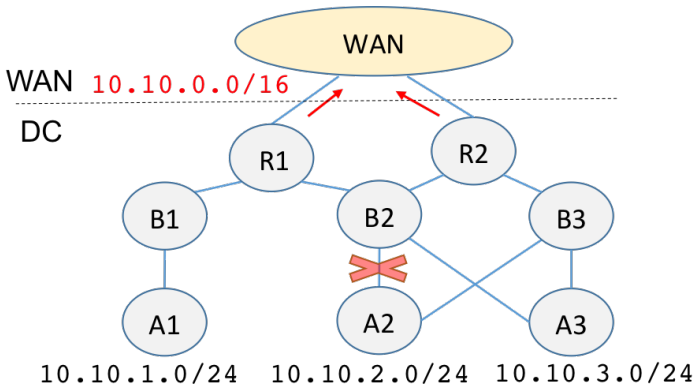


Figure 4-4: A bug triggered by link failure.

This blackhole was created because R_1 continued to make the aggregate announcement after the failure of link A_2–B_2, as it was still hearing announcements for the other two subprefixes corresponding to A_1 and A_3 (aggregate routes are announced as long as there is at least one subprefix present). As a result, the WAN sent (some) traffic for 10.10.2.0/24 toward R_1. But R_1 dropped those packets per the *sinkhole* semantics (see above).

4.2 Related work

We saw the prior work on checking the behavior or the network data plane in §2.2. Checking the data plane alone has the fundamental limitation that a network is in a constant churn, which manifests itself as different data planes. For example, a single route advertisement can dramatically change the network behavior (e.g., see [46]). Moving from the data plane to the control plane potentially enables more powerful analysis, as the former is generated by the latter.

[1]For instance, if W announced the default route to R2, R2 would forward traffic for 10.10.2.2 to W, which may then forward them to R2 (because R2 announces the aggregate /16 to W), and so on.

Prior work is limited due to supporting only a single routing protocol (e.g., BGP in Bagpipe [175] and rcc [97]) or a limited set of routing protocol features (e.g., ARC [104]). Batfish [99] can reason about the entire control plane but its analysis is expensive because it simulates the individual steps of each routing protocol. In contrast, ERA enables fast exploration using a succinct encoding of control plane behavior.

Metarouting [110], glue logic [133], and Propane [73] aim to build a correct-by-design control plane. While worthwhile in the long term, these efforts cannot reason about existing networks.

To summarize, what is critically missing today is the ability to efficiently explore the control plane involving various routing protocols. We illustrate this need below.

4.3 System design

In this section, after presenting our high-level approach and the challenges in realizing it, we present our solution ERA, which is a control plane analysis tool to find latent reachability bugs due to router mis-configuration.

4.3.1 ERA Overview

Our target is a (datacenter, enterprise, or ISP) network of a realistic size (e.g., a few to hundreds of routers). As shown in Figure 4-5, our user is a network operator responsible for configuring routers. The operator has a set of intended reachability policies of the form "Router port A can talk to router port B" (as we will discuss in §4.3.4, several other practical policies are derivatives of "A can talk to B"). ERA allows operators to input their assumptions on what the network's environment will send (e.g., based on relationship with peers/providers). It then analyzes the network's behavior under these assumptions and checks whether the behavior satisfies the intended reachability policies. This process can then be iterated with other environmental assumptions, in order to cover a range of possible environments.

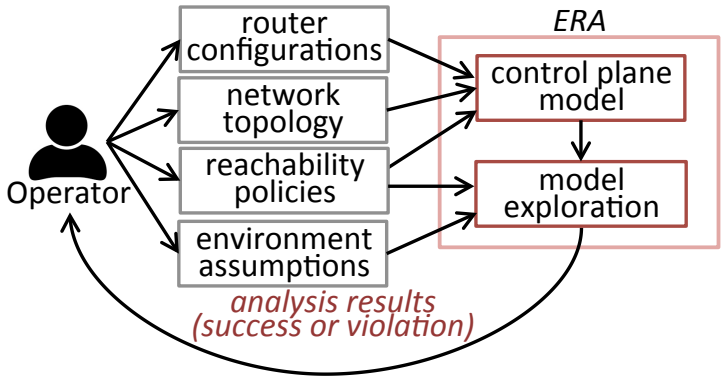


Figure 4-5: High-level vision of ERA.

Here we give the intuition behind our approach to control plane analysis.

Relationship between data and control planes: The data plane takes as input a packet on a router port and moves the (possibly modified) packet to another port (on the same or a neighboring router). Thus, we can think of the data plane as a function of the form $DP : (pkt, port) \rightarrow (pkt, port)$. The data plane itself is generated by the control plane function given routers' configuration files, the network topology (i.e., which router ports are inter-connected), and the current environment (which captures the route advertisements sent to the network by the "outside world") of the network: $CP : (env, Topo, Configs) \rightarrow DP(.)$.

Reachability policies via control plane analysis: Since packets are forwarded by the data plane, it is natural to think of an intended reachability policy $\phi_{A \rightarrow B}$ as a predicate that indicates whether a given packet should be able to reach from router port A to router port B. We say data plane DP is policy-compliant if $\phi_{A \rightarrow B}(pkt, DP)$ evaluates to $true$ for all A-to-B packets.

A seemingly natural approach for finding latent bugs is to produce the data plane associated with a given environment and then check reachability on that data plane [99]. However, this approach makes it prohibitively expensive to iteratively check multiple environments (§4.4.2). This is because for each possible environment (of which there are many), to compute the resulting data plane, we need to account for all low-level message passings and nuances of routing protocols. Instead, we want to be able to reason about the network directly at the level of the control plane and without explicitly computing the data plane.

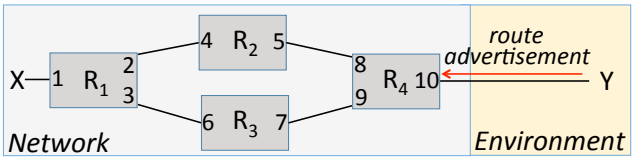


Figure 4-6: X-to-Y reachability depends on routers configurations and the environment.

To this end, our insight is as follows. Rather than producing the data plane that results from a given environment, we can analyze the control plane under that environment to determine i) the routes that each router in the network learns via its neighbors (e.g., a BGP advertisement) or its configuration file (e.g., static routes); and ii) the best route when multiple routes to the same prefix are learned. We can then use this information to directly check reachability.

An illustrative example: To visualize what it means to reason about reachability using control plane analysis, consider the example shown in Figure 4-6. Here we want to see what traffic reaches from port X to port Y so that we can check whether it is policy-compliant. From the figure we can see that to find the above traffic, we can try to find the routes that traverse the opposite direction on each of the two paths. Let $T^{i \to j}_{Router}(route)$ show the output of the configured router $Router$ on its port j given the input $route$ on its port i. (Intuitively, $route$ can be thought of as an abstraction for a route advertisement. The following section will elaborate on this abstraction.) If we knew $T(.)$, the answer would be: $T^{2 \to 1}_{R_1}(T^{5 \to 4}_{R_2}(T^{10 \to 8}_{R_4}(env))) \cup T^{3 \to 1}_{R_1}(T^{7 \to 6}_{R_3}(T^{10 \to 9}_{R_4}(env)))$. The argument env here represents the assumptions that the user makes about the environment.

Challenges: Control plane-based reachability analysis requires us to address two key challenges:

- **An expressive and tractable control plane model:** To be expressive, this model needs to capture key behaviors of diverse protocols (e.g., BGP, OSPF route advertisements). A naive model (e.g., capturing protocol-specific behaviors verbatim), while expressive, is impractical because it will be too complex to explore. On the other extreme, a very high-level model (e.g., ignoring protocol-specific behaviors altogether) may be tractable to explore, but not expressive (e.g., BGP and OSPF have different ways of preferring routes).

- **Scalable control plane exploration:** Once we have a control plane model, we need the ability to efficiently explore the model with respect to the environment, in order to identify violations of intended reachability policies.

We tackle these challenges in §4.3.2 and §4.3.3, respectively.

Scope and Limitations: ERA's analysis requires the user to provide assumptions on the environment (or defaults to assuming that the environment makes all possible route announcements). If these assumptions are incorrect or overly permissive, then ERA can produce false positives, identifying purported errors that in fact will never arise in practice; e.g., a reputable ISP is not likely to hijack its peer's traffic. ERA is designed to have no other source of false positives (i.e., its control plane model is accurate). Though we have not formally proven this yet, empirically speaking, all the bugs that ERA has identified were real bugs.

ERA also has several sources of false negatives. First, ERA will only find bugs under environments specified as inputs and cannot guarantee the absence of bugs under all environments (unless exhaustively iterated on all possible environments). Second, certain classes of errors cannot be found by ERA by design. Specifically, ERA assumes that routing will converge and only analyzes this convergent state, which is key to efficient exploration of the control plane. Therefore, convergence errors as well as reachability errors in transient states of the network will not be found (e.g., [109, 111]).

Finally, while ERA supports most of the common configuration directives, our current implementation does not support certain directives such as regular expressions in routing filters. Keeping up with configuration directives is a software engineering challenge due to their large and growing universe. Such limitations, however, are not fundamental to the design of ERA (unlike ARC [104], where the design itself cannot handle certain routing features).

As we will see in §4.4, ERA can find a large class of real-world bugs despite these limitations.

4.3.2 Modeling the control plane

We now describe our model for the network control plane. It $i)$ captures all routing protocols using a common abstraction; $ii)$ is expressive with respect to routing behaviors of individual protocols; and $iii)$ lends itself to scalable exploration. At a high level, we identify key behaviors of the control plane (e.g., route selection, route aggregation) and compactly encode them using binary decision diagrams (BDDs) [126].

Since the network control plane is a composition of the control planes of individual routers, we break down the problem of modeling the network control plane into modeling *(i)* the I/O unit of a router's control plane, and *(ii)* the processing logic of a router's control plane.

Route as the Model of Control Plane I/O: A naive way of modeling the I/O unit of the control plane of a router is to use the actual specification of route advertisements of different routing protocols, including their low-level details (e.g., keep-alive messages, sequence numbers [9, 39]). While expressive, such an I/O unit makes the control plane model too cumbersome. Conversely, if we completely ignore differences across protocols to simplify our I/O unit model, such a model may not be expressive; e.g., it cannot capture the fact that if a router learns two routes to the same destination prefix from two different routing protocols, the one offered by the protocol that has a smaller administrative distance (AD) will be selected [14, 28]. (We will see an example bug scenario due to this effect in §4.4.1, Figure 4-18b.)

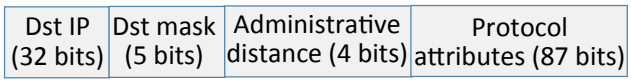


Figure 4-7: *route* as the model of control plane I/O.

To strike a balance between expressiveness and tractability, we introduce the notion of an abstract *route* as a succinct yet expressive I/O unit for the control plane model. Conceptually, a route mimics a route advertisement. It is a succinct bit-vector conveying key information in route advertisements that affects routing decisions of a router (see Figure 4-7). While not fundamental to our design, we have chosen a 128-bit vector to encode a route to enable fast CPU operations as we will discuss in §4.4.2. To accommodate diverse routing

protocols, a route unifies key attributes of various protocols that affect a router's behaviors (i.e., administrative distance and protocol-specific route attributes).[2] To improve scalability, a route abstracts away the low-level nuances of actual protocols (e.g., seq. numbers, acknowledgements).

The fields of our route abstraction are:

- *Destination IP and mask*: Together, they represent the destination prefix that the route advertises. To make a route compact, we store the mask in 5 bits (instead of its naive storage in 32 bits). To make this concrete, let $dstIP$ and $dstMask$ denote a 32-bit destination IP address and our 5-bit encoding of the destination mask. To compute the destination prefix that the destination IP and mask represent, we first transform the mask to its customary 32-bit representation (e.g., 255.255.0.0), and then AND it with the IP address:
  $dstPrefix \leftarrow dstIP \& ((2^{32} - 1) << (32 - dstMask))$
  where $<<$denotes the shift left operator.
- *Administrative distance (AD)*: This is a numerical representation of the routing protocol (e.g., BGP, OSPF) of the route such that $AD_A < AD_B$ denotes routing protocol A is preferred to protocol B.
- *Protocol attributes*: This captures protocol-specific attributes of the routing protocol represented by AD. For example, if the value of AD corresponds to BGP, the protocol attributes field encodes the BGP attributes (i.e., weight, local preference). To enable fast implementation of route selection in our router model (that we will discuss next), we carefully encode the attributes so that preferring a route between two routes $route_1$ and $route_2$ simply becomes a matter of choosing the smaller of two bit-vectors $AD_1.attrs_1$ and $AD_2.attrs_2$ when interpreted as unsigned integers (the symbol . denotes concatenation of the AD and protocol attributes fields of a route). For example, since route selection in BGP involves checking a prioritized list of BGP attributes (e.g., first checking the weight, then local preference, etc.) [11], for a BGP route, the highest order bits of the protocol attributes field of the route encode the complement of the BGP weight attribute, followed by the complement of the local preference, and so forth. Note that

[2]Since our route model resembles routing messages in distance-vector protocols, we accommodate link state protocols (e.g., OSPF) by letting the attributes refer to the routes output by the Dijkstra algorithm.

the designated 87 bits for succinctly capturing protocol attributes have been sufficient in a range of realistic scenarios we have considered (§4.4), but there might be scenarios where more bits are needed to encode many distinct attributes.

Control plane as a visibility function: Given the I/O unit of the control plane, next we need to model the processing logic that a router applies to input routes. Intuitively the router model is a function that given a route as its input, computes the corresponding output route(s). We identify 5 key operations of the router control plane: *(i) Input filtering*, which modifies/drops incoming route advertisements to the router; *(ii) Route redistribution*, which is necessary to capture cross-protocol interactions [131,133]; *(iii) Route aggregation*, which is a common mechanism to shrink forwarding tables, yet its improper use can lead to reachability violations [134]; *(iv) Route selection*, which is in charge of selecting the best route to a given destination prefix; and *(v) Output filtering*, which modifies/drops outgoing route advertisements.

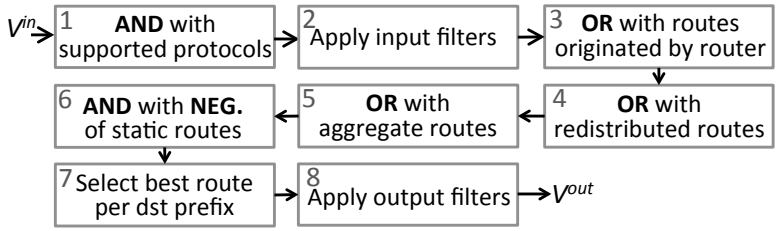


Figure 4-8: High-level router model processing boolean representation of input routes.

Unfortunately, reasoning about the control plane one routing announcement at a time is not scalable. Instead, we *lift* our router model to work simultaneously on a *set* of route announcements. We refer to our router model as the *visibility function* because it captures how the router control plane processes the routing information made visible (i.e., given as input) to it. The input to the router visibility function, V^{in}, is the set of input routes sent by its neighbors and configured static routes; and its output, V^{out}, is the set of corresponding output routes that are sent downstream by the router. The notation $V^{out}_{Router} = T_{Router}(V^{in}_{Router})$ denotes the control plane visibility function of $Router$.

For fast exploration, we use BDDs to symbolically encode the set of I/O routes in a router model. A BDD is a compressed representation of a boolean function that enables fast implementation of operations such as conjunction, disjunction, and negation [126].

Our BDD encoding enables fast router operations by transforming operations on sets to quick operations on BDDs. For example, taking the complement of a set simply requires flipping the true/false leaves of the corresponding BDD.

Figure 4-8 shows the high-level procedure for processing a boolean representation of sets of routes. The steps to turn V^{in} into V^{out} are as follows:

1. *Supported protocols:* First, the routing protocols present in the configuration file are accounted for.
2. *Input filtering*: Then, the input filters are applied.
3. *Originated routes*: In addition to the input route, there are routes that directly stem from the configuration files, which are conceptually ORed with the input.
4. *Route redistribution*: A route redistribution command propagates routing information from a routing protocol (e.g., BGP) into another protocol (e.g., OSPF).
5. *Route aggregation*: If the router receives any input route that is more specific than any configured aggregate route, the aggregate route gets activated.
6. *Static routes*: A static route is a route locally known to the router (i.e., not shared with its neighbors). Further, by default, static routes take precedence over dynamic routes (e.g., OSPF, BGP, RIP, IS-IS) due to having a lower AD value. This behavior is captured by ANDing the negation of static routes with all other routes.
7. *Route selection*: Selecting the best of multiple routes to a destination prefix works as follows: *(i)* if the routes belong to different routing protocols, the one with the lowest AD value is selected, *(ii)* if the routes belong to the same routing protocol, the protocol-specific attributes determine the winner.
8. *Output filtering*: The router applies its output filters.

For completeness, the pseudocode for this process in Figure 4-9. The pseudocode shows the details of the above router control plane model. The pseudocode describes how a configured router turns the boolean representation of its input routes to output routes. Note that we account for per-port (i.e., router interface) behaviors because, in general, a router can have distinct routing behaviors configured on its different ports.)

1 ▷ Inputs: (1) Configuration information pertaining to router output port $Router_{port}$ including: static routes $sr[.]$, route redistribution $rr[.]$, route aggregation $ra[.]$, supported routing protocols $proto[.]$, input filters $if[.]$, output filters $of[.]$ (2) Input to the router is a boolean function in DNF form: $V^{in} = X_1^{in} \vee \cdots \vee X_N^{in}$

2 ▷ Output: Boolean representation of $Router_{port}$ in DNF

3 ▷ Route bit vector from Figure 4-7, denoted by X, is concatenation of 3 fields: $X = X_{prefix}.X_{proto}.X_{attr}$

4 $V^{out} = V^{out} \wedge \{\bigvee_i X_{proto[i]}\}$ ▷ Applying supported routing protocols

5 ▷ Applying input filters

6 **for** $i = 1$ to $size(if[.])$

7 **for each** disjunctive term of V^{out}, denoted by V_j^{out}

8 **if** V_j^{out} matches $if[i].condition$

9 apply action $if[i].action$

10 $V^{out} = V^{out} \vee V^{local}$ ▷ Accounting for routes that $Router$ originates, denoted by V^{local}

11 **for** $i = 1$ to $size(rr[.])$ ▷ Applying route redistribution

12 **for each** disjunctive term of V^{out}, denoted by V_j^{out}

13 **if** $V_j^{out}.X_{proto} == rr[i].fromProto$

14 $newTerm = V_j^{out}$

15 $newTerm.X_{proto} = rr[i].toProto$

16 $newTerm.X_{attr} = defaultAttr[proto]$

17 $V^{out} = V^{out} \vee newTerm$

18 **for** $i = 1$ to $size(ra[.])$ ▷ Applying route aggregation

19 $newTerm.X_{prefix} = ra[i].prefix$

20 $newTerm.X_{proto} = ra[i].proto$

21 $newTerm.X_{attr} = defaultAttr[proto]$

22 $V^{out} = V^{out} \vee newTerm$

23 **for** $i = 1$ to $size(sr[.])$ ▷ Applying static routes

24 **for each** disjunctive term of V^{out}, denoted by V_j^{out}

25 **if** $AD(V_j^{out}.X_{proto}) > AD(static)$

26 $V_j^{out} = V_j^{out} \wedge (\overline{sr[i].prefix})$

27 **for each** prefix $prfx$ present in V^{out} ▷ Applying route selection

28 $precedence = +\infty$

29 **for each** disjunctive term of V^{out}, denoted by V_j^{out}

30 **if** $(V_j^{out}.prefix == prfx)\&\&(V_j^{out}.AD.attr < precedence)$

31 $precedence = V_j^{out}.AD.attr$ ▷ Finding best route

32 **for each** disjunctive term of V^{out}, denoted by V_j^{out}

33 **if** $(V_j^{out}.prefix == prfx)\&\&(V_j^{out}.AD.attr > precedence)$

34 Eliminate V_j^{out} from V^{out} ▷ Eliminating others

35 $V_j^{out} = V_j^{out} \vee prfx.precedence$

36 **for** $i = 1$ to $size(of[.])$ ▷ Applying output filters

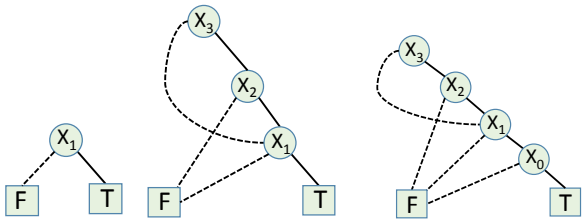
37 **for each** disjunctive term of V^{out}, denoted by V_j^{out}

38 **if** V_j^{out} matches $of[i].condition$

39 apply action $of[i].action$

40 **return** V^{out}

Figure 4-9: Route control plane visibility function.



(a) RIP. (b) Static route. (c) Output filter.

Figure 4-10: Example router model as a BDD. Dashed and solid lines represent the values 0 and 1 of the corresponding binary variable, respectively.

An illustrative example: We illustrate the procedure of Figure 4-8 using a small example. For ease of presentation, a route here has only 4 bits $x_3x_2x_1x_0$, with two bits x_3x_2 representing IP prefix, the bit x_1 representing AD, and the bit x_0 representing protocol attributes. A bar over a binary variable denotes its negation. In this example, the network operator assumes the router accepts *all* routes as input, which is captured by setting $V^{in} = 1$ (i.e., $true$).

Suppose a router is configured with a static route and RIP, with AD values of 0 and 1, respectively. Figure 4-10 shows the BDD representation of the router that has the following four (simplified) configuration commands:

- `RIP`, denoting the presence of RIP on the router, is captured by $1 \wedge x_1 = x_1$, as shown in Figure 4-10a.
- `static 10/2`: Since this static route overrides the RIP routes with the same prefix, the resulting predicate is $(\overline{x_3}\,\overline{x_2})x_1 = \overline{x_3}x_1 \vee x_2x_1$. This is shown in Figure 4-10b.
- `output filter: if RIP attribute is 0, make it 1`: The effect of the filter is to replace all occurrences of x_1 by x_1x_0. The resulting predicate is $\overline{x_3}x_1x_0 \vee x_2x_1x_0$. This is captured in Figure 4-10c.

Intuitively, the output $V^{out} = \overline{x_3}x_1x_0 \vee x_2x_1x_0$, simplified to $V^{out} = (\overline{x_3} \vee x_2) \wedge x_1x_0$, represents the fact that given $every$ environment as the input, the router outputs RIP (noted by x_1) with attribute 1 (noted by x_0) and the dest. prefix can be 00, 01, or 11 (noted by $\overline{x_3} \vee x_2$).

In the following section, we will discuss how to reason about the reachability behaviors of the network by exploring the router model we developed in this section.

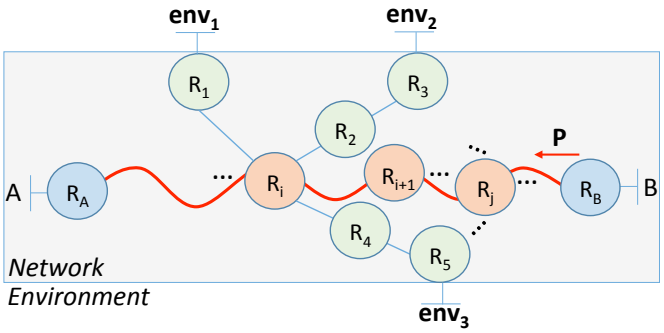


Figure 4-11: Computing A to B reachability.

4.3.3 Exploring the control plane model

Our reachability analysis is based on an exploration of the control plane model above. We first describe this exploration, and then describe how we leverage our BDD-based encoding to devise a set of scalable exploration mechanisms that use *(i)* the Karnaugh map, *(ii)* equivalence classes, and *(iii)* vectorized CPU instructions.

Exploration method: We present our approach to finding traffic reachable from port A to port B using a representative example. Consider the scenario shown in Figure 4-11. The red path is an A-to-B path involving routers $R_A, \ldots, R_i, R_{i+1}, \ldots, R_B$. For ease of presentation, in this example, there is only one path from A to B.

To see the effect of the environment, consider router R_i, which has three paths to router ports that face the outside world (namely, outside facing ports of routers R_1, R_3, and R_5). Unless the operator makes a more specific assumption on an environment input (i.e., what route advertisements the outside world will send to the network), ERA starts analysis using the boolean value $true$ (represented by a BDD with only one leaf with the value $true$), which represents the fact that $every$ possible route are provided by the environment. On the other hand, if the operator is able to make a more scoped assumption about the environment (e.g., based on expected routes from a neighbor), the starting environment will reflect the assumption. Such assumptions can be encoded as a BDD that explicitly includes the relevant variables on the assumed prefix, administrative distance, or attributes values of incoming routes from the environment.

Computing traffic reachable from A to B involves the following steps:

1. *Applying the effect of the environment:* Every router on a A-to-B path that has a topology path to the environment, is affected by it. For router R_i in our examples, it means R_i

receives the environment input E_i^{in}, where

$$E_i^{in} = T_1(env_1) \vee T_2(T_3(env_2)) \vee T_4(T_5(env_3))$$

2. *Computing routes reachable from B to A:* As we saw in §4.3.1, the key to computing traffic prefixes that reach from A to B using control plane analysis is to compute what route prefixes are made visible from B back to A. Let $assumed_B$ show the input the operator assumes about what port B receives from the environment. For the red path, this is captured by

$$reach_{A \rightsquigarrow B} = T_A(E_A^{in} \vee \ldots (T_{i+1}(E_{i+1}^{in} \vee \ldots T_N(E_B^{in} \vee assumed_B) \ldots)))$$

3. *Extracting prefixes reachable from A to B:* Since we are interested in route prefixes reachable from B to A, we eliminate binary variables in the route fields that do not correspond to prefix (i.e., AD and protocol attributes) in all boolean terms of $reach_{A \rightsquigarrow B}$.

4. *Accounting for on-path static routes:* In addition to the routes that reach from B to A, which cause traffic to reach from A to B, there is potentially other traffic that can reach from A to B due to static routes configured on on-path routers. This is because while a router does not advertise its static routes, activated static routes end up in its forwarding table. We account for such prefixes and OR them with the answer from step 3.

5. *Applying ACL rules affecting A-to-B traffic:* While a router configuration file primarily includes directives to configure the router control plane, it may include access control lists (ACLs) that restrict the actual traffic that can pass through the data plane of the router. We, therefore, account for ACLs by taking the result of step 4 and applying the ACLs of the on-path routers.

Once traffic prefixes reachable from A to B are computed, the network is policy-compliant if the prefixes are equal to $\phi_{A \rightarrow B}$ from §4.3.1. If ϕ is violated, ERA applies the Karnaugh map [113] to the DNF representation of the violating routes to provide the human operator with fewer distinct items to investigate; e.g., instead of reporting distinct

prefixes $10.20.0.0/17$ and $10.20.128.0/17$ as violations, ERA summarizes and outputs them as $10.20.0.0/16$.

The process above finds policy violations in the context of a single set of environmental assumptions. The user can iterate multiple times with different assumptions in order to expose more errors. Conceptually, each iteration of ERA over a BDD input analyzes a *set* of concrete environments for which the network has an identical behavior. The analysis implicitly identifies this set during exploration, by accumulating constraints from the visibility function of each router in the network. Thus, the number of iterations needed for exhaustive exploration using ERA is far less than those needed with data plane based analysis tools such as Batfish.

For completeness, the pseudocode presented in Figure 4-12 shows how ERA computes traffic prefixes reachable from A to B considering all A-to-B paths.

Scalability Optimizations: To build an interactive tool for network operators, we want ERA to be able to compute $A - to - B$ reachability in no more than a few seconds. Even with the tractable control plane model that we developed in §4.3.2, a naive implementation of the exploration mechanism fails to satisfy our goal. This is because of the very large range of possible environments. Here we present three techniques to scale control plane exploration.

Minimizing collection of routes with the K-map: As a first step, to minimize the binary representation of the router I/O, we apply the Karnaugh map (K-map), which is a common technique in circuit design [113].

Finding equivalence classes: Performing computations (e.g., conjunction and disjunction) on boolean representation of a real control plane is cumbersome. For example, the same or similar destination prefixes may appear on multiple routers. As such, if we encode prefixes naively, this may slow down control plane exploration.

Given this observation, before performing reachability analysis, ERA gets rid of redundant data by finding equivalence classes of routes which are treated identically by the network, using which the data can be rebuilt [178]. The advantage of doing so is that now performing disjunction and conjunction on boolean terms boils down to doing union and intersection on sets of integers (known as atomic predicates [178]). These integers are the

1 ▷ Inputs: (1) router-level topology of network
(2) Set of router ports facing environment Env
(3) routers configurations
2 ▷ Output: Prefix(es) of traffic reaching from router port A to router port B

3 Parse router configurations into boolean functions (using Figure 4-9)
4 Initialize $assumed_e$ on port e (by default, $true$)
5 initialize $assumed_B$ on port B (by default, $true$)

6 ▷ Accounting for effect of environment on routers on an A-to-B path
7 **for each** router $router_i$ on an $A - to - B$ path
8 **for each** environment-facing port $e \in Env$
9 **for each** path p from port e to $router_i$
10 ▷ $router_j$ is the jth router on $e \rightsquigarrow i$, where $1 \leq j \leq M(j)$
11 $E^{in}_{e \rightarrow i,p} = E^{in}_{e \rightarrow i,p} \vee T_{M(j)}(\ldots(T_1(assumed_e))\ldots)$
12 $E^{in}_{e \rightarrow i} = E^{in}_{e \rightarrow i} \vee V^{in}_{e \rightarrow i,p}$
13 $E^{in}_i = E^{in}_i \vee E^{in}_{e \rightarrow i}$

14 ▷ Compute per-path reachability
15 Find all paths from B to A in G: $Path_{B \rightsquigarrow A} = \{path^1_{B \rightsquigarrow A}, \ldots, path^N_{B \rightsquigarrow A}\}$
16 ▷ $router^j_i$ is the jth router on $path^i_{B \rightsquigarrow A}$, where $1 \leq j \leq M(j)$
17 $reachability^{path^i_{B \rightsquigarrow A}}_{B \rightsquigarrow A} = T_{M(j)}(\ldots(T_2(E^{in}_2 \vee (T_1(E^{in}_1 \vee assumed_B))))$

18 Eliminate binary variables in $reachability_{A \rightsquigarrow B}$ except those corresponding to X_{prefix}

19 ▷ Accounting for static routes
20 $static_{A \rightsquigarrow B} = \bigvee_i (\bigwedge_k (StaticPrefix_{Router^k_i}))$
21 $reachability_{A \rightsquigarrow B} = reachability_{A \rightsquigarrow B} \vee static_{A \rightsquigarrow B}$

22 ▷ Accounting for on-path ACLs. $Router^k_i$ is the kth router on $path^i_{A \rightsquigarrow B}$
23 $reachability^{path^i_{B \rightsquigarrow A}}_{B \rightsquigarrow A} = reachability^{path^i_{B \rightsquigarrow A}}_{B \rightsquigarrow A} \wedge (\bigvee_k ACLs_{Router^k_i})$

24 ▷ Compute all paths reachability
25 $reachability_{A \rightsquigarrow B} = \bigvee_i reachability^{path^i_{B \rightsquigarrow A}}_{A \rightsquigarrow B}$

26 **return** $reachability_{A \rightsquigarrow B}$

Figure 4-12: Computing A-to-B reachability.

indices of the equivalence classes. We illustrate this technique using an example. Suppose we need to compute the conjunction of the boolean terms X, Y, and Z (e.g., representing three routes). Instead of naively computing the conjunction on the raw boolean form of these terms, we do the following:

1. Express each term in terms of equivalence classes as depicted in Figure 4-13; e.g., $X = a_2 \vee a_5 \vee a_6 \vee a_7$.

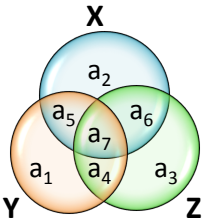
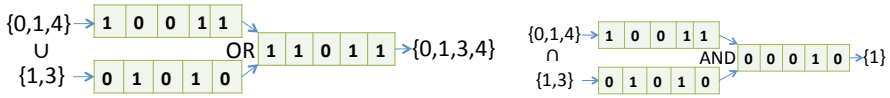


Figure 4-13: Visualization of predicates X, Y, and Z in terms of members of equivalence classes $a_1, \ldots, a_7$.

2. Represent each term using the indices of members of equivalence classes, e.g., X is the union of members 2, 5, 6, and 7. (This way, irrespective of how bulky the raw form of term a_i might be, it is represented by integer value i.)

3. To compute $X \wedge Y \wedge Z$, intersect the sets of their corresponding indices: $\{1, 5, 6, 7\} \cap \{1, 4, 5, 7\} \cap \{3, 4, 6, 7\} = \{7\}$, which indicates the answer to $X \wedge Y \wedge Z$ is a_7.

Implementing fast set operations: As we saw above, using equivalence classes, reachability analysis involves computing union and intersection of sets of integers. We leverage vectorized instructions on recent processors to perform fast set union and intersection of two sets of integers (i.e., the indices of the equivalence classes). The intuition is simple: if a set of integers is represented as a bit vector where each bit represents the presence/absence of the corresponding value, then the union (intersection) of two sets of integers is the bit-wise OR (AND) of the two bit vectors.



(a) Set union using OR. (b) Set intersection using AND.

Figure 4-14: Fast ∪ and ∩ of two sets of integers.

Figure 4-14 shows this approach using an example. In our implementation, we use instructions on 256-bit vectors in our Intel AVX2 implementation [52].

4.3.4 Going beyond reachability

Building on basic A-to-B reachability, ERA can be used to check a wider range of policies. In §4.4, we will discuss scenarios involving these policies.

Valley-free routing: Operators often want to implement "valley-free" routing [102],

which means that traffic from a neighboring peer or provider must not reach another such neighbor. This condition is a form of reachability policy that ERA can easily check.

Equivalence of two routers: Operators often use multiple routers to provide identical connectivity for fault tolerance. Checking if they are identically configured (e.g., using configuration syntax) is hard because the routers may be from different vendors and many aspects of the configuration (e.g., interface IP addresses) can legitimately differ across routers of even the same vendor. To check semantic equivalence of two routers' policies, we use the following property of BDDs: if two boolean functions defined over n boolean variables are equivalent (i.e., they generate the same output for the same input), their Reduced Ordered BDDs (ROBDDs) are identical [76]. In our implementation, we check the equality of the adjacency matrix representations of the BDDs of the two functions, which takes $O(n^2)$. In contrast, a brute force method will take $O(2^n)$.

Blackhole-freeness: A blackhole is a situation where a router unintentionally drops traffic. The blackholed traffic from A to B is the complement of the reachable traffic: $blackhole_{A \rightsquigarrow B} = \overline{reachability_{A \rightsquigarrow B}}$. Note that computing blackholes by ERA having computed reachability takes $O(1)$, as the negation of a BDD is the same BDD with its two leaves (corresponding to true and false) flipped.

Waypointing: Operators may want traffic from A to B to go through an intended sequence of routers (e.g., to enforce advanced service chaining policies [62, 136]). ERA checks waypointing by explicitly checking whether traffic reachable from A to B goes through the intended routers. The pseudocode in Figure 4-15 shows how FlowTags checks a waypointing policy.

Loop-freeness: ERA can find permanent forwarding loops (e.g., created by static or aggregate routes—see Figure4-16c in §4.4.1) by checking whether the same router port appears twice in the reachability result.

4.3.5 Implementation

Our implementation of ERA [2] supports several configuration languages (e.g., Cisco IOS, JunOS, Arista). It uses Batfish's configuration parser, which normalizes a vendor-specific configurations to vendor-agnostic format. ERA, then, uses this vendor-agnostic format as

```
1  ▷ Inputs: (1) router-level topology of network G = (V, E)
             (2) routers configurations
             (3) Intended way-pointing path from router port A to router port B
2  ▷ Output: Whether the way-pointing policy is enforced correctly

3  Compute all paths from B to A in G: Path_{A⇝B} = {path_1, ..., path_N}
   ▷ Note that the intended way-pointing path path* ∈ Path_{A⇝B}
4  Parse router configurations into boolean functions (using Figure 4-12)
5  Compute atomic predicates of routers present in Path_{A⇝B} (using Figure 4-12)
   denoted by AP_{A⇝B} = {AP(Router_1), ..., AP(Router_M)}
6  ▷ Compute per-path reachability by computing intersections (using Figure 4-14)
7  ViolatingPaths = ∅ ▷ Initializing the result
8  for i=1 to N
9     if (path_i ≠ path*)&&(reachability^{path_i}_{A⇝B} ≠ ∅)
10       ViolatingPaths = ViolatingPaths ∪ path_i
11 ▷ Checking whether all traffic from A to B only goes through the intended path
12 if (ViolatingPaths == ∅)&&(reachability^{path*}_{A⇝B} == reachability_{A⇝B})
13    return true
14 else
15    return ViolatingPaths
```

Figure 4-15: Pseudocode for checking waypointing for A-to-B traffic.

input. We implement the control plane model, the K-map, and atomic predicates in `Java`. To operate on BDDs, we use the `JDD` library [27]. We implement our fast set intersection and union algorithms in `C` using Intel AVX2, which expands traditional integer instructions to 256 bits [52].

A natural question might be how much effort it takes to add support for various routing protocols to ERA. In our experience, this effort is minimal. It took two of the authors a few hours to model the common routing protocols because of two reasons. First, there are fewer than 10 common routing protocols (e.g., BGP, OSPF, RIP, IS-IS). Second, for each protocol, the key insight for creating the model is to know how the protocol prefers a route over another in the steady state, which is concisely defined in protocol specifications.

4.4 Evaluation

In this section, we evaluate ERA and find that:

- It can help find both known and new reachability violations (§4.4.1);
- It can scale to large networks (e.g., it can analyze a network with over 1,600 routes in 6 seconds), and our design choices are key to its scalability (§4.4.2).

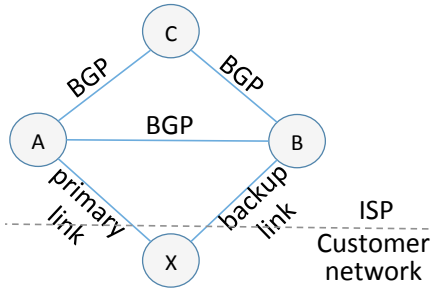
4.4.1 Finding reachability bugs with ERA

First we show the utility of ERA in finding reachability violations in scenarios involving known bugs as well as new bugs across both real and synthetic scenarios. These scenarios illustrate violations that are latent and get triggered only in certain environments (i.e., a certain router advertisement sent to the network by the routers located int the outside world). Even for scenarios involving only a small number of routers, existing network verification techniques lack the ability to find latent bugs (§4.2), and trying to extend these tools to enumerate different environments poses a serious scalability challenge (e.g., we will quantify this for Batfish, a recent network verification tool, in §4.4.2). Further, as we will discuss in §4.4.2, ERA scales to large networks (e.g., over 1,600 routers).

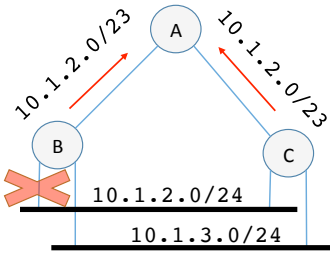
All experiments below were done under the assumption that the environment sends *all* possible route announcements, i.e. the BDD of each environmental input is simply the predicate *true*. Though this environmental assumption is not guaranteed to cover all possible environments, in practice it is effective at rooting out latent bugs due to its "maximal" nature, as we show below. This points out an important advantage of ERA over Batfish [99]. While both tools require an environment as input, Batfish's low-level simulation of routing protocols makes it prohibitively expensive to run with such a maximal environment, so in practice Batfish users must craft specific environments that are suspected to cause problems.

Finding Known Bugs in Synthetic Scenarios

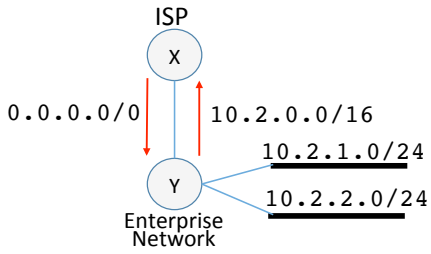
- **Violation of waypointing due to route redistribution:** In this scenario borrowed from [132] and shown in Figure 4-16a, the customer wants to waypoint its traffic through $X - A - C$ and use $X - B - C$ as the backup path. However, static routes configured on routers A and B are redistributed into BGP, and the ISP advertises them into the rest of the Internet. As a result, $B - X$ acts as a primary link. (One way to prevent this would be for the customer to adjust the default AD values of BGP and static routes on B.)
- **Blackhole due to route aggregation:** In this scenario borrowed from [134] and shown in Figure 4-16b, both routers B and C are configured to announce aggregate route 10.1.2.0/23 to router A. After the marked interface of B fails, B continues to announce



(a) Violation of waypointing [132].



(b) Blackhole [134].



(c) Permanent loop [134].

Figure 4-16: Finding known bugs in synthetic scenarios.

the aggregate route, which causes A to send packets destined to 10.1.2.0/24 to B. B will drop this traffic, as the its link to the 10.1.2.0/24 subnetwork is down.

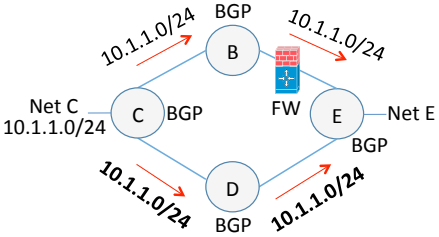
- **Permanent loop due to route aggregation:** In this scenario borrowed from [134] and shown in Figure 4-16c, the ISP router X advertises the default route 0.0.0.0/0 to router Y. Even though Y has connectivity to only 10.2.1.0/24 and 10.2.2.0/24, it has been configured to advertise to the ISP the aggregate route for the entire 10.2.0.0/16 prefix. Now since 10.3.0.0/24 is as sub-prefix of 10.2.0.0/16, the ISP may send traffic to destination prefix 10.3.0.0/24 to Y. Consequently, since Y does not know how to reach 10.3.0.0/24, this traffic will match its default route entry and be bounced back to the ISP. This traffic, therefore, will trap in a permanent loop between X and Y.

To further evaluate the effectiveness of ERA, we did a red team-blue team exercise. In each scenario, the red team introduced misconfigurations that cause a reachability violation unbeknownst to the blue team. Then the blue team uses ERA to check whether the intended policy is violated. Across all scenarios, the blue team successfully found the violation. Here is a summary of the scenarios:

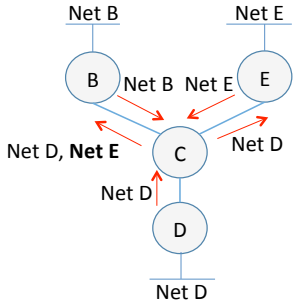
- **Violation of waypointing:** In Figure 4-17a, the intended policy is to ensure traffic originating from network E destined to network C goes through path $E - B - C$ (so that it

is scrubbed by the firewall). However, this policy is violated because router E receives the prefix of network C from both routers B and D, which means $NetE \rightarrow NetC$ traffic may go through path $E - D - C$ skipping the firewall. The root cause of the problem was the fact that none of routers C, D, or E filtered the route advertisement for the 10.1.1.0/24 prefix on the $E - D - C$ path.

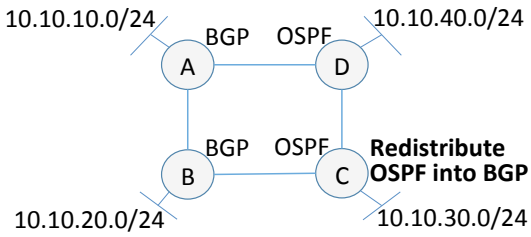
- **Violation of valley-free routing:** In Figure 4-17b, B and E are providers for C, which in turn, is a provider for D. A missing export filter on C caused C to advertise the prefix for $NetE$ to B. This is a violation of the valley-free routing property, specifically, due to customer C providing connectivity between two of its providers, namely, B and E.
- **Violation of intended isolation:** In Figure 4-17c, we want the traffic from segments $\{A, B\}$ (running BGP) and $\{C, D\}$ (running OSPF) to remain isolated from each other. However, this policy is violated due to a misconfiguration on C whereby OSPF is redistributed into BGP, that will allow traffic from $\{A, B\}$ to reach $\{C, D\}$.
- **Misconfigured backup path:** In Figure 4-17d, the client has two /16 networks connected to A and intends to maintain two paths to the provider to ensure reachability in case of failure on one of them. This policy is violated because of an incorrect filter configured on B that drops the advertisement for the 10.20.0.0/16 network. As a result, if path $D - C - A$ fails, the 10.20.0.0/16 network will be unreachable from the provider.



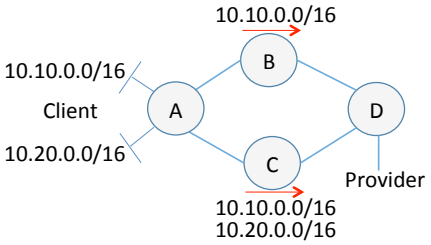
(a) Violation of waypointing via B.



(b) Not valley-free.



(c) Violation of isolation between $\{A, B\}$ and $\{C, D\}$.



(d) Misconfigured backup path $D - B - A$.

Figure 4-17: Finding known bugs in synthetic scenarios using the red-blue teams exercise.

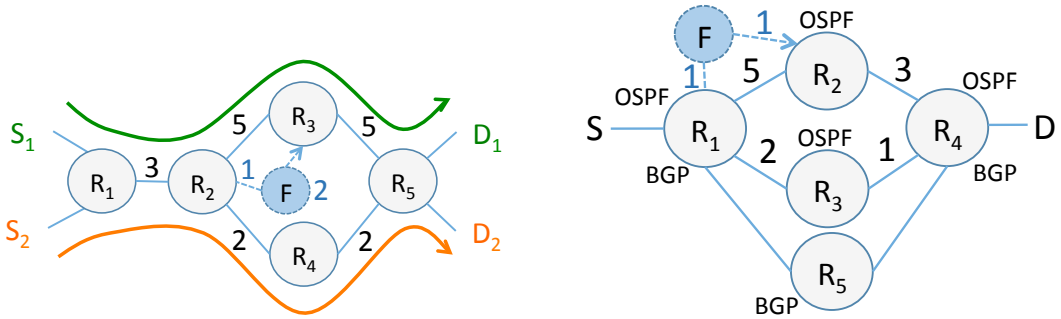
Finding New Bugs in Synthetic Scenarios

Finding reachability bugs in hybrid networks: Operators may prefer to opt for a hybrid network, which involves deploying SDN alongside traditional network routing infrastructure for scalability and fault tolerance [174]. Next we show how ERA can find policy violations arising in such hybrid deployments.

Fibbing [174] is a recent method to allow an operator to use an SDN controller to flexibly enforce way-pointing policies in a network running vanilla OSPF. The key primitive is "fibbing" whereby the SDN controller pretends to be a neighboring router and makes fake route advertisements with carefully crafted costs. For example, consider the network of Figure 4-18a, where links are annotated with their OSPF weights. If we run OSPF, both source to destination flows will take the cheaper path $R_1 - R_2 - R_4 - R_5$. Now, for load balancing purposes, the operator wants to make $S_1 \rightarrow D_1$ traffic take the path $R_1 - R_2 - R_3 - R_5$ without fiddling with OSPF wights. Fibbing will let her accomplish this by using a fake router F that claims to be able to reach D_1 at a cost of 2. As a result, now R_2 will start sending traffic destined to D_1 through F, as the new cost 1+2=3 is better

than the cost 2+2=4 of going through R_4.

A hybrid network is particularly error-prone due to intricate interactions between SDN and traditional protocols. To show the utility of ERA in reasoning about such networks, we describe two scenarios:



(a) Route aggregation on R_2.

(b) Cross-protocol effects.

Figure 4-18: New bugs in a synthetic scenario involving hybrid (i.e., SDN-traditional) networks.

- **Interaction between fibbing and aggregate routes:** In Figure 4-18a, the goal is to use fibbing to enforce the waypointing denoted by green and orange paths. We used ERA to find a violation of this policy. The root cause was an aggregate route configured on R_2 to destination prefix $D_1 \cup D_2$ pointing to R_4 as its next hop. As a result, both $S_1 \rightarrow D_1$ and $S_2 \rightarrow D_2$ traversed the orange path, which violated the policy.
- **Cross-protocol effects:** In Figure 4-18b, the goal is to use fibbing to waypoint traffic to D through $R_1 - R_2 - R_4$. We used ERA to find a violation of this in a red team-blue team exercise. Each router in the figure is annotated with the routing protocol(s) it runs. Router R_4 had a static route to D that is redistributed into BGP and OSPF. As a result, router R_1 received route advertisements for D from both OSPF (from R_2 and R_3) and BGP (from R_5). Now since BGP, by default, has a lower AD value than OSPF, R_1 chose the advertisement offered by R_5! Therefore, fibbing here fails to enforce the waypointing policy.

Fibbing is proven to be correct [174], but only if the network merely runs OSPF. The takeaway from the above scenarios is that for hybrid networks to be practical, we need to account for realistic router configurations (e.g., route aggregation by R_2 in Figure 4-18a) and cross-protocol interactions (e.g., BGP/OSPF in Figure 4-18b).

Note that finding arbitrary SDN bugs is beyond the scope of ERA. ERA handles SDN only if its behavior can be abstracted in our control plane model, in a manner similar to what we do for conventional routing protocols.

Finding Known Bugs in Real Scenarios

Bugs reported in a cloud provider: The motivating scenarios we saw in §4.1 are based on bugs in a production network that we successfully reproduced using ERA.

Finding BGP route leaks: Roughly speaking, a route leak scenario involves: *(i)* a router incorrectly advertising the destination prefix of a service, and *(ii)* another router incorrectly accepting it. The combination of these results in absorbing traffic destined to the service on the wrong path, which can cause high-impact disruptions. Route leak is not a new problem (e.g., see AS 7007 incident [3]), but continues to plague the Internet to date (e.g., Google [19] and Amazon AWS [46] outages in 2015). To demonstrate the utility of ERA in proactive finding of route leak-prone configurations, we use a representative scenario shown in Figure 4-19. The intended path from the client to the service is through R_2; however, the client's traffic ends up taking the wrong path $C \rightarrow R_1$ because *(i)* R_1 incorrectly advertises the service prefix, and *(ii)* C prefers the route advertisement made by R_1 over the one made by R_2. ERA can proactively find route leaks, as a route leak is essentially a violation of waypointing. In this example, the traffic from client to server needs to be exclusively waypointed through R_2. We have synthesized a few route leak scenarios and used ERA to successfully find violations.

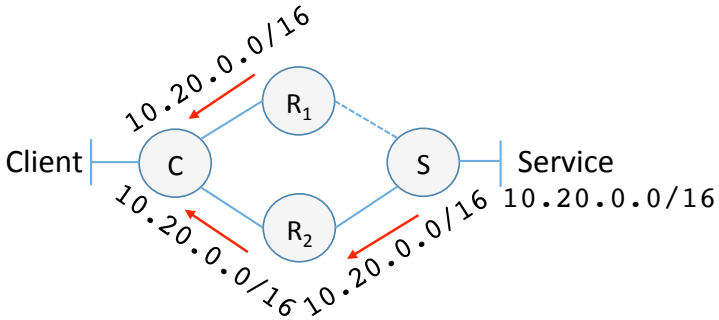


Figure 4-19: R_1 **leaks the service prefix.**

Finding New Bugs in Real Scenarios

Next we show the utility of ERA in finding new bugs in a campus (*CampusNet*) and a large cloud (*CloudNet*).

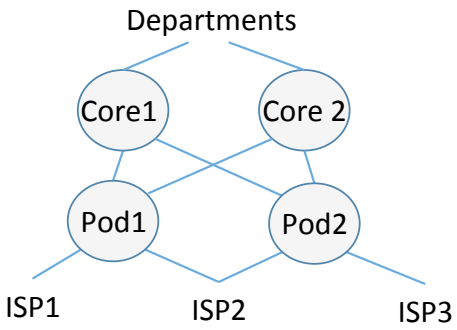


Figure 4-20: A schematic of the analyzed *CampusNet*.

Finding new bugs in *CampusNet*: Figure 4-20 shows a simplified topology of the core of a large campus network, with a global footprint and over 10K users. The two core routers are in charge of interconnecting the three ISPs and the departments. There are two intended policies involving these four routers, both of which are violated:

- **Equivalence of core routers:** $Core2$ is meant to be $Core1$'s backup. ERA revealed that $Core1$ has OSPF configured on one of its interfaces, which is missing on $Core2$. As a result, if $Core1$ fails, the departments that rely on OSPF will be disconnected from the Internet.
- **Equivalence of pod routers:** $Pod1$ and $Pod2$, connecting the campus to the Internet, are both connected to $ISP2$ with the intention that link $Pod1 - ISP2$ is active and $Pod2 - ISP2$ is its backup. ERA revealed that the ACLs on $Pod1$ and $Pod2$ affecting their respective links with $ISP2$ are different. Specifically, $Pod2$ has more restrictive ACLs than $Pod1$. This means if link $Pod1 - ISP2$ fails, a subset of campus-to-$ISP2$ traffic will be mistakenly dropped by $Pod2$.

Finding new bugs in *CloudNet*: We used ERA to check equivalence of same-tier routers (analogous to routers R_1 and R_2 in Figure 4-2) on configurations of seven production datacenters of a large cloud provider. ERA revealed that seven routers in two datacenters had a total of 19 static routes responsible for violations of equivalence policies. The operators later removed all of these violating routes.

4.4.2 Scalability of ERA

Testbed: We run our scalability evaluation experiments on a desktop machine (4-core 3.50GHz, 16GB RAM).

Why not existing tools? The closest tool to ERA is Batfish [99], which *(1)* takes a concrete network environment; *(2)* runs a high-fidelity model of the control plane (e.g., low-level models of various routing protocols) to generate the data plane (i.e., routers forwarding tables); and *(3)* performs data plane reachability analysis. To put this in perspective, in an example scenario involving a chain topology with two routers, Batfish took about 4 seconds. In contrast, ERA took 0.17 seconds to analyze the same network (a 23X speedup over Batfish). Further, as mentioned earlier, Batfish's performance will degrade as the size of the environment increases, while ERA's BDD-based approach allows it to naturally handle even the "maximal" environment, represented by the BDD *true*.

Effect of optimizations: Table 4.1 shows the effect of our optimizations from §4.3.3, namely, the K-map, equivalence classes (EC), and fast set operations compared to a baseline involving use of BDDs without these optimizations. The tables shows the average values from 100 runs, each involving A-to-B reachability analysis between two randomly selected ports. Stanford [50] and Purdue [43] are campus networks, OTEGlobe [53] is an ISP, and FatTree is a synthetic datacenter topology. The takeaway here is that our optimizations yield a speedup of 2.5$\times$ to 17$\times$, making ERA sufficiently fast to be interactively usable.

Topo.	#routers/ave path len.	Reachability analysis latency (sec)			
		baseline	kmap	kmap+EC	ERA
Stanford	16/2	5	1.8	0.30	0.29
OTEglb	92/3.3	7.8	3.5	1.97	1.84
FatTree	1,024/5.89	13.8	7.01	6.1	5.4
Purdue	1,646/6.8	15	8	6.5	6

Table 4.1: Effect of our optimizations.

To see the effect of the type of policy on the analysis latency, we measured the analysis latency for all properties from §4.3.4 on the Purdue and OTEGlobe topology, none of which took more than 6.1 seconds. This is expected, as these policies are derivatives of

reachability analysis.

4.5 Summary

Since networks are constantly changing (e.g., new route advertisements, link failures), operators want the ability to reason about reachability policies across many possible changes. In contrast to prior work, which either focuses on a subset of the network's control plane or focuses on one incarnation of the network as represented by a single data plane, ERA models the entire control plane and checks network reachability directly in that model. Our design addresses key expressiveness and scalability challenges via a unified protocol-invariant routing abstraction, a compact binary decision diagram based encoding of the routers' control plane, and a scalable application of boolean operations (e.g., vector arithmetic).

We showed that ERA provides near-real-time analysis capabilities that can scale to datacenter and enterprise networks with hundreds of nodes and uncover a range of latent reachability bugs. While ERA does not automatically reason about all possible of environments, it helps find latent reachability bugs by allowing the users to specify a rich *set* environments using BDDs and quickly analyzing each such set. For instance, a particularly challenging environment, of all possible routing announcements from a neighbor, can be captured simply using BDD *true*.

A direction for future work is to identify conditions under which a single run of ERA is guaranteed to cover all possible environments and extend ERA to automatically explore all possible environments. Another natural direction for future work is to prioritize bug fixing based on the likelihood of occurrence and severity of aftermath, and to bring the human operator into the debugging and repair loop.

Chapter 5

Conclusions and future work

While there are many tools as well as a significant body of work related to network policy enforcement, checking whether the intended policies have been enforced correctly remains a mostly manual, slow, and error-prone process. The current practice has led to many network outages and policy violations resulting in monetary and reputation damage to organizations.

What makes it hard to check the correctness of network policies is *diversity* (e.g., different routing protocols and their interactions) and *stateful behaviors* in both the data and control planes (e.g., the route advertisements a router has received so far) and the data plane (e.g., a firewall's state with respect to a TCP session).

This thesis argues for the feasibility of checking realistic network policies by addressing the above challenges via a synergistic combination of two key insights *(i)* designing unifying data abstractions that glue together different routing protocols in the control plane and diverse elements in the data plane; *(ii)* exploring the state space of the network in a scalable manner, we build tractable models of the control and data planes (e.g., by decomposing logically independent tasks) and design domain-specific optimizations (e.g., by narrowing down the scope of search given the intended policies). We have shown the utility and performance of these techniques across a range of synthetic and real settings.

Going forward, there are several natural directions for future work :

- **Automatic policy synthesis** The input to both BUZZ and ERA is a specification of

the intended policies by the network operator. While the level of details at which the operator is expected to provide this input is similar to the current practice (e.g., existing firewalls ACL files), we believe it is worthwhile to provide the operator with a more intuitive policy interface (e.g., human language [183]).

- **Automatic model synthesis for control and data planes:** BUZZ and ERA use hand-generated models of the data and control planes, respectively. A natural direction for future work is to use program analysis to automatically synthesize these models from actual middlebox and router implementation code (e.g., [84]) or input-output logs (e.g., [74]).

- **Proof of soundness:** We found both BUZZ and ERA to be empirically sound in that every bug they found was a real bug. However, we have not proven the soundness of these tools. While it is customary in system research to rely on empirical soundness (e.g., [148, 179]), it will be worthwhile to try to prove if these tools, or at least their core algorithms, are false-positive-free.

- **Reducing false negatives:** A major direction for future work would be trying to build policy checkers that minimize false negatives. Note that there are two types of false negatives associated with network policy checking tools, including BUZZ and ERA. The first category of false negatives are not due to an inherent limitation of the policy checker but is considered a false negative only because of the real-world interpretation of the scenario. For example, ERA finds route advertisements that, if sent by a neighboring ISP, will cause a reachability policy violation. The human operator, however, may consider such an output from ERA a false negative considering the fact that the neighboring ISP is highly reputable and it very unlikely to make such a blatant mistake. As reflected in the design of ERA, we believe it is better to cover such scenarios as bugs and bring them to the operator's attention; e.g., there have been many cases of route leak by major ISPs due to unexpected mis-configuration.

  The second type of false negatives are due to the limitation of the tools we have built. Specifically, ERA assumes routing protocols have converged; therefore, it cannot find

transient and convergence bugs in routing protocols. Further, ERA cannot find bugs that stem from BGP community tags, as these tags can be used to arbitrarily override the normal course of actions taken by routers in route selection. Trying to accommodate community tags would make our router model significantly more complex and harder to explore. Similarly, BUZZ assumes that middleboxes perform state transitions given the current state and the input as expected, and does not account for race conditions. In our experience in building both ERA and BUZZ, there is fundamental trade-off between reducing false negatives and scalability of the tool: A more faithful modeling approach (e.g., capturing community tags for control plane analysis, or code-level details in modeling middleboxes) would help reduce false negatives at the cost of slowing down the analysis.

- **Root cause analysis of policy violations:** While BUZZ and ERA find policy violations, they do not directly provide an actionable output to the operator to help fix the violation. For example, while ERA identifies which route advertisements would violate an intended reachability policy by a router, it does not specify where in the router's configuration file the operator should look for the violation. This is a hard problem, as a given mis-configuration can be due to either a commission (e.g., the presence of an incorrect configuration directive) or omission (e.g., the lack of a necessary output filter in a configuration file) error. To make it even harder, a given violation may be fixable in many different ways. Therefor, assisting the human operator in fixing a policy violation, will be an important direction for future work.

- **Providing minimal output to human operator:** Network operators are typically busy professionals and may not be able to pay attention to every potential policy violation. A broad direction for future work, therefore, is to try to prioritize bugs based on the likelihood of occurrence and severity of aftermath.

- **Automatic implementation of data and control planes:** Instead of reasoning about exiting networks, a bold direction for future work will be to build a system for synthesizing correct-by-design router and middlebox code.

Bibliography

[1] BUZZ. https://github.com/network-policy-tester/buzz.

[2] ERA. https://github.com/Network-verification/ERA.

[3] 7007 Explanation and Apology, 1997. http://seclists.org/nanog/1997/Apr/444.

[4] Arbor Networks, worldwide infrastructure security report, volume IX, 2014. http://pages.arbornetworks.com/rs/arbor/images/WISR2014.pdf.

[5] AT&T and Intel: Transforming the Network with NFV and SDN, 2014. https://www.youtube.com/watch?v=F55pHxTeJLc#t=76.

[6] AT&T Denial of Service Protection, 2015. http://soc.att.com/1IIlUec.

[7] AT&T Domain 2.0 Vision White Paper, 2013. http://soc.att.com/1kAw1Kp.

[8] Balance. http://www.inlab.de/balance.html.

[9] BGP Message Generation and Transport, and General Message Format, 2016. http://www.tcpipguide.com/free/t_BGPMessageGenerationandTransportandGeneralMessageF-2.htm.

[10] Bit-Twist. http://bittwist.sourceforge.net/.

[11] Border Gateway Protocol Path Selection, 2016. http://docwiki.cisco.com/wiki/Border_Gateway_Protocol#BGP_Path_Selection.

[12] The CAIDA AS relationships dataset, Jan 2010. http://www.caida.org/data/active/as-relationships/.

[13] CBMC. http://www.cprover.org/cbmc/.

[14] Cisco—What Is Administrative Distance?, 2013. http://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/15986-admin-distance.html.

[15] Cisco's Reflexive Access Lists, 2016. http://www.cisco.com/c/en/us/td/docs/ios/12_2/security/configuration/guide/fsecur_c/scfreflx.html.

[16] CloudFlare. https://www.cloudflare.com/ddos.

[17] Dell PowerEdge Rack Servers, 2015. http://www.dell.com/us/business/p/poweredge-rack-servers.

[18] Emulab. http://www.emulab.net/.

[19] Finding and Diagnosing BGP Route Leaks, 2015. https://blog.thousandeyes.com/finding-and-diagnosing-bgp-route-leaks/.

[20] Graphplan. http://www.cs.cmu.edu/~avrim/graphplan.html.

[21] GSA Advantage, 2015. http://1.usa.gov/1ggEgFN.

[22] How pakistan knocked youtube offline (and how to make sure it never happens again). http://www.cnet.com/news/how-pakistan-knocked-youtube-offline-and-how-to-make-sure-it-never-happens-again/.

[23] httperf. https://code.google.com/p/httperf/.

[24] Incapsula Survey : What DDoS Attacks Really Cost Businesses, 2014. http://lp.incapsula.com/rs/incapsulainc/images/eBook%20-%20DDoS%20Impact%20Survey.pdf.

[25] iperf. https://code.google.com/p/iperf/.

[26] iptables. http://www.netfilter.org/projects/iptables/.

[27] JDD, a pure Java BDD and Z-BDD library. https://bitbucket.org/vahidi/jdd/wiki/Home.

[28] Juniper—Route Preferences. http://juni.pr/1fQC4LY.

[29] KCachegrind. http://kcachegrind.sourceforge.net/html/Home.html.

[30] Kinetic. http://resonance.noise.gatech.edu/.

[31] KVM. http://www.linux-kvm.org/page/Main_Page.

[32] Mininet. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

[33] Network Function Virtualization Research Group (NFVRG). https://irtf.org/nfvrg.

[34] ns-3. http://www.nsnam.org/.

[35] NTP attacks: Welcome to the hockey stick era, 2014. http://it.toolbox.com/companies/arbor-networks-inc/news/ntp-attacks-welcome-to-the-hockey-stick-era-92429.

[36] ONS 2014 Keynote: John Donovan, Senior EVP, AT&T Technology & Network Operations. https://www.youtube.com/watch?v=tLshR-BkIas.

[37] OpenDaylight project. http://www.opendaylight.org/.

[38] Openflow switch specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[39] OSPF Message Formats. http://www.tcpipguide.com/free/t_OSPFMessageFormats.htm.

[40] POX Controller. http://www.noxrepo.org/pox/about-pox/.

[41] PRADS. http://gamelinux.github.io/prads/.

[42] Prolexic. http://www.prolexic.com/.

[43] Purdue campus network configuration files, 2008. https://engineering.purdue.edu/~isl/network-config/.

[44] Radware. http://www.radware.com/Solutions/Security/.

[45] Rfc 3954, cisco systems netflow services export version 9, 2004. https://tools.ietf.org/html/rfc3954.

[46] Route Leak Causes Amazon and AWS Outage, 2015. https://blog.thousandeyes.com/route-leak-causes-amazon-and-aws-outage/.

[47] Scapy. https://isc.sans.edu/forums/diary/TCP+Fuzzing+with+Scapy/14080/.

[48] Snort. http://www.snort.org/.

[49] Squid. http://www.squid-cache.org/.

[50] Stanford campus network configuration files, 2012. https://bitbucket.org/peymank/hassel-public/src/697b35c9f17ec74ceae05fa7e9e7937f1cf36878/hsa-python/examples/stanford/Stanford_backbone/?at=master.

[51] tcpdump. http://www.tcpdump.org/.

[52] The Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX2.

[53] The Internet Topology Zoo. http://www.topology-zoo.org/index.html.

[54] Time for an SDN Sequel?, 2014. https://www.sdxcentral.com/articles/news/scott-shenker-preaches-revised-sdn-sdnv2/2014/10/.

[55] Troubleshooting the network survey, 2011. http://eastzone.github.io/atpg/docs/NetDebugSurvey.pdf.

[56] Valgrind. http://www.valgrind.org/.

[57] Vast Challenge, 2013. http://vacommunity.org/VAST+Challenge+2013%3A+Mini-Challenge+3.

[58] Verizon-Carrier Adoption of Software-defined Networking, 2012. https://www.youtube.com/watch?v=WVczl03edi4.

[59] What was wrong with United's router?, 2015. http://www.networkworld.com/article/2946070/cisco-subnet/what-was-wrong-with-uniteds-router.html.

[60] World Enterprise Network and Data Security Markets. https://www.abiresearch.com/market-research/product/1006059-world-enterprise-network-and-data-security/.

[61] Enabling Service Chaining on Cisco Nexus 1000V Series. http://www.cisco.com/c/en/us/products/collateral/switches/nexus-1000v-switch-vmware-vsphere/white_paper_c11-716028.pdf, 2013.

[62] High Performance Service Chaining for Advanced Software-Defined Networking (SDN). https://networkbuilders.intel.com/docs/Intel_Wind_River_Demo_Brief.pdf, 2014.

[63] Tackling the Dynamic Service Chaining Challenge of NFV/SDN Networks with Wind River and Intel. http://itpeernetwork.intel.com/tackling-the-dynamic-service-chaining-challenge-of-nfvsdn-networks-with-wind-river-and-intel/, 2014.

[64] Paving the way for nfv: Simplifying middlebox modifications using statealyzr. In *Proc. NSDI*, 2016.

[65] Richard Alimi, Ye Wang, and Y. Richard Yang. Shadow configuration as a network management primitive. In *Proc. SIGCOMM*, 2008.

[66] Shane Amante, Brian Carpenter, Sheng Jiang, and Jarno Rajahalme. IPv6 Flow Label Update. http://rmv6tf.org/wp-content/uploads/2012/11/rmv6tf-flow-label11.pdf.

[67] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.

[68] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xOMB: extensible open middleboxes with commodity servers. In *Proc. ANCS*, 2012.

[69] Bilal Anwer, Theophilus Benson, Nick Feamster, and Dave Levin. Programming slick network functions. In *Proc. SOSR*, 2015.

[70] Thomas Ball, Nikolaj Bjorner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarskyi. VeriCon: Towards verifying controller programs in software-defined networks. In *Proc. PLDI*, 2014.

[71] Gaurav Banga and Jeff Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. USENIX ATC*, 1998.

[72] Paul Barford, Jeffery Kline, David Plonka, and Amos Ron. A signal analysis of network traffic anomalies. In *Proc. ACM SIGCOMM Workshop on Internet Measurement*, 2002.

[73] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proc. SIGCOMM*, 2016.

[74] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proc. ESEC/FSE*, 2011.

[75] Pat Bosshar, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proc. SIGCOMM*, 2013.

[76] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[77] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10ˆ20 States and Beyond. *Inf. Comput.*, 98(2), 1992.

[78] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, 2008.

[79] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[80] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test openflow applications. In *Proc. NSDI*, 2012.

[81] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *Proc. SIGCOMM*, 2007.

[82] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A retrospective on evolving sdn. In *Proc. HotSDN*, 2012.

[83] Margaret Chiosi and et al. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. `http://portal.etsi.org/nfv/nfv_white_paper.pdf`, 2012.

[84] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, 2000.

[85] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[86] Ryan Craven, Robert Beverly, and Mark Allman. A middlebox-cooperative tcp for a non end-to-end internet. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 151–162. ACM, 2014.

[87] Jakub Czyz, Michael Kallitsis, Manaf Gharaibeh, Christos Papadopoulos, Michael Bailey, and Manish Karir. Taming the 800 pound gorilla: The rise and decline of ntp ddos attacks. In *Proc. IMC*, 2014.

[88] Mihai Dobrescu, Katerina Argyarki, and Sylvia Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *Proc. NSDI*, 2012.

[89] Mihai Dobrescu and Katerina Argyraki. Software dataplane verification. In *Proc. NSDI*, 2014.

[90] Daniel J. Dougherty, Timothy Nelson, Christopher Barratt, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *Proc. LISA*, 2010.

[91] Li Erran Li, Z. Morley Mao, and Jennifer Rexford. CellSDN: Software-defined cellular networks. In *Techinical Report, Princeton University, 2013*.

[92] Seyed K. Fayaz, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *Proc. NSDI*, 2014.

[93] Seyed K. Fayaz and Vyas Sekar. Testing stateful and dynamic data planes with FlowTest. In *Proc. HotSDN*, 2014.

[94] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *Proc. OSDI*, 2016.

[95] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic DDoS defense. In *Proc. USENIX Security Symposium*, 2015.

[96] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. BUZZ: Testing context-dependent policies in stateful networks. In *Proc. NSDI*, 2016.

[97] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proc. NSDI*, 2005.

[98] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn. *Queue*, 11(12):20:20–20:40, December 2013.

[99] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proc. NSDI*, 2015.

[100] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proc. NSDI*, 2007.

[101] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *SIGPLAN Not.*, 46(9), September 2011.

[102] Lixin Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Trans. Netw.*, 9(6), December 2001.

[103] Aaron Gember, Anand Krishnamurthy, Saul St. John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *CoRR*, abs/1305.0209, 2013.

[104] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proc. SIGCOMM*, 2016.

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

[106] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. Management plane analytics. In *Proc. IMC*, 2015.

[107] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *ACM Queue*, 2012.

[108] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. *ACM CCR*, 2005.

[109] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 10(2):232–243, April 2002.

[110] Timothy G. Griffin and Joäo Luís Sobrinho. Metarouting. In *Proc. SIGCOMM*, 2005.

[111] Timothy G. Griffin and Gordon Wilfong. An analysis of bgp convergence properties. In *Proc. SIGCOMM*, 1999.

[112] Brandon Heller, Rob Sherwood, and Nick McKeown. The Controller Placement Problem. In *Proc. HotSDN*, 2012.

[113] Frederick J. Hill and Gerald R. Peterson. *Introduction to Switching Theory and Logical Design*. 1981.

[114] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend tcp? In *Proc. IMC*, 2011.

[115] Jain et al. B4: Experience with a globally-deployed software defined wan. In *Proc. SIGCOMM*, 2013.

[116] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 2009.

[117] Xin Jin, Li Erran Li, Laurent Vanbever, and Jennifer Rexford. Softcell: Scalable and flexible cellular core network architecture. In *Proc. CoNext*, 2013.

[118] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu. Research directions in network service chaining. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, 2013.

[119] Wolfgang John, Kostas Pentikousis, George Agapiou, Eduardo Jacob, Mario Kind, Antonio Manzalini, Fulvio Risso, Dimitri Staessens, Rebecca Steinert, and Catalin Meirosu. Research directions in network service chaining. *CoRR*, abs/1312.5080, 2013.

[120] D. Joseph and I. Stoica. Modeling middleboxes. *Netwrk. Mag. of Global Internetwkg.*, 22(5), 2008.

[121] Dilip A. Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers. In *Proc. SIGCOMM*, 2008.

[122] Min Suk Kang, Soo Bum Lee, and V.D. Gligor. The crossfire attack. In *Proc. IEEE Security and Privacy*, 2013.

[123] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Proc. NSDI*, 2013.

[124] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: static checking for networks. In *Proc. NSDI*, 2012.

[125] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.

[126] Donald Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. 2011.

[127] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 2000.

[128] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *Proc. NSDI*, 2014.

[129] Diego Kreutz, Fernando M.V. Ramos, and Paulo Verissimo. Towards secure and dependable software-defined networks. In *Proc. HotSDN*, 2013.

[130] Franck Le, Erich Nahum, Vasilis Pappas, Maroun Touma, and Dinesh Verma. Experiences deploying a transparent split tcp middlebox and the implications for nfv. In *Proc. HotMiddlebox*, 2015.

[131] Franck Le, Geoffrey G. Xie, Dan Pei, Jia Wang, and Hui Zhang. Shedding light on the glue logic of the internet routing architecture. In *Proc. SIGCOMM*, 2008.

[132] Franck Le, Geoffrey G. Xie, and Hui Zhang. Instability free routing: beyond one protocol instance. In *Proc. CoNEXT*, 2008.

[133] Franck Le, Geoffrey G. Xie, and Hui Zhang. Theory and new primitives for safely connecting routing protocol instances. In *Proc. SIGCOMM*, 2010.

[134] Franck Le, Geoffrey G. Xie, and Hui Zhang. On route aggregation. In *Proc. CoNEXT*, 2011.

[135] Erran Li, Vahid Liaghat, Hongze Zhao, MohammadTaghi Hajiaghayi, Dan Li, Gordon Wilfong, Y. Richard Yang, and Chuanxiong Guo. PACE: Policy-Aware Application Cloud Embedding. In *Proc. IEEE INFOCOM*, 2013.

[136] W. Liu, H. Li, O. Huang, M. Boucadair, N. Leymann, and J. Hu Z. Cao. Service Function Chaining (SFC) Use Cases. `https://tools.ietf.org/html/draft-liu-sfc-use-cases-01`, 2014.

[137] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *Proc. NSDI*, 2015.

[138] Lori MacVittie. Service chaining and unintended consequences. https://devcentral.f5.com/articles/service-chaining-and-unintended-consequences#.Uvbz0EJdVe9.

[139] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proc. SIGCOMM*, 2011.

[140] David A. Maltz, Geoffrey Xie, Jibin Zhan, Hui Zhang, Gísli Hjálmtýsson, and Albert Greenberg. Routing design in operational networks: A look from the inside. In *Proc. SIGCOMM*, 2004.

[141] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[142] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 1990.

[143] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. In *CCR*, 2004.

[144] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proc. NSDI*, 2013.

[145] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software Defined Networks. In *Proc. NSDI*, 2013.

[146] David Moore, Colleen Shannon, Douglas J. Brown, Geoffrey M. Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *ACM Trans. Comput. Syst.*, 2006.

[147] Y Mundada et al. Practical Data-Leak Prevention for Legacy Applications in Enterprise Networks. http://hdl.handle.net/1853/36612.

[148] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *Proc. OSDI*, 2002.

[149] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *J. Netw. Syst. Manage.*, 16(3), September 2008.

[150] Network functions virtualisation – introductory white paper. http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[151] Arbor Networks. ATLAS Summary Report: Global Denial of Service. http://atlas.arbor.net/summary/dos.

[152] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. NDSS*, 2005.

[153] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang amd Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for NFV applications. In *Proc. SOSP*, 2015.

[154] Aurojit Panda, Ori Lahav, Katerina J. Argyraki, Mooly Sagiv, and Scott Shenker. Verifying isolation properties in the presence of middleboxes. *CoRR*, 2014.

[155] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: cloud scale load balancing. In *Proc. ACM SIGCOMM*, 2013.

[156] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1999.

[157] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: Using graphs to express and automatically reconcile network policies. In *Proc. SIGCOMM*, 2015.

[158] Zafar Qazi, Cheng Tu, Luis Chiang, Rui Miao, and Minlan Yu. SIMPLE-fying middlebox policy enforcement using sdn. In *Proc. SIGCOMM*, 2013.

[159] P Quinn et al. Network service chaining problem statement. http://tools.ietf.org/html/draft-quinn-nsc-problem-statement-03.

[160] P. Quinn, J. Guichard, S. Kumar, P. Agarwal, R. Manur, A. Chauhan, N. Leymann, M. Boucadair, C. Jacquenet, M. Smith, N. Yadav, T. Nadeau, K. Gray, and B. McConnell. Network Service Chaining Problem Statement. https://tools.ietf.org/html/draft-quinn-nsc-problem-statement-02, 2013.

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

[162] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.

[163] Christian Rossow. Amplification hell: Revisiting network protocols for ddos abuse. In *Proc. USENIX Security*, 2014.

[164] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. NSDI*, 2012.

[165] Vyas Sekar, Sylvia Ratnasamy, Michael K. Reiter, Norbert Egi, and Guangyu Shi. The middlebox manifesto: enabling innovation in middlebox deployment. In *Proc. HotNets*, 2011.

[166] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback recovery for middleboxes. In *Proc. SIGCOMM*, 2015.

[167] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. SIGCOMM*, SIGCOMM, 2012.

[168] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks. In *Proc. CCS*, 2013.

[169] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP topologies with Rocketfuel. In *Proc. SIGCOMM*, 2002.

[170] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Static checking for stateful networks. In *Proc. HotMiddlebox*, 2013.

[171] Ahren Studer and Adrian Perrig. The coremelt attack. In *Proc. ESORICS*, 2009.

[172] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5), 2012.

[173] Patrick Verkaik, Dan Pei, Tom Scholl, Aman Shaikh, Alex C. Snoeren, and Jacobus E. van der Merwe. Wresting Control from BGP: Scalable Fine-grained Route Control. In *Proc. USENIX ATC*, 2007.

[174] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. Central control over distributed routing. In *Proc. SIGCOMM*, 2015.

[175] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Bagpipe: Verified BGP configuration checking. In *Proc. OOPSLA*, 2016.

[176] Wenfei Wu, Guohui Wang, Aditya Akella, and Anees Shaikh. Virtual network diagnosis as a service. In *Proc. SoCC*, 2013.

[177] Geoffrey Xie, Jibin Zhan, David Maltx, Hui Zhang Gisli Hjalmtysson, and Jennifer Rexford. On Static Reachability Analysis of IP Networks. In *Proc. INFOCOM*, 2005.

[178] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE Transactions on Networking*, 2015.

[179] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4), November 2006.

[180] S.H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of software-defined networking. *Communications Magazine, IEEE*, 2013.

[181] Tianlong Yu, Seyed K. Fayaz, Michael Collins, Vyas Sekar, and Srinivasan Seshan. PSI: Precise security instrumentation for enterprise networks. In *Proc. NDSS*, 2017.

[182] Lihua Yuan and Hao Chen. FIREMAN: a toolkit for FIREwall Modeling and ANalysis. In *Proc. IEEE Symposium on Security and Privacy*, 2006.

[183] Yifei Yuan, Rajeev Alur, and Boon Thau Loo. Netegg: Programming network policies by examples. In *Proc. HotNets*, 2014.

[184] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proc. CoNEXT*, 2012.

[185] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proc. NSDI*, 2014.

[186] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proc. SIGMOD*, 2010.