Abstract of “Towards a Reliable and Predictable Network” by Da Yu, Ph.D., Brown University, **Build time: Tuesday 26th March, 2019 (22:44)**.

Today’s tools to improve the reliability and manageability of networks can be generally classified into two different classes: before-the-fact network verification and after-the-fact network troubleshooting. Unfortunately, neither of the two classes can individually make the network fully reliable and predictable due to fundamental limitations.

Recently, there have been proposals to make the verification and troubleshooting constantly work together in a continuous cycle. The cycle involves verifying network-wide properties with latest network configurations, then monitoring the run-time network state and localizing the root cause once some network issue happens, and changing configurations in response.

However, state-of-the-art tools in this cycle cannot be used in large real-world production networks. This is because these tools fail to take some important realistic challenges into account. For example, modeling an enterprise-scale network’s behavior is non-trivial. A wrong model impacts the verification accuracy. For another example, handling complex header transformations due to multiple encapsulations is difficult, thus making existing troubleshooting efforts impractical to locate deep root causes.

This dissertation introduces two practical tools fit into this cycle and address important limitations of previous works. Meanwhile, we propose a general and flexible network behavior model. The first tool, Titan, focuses on verification. It consumes configurations to be deployed and answers questions the operators have regarding the reachability of routes and packets under failure cases. We show that Titan performs orders of magnitude faster than the state-of-the-art tools and achieves near-100% verification accuracy on a production global WAN.

Our second work, dShark, mainly targets troubleshooting. dShark is a general and scalable framework for analyzing in-network packet traces collected from distributed devices. With dShark, operators can quickly express their intended analysis logic without worrying about scaling and some practical challenges including header transformations and packet capturing noise.

Our third work, Simon, focuses on offering network operators a general and flexible programming model on top of a stream of network events. With this model, operators can probe the network behavior with queries interactively or compose scripts for repetitive tasks, monitoring of invariants. We present the design of Simon and discuss its implementation and use.

Towards a Reliable and Predictable Network

by

Da Yu

B. Eng., Sun Yat-sen University, 2011

M. Eng., Peking University, 2013

M. Sc., Brown University, 2015

A dissertation submitted in partial fulfillment of the

requirements for the Degree of Doctor of Philosophy

in the Department of Computer Science at Brown University

Providence, Rhode Island

Build time: Tuesday 26th March, 2019 (22:44)

This dissertation by Da Yu is accepted in its present form by

the Department of Computer Science as satisfying the dissertation requirement

for the degree of Doctor of Philosophy.

Date ______________	______________________________
	Rodrigo Fonseca, Director

Recommended to the Graduate Council

Date ______________	______________________________
	Theophilus A. Benson, Reader Brown University
Date ______________	______________________________
	Ming Zhang, Reader Alibaba Group

Approved by the Graduate Council

Date ______________	______________________________
	Andrew G. Campbell Dean of the Graduate School

Contents

List of Figures

Chapter 1

Introduction

Thesis Statement Existing network-reliability tools can be categorized into before-the-fact verification tools and during-the-fact network diagnosis tools. Neither of them is able to solely make the network highly available and reliable due to *their goals and scopes*. Integrating these two categories into a continously-working-cycle can fully leverage their strengths and mitigate their weaknesses. Deploying such cycle into practice can drive the network evolves torwards a reliable and predictable state.

1.1 Overview

High reliability has been accepted as an explicit need for today's network management. An ISP outage [78], for example, can cause millions of users to disconnect from the Internet, while a cloud-scale network downtime is very expensive. Recent years, various management tools have been proposed to either ensure network reliability or ease the operators to manage their networks. The state of the art can be mainly classified into two classes: before-the-fact network verification tools [12, 30] and after-the-fact network issue debuggers [103, 104]:

- **Before-the-fact network verification tools** are typically designed to prevent incidents resulting from network configurations errors. These tools, in principle, build a model representing the behaviors of network devices based on their configurations, and then check whether this model meets the properties of interest, such as reachability, device equivalence, and routing loops. This type of solutions have two limitations: 1) they cannot detect issues triggered in runtime, and 2) they are not able to detect the network failures resulting from software and hardware bugs.

- **After-the-fact network issue debuggers** aim to help network operators analyze and locate the root causes of network failures, after a failure occurs. Complementing the before-the-fact and during-the-fact efforts, these debuggers can help the network operators locate the network issues resulting from configuration, software and hardware bugs.

Despite the fact that the state-of-the-art efforts have made significant contributions, they are not able to solely make the network with high availability and management transparency. Fundamentally, their goals and scopes are limited by the class they belong to. For example, before-the-fact network verification prevents service downtime by proactively checking the correctness of network configuration (rather than firmware and software bugs); and after-the-fact network troubleshooting is mainly responsible for locating root causes like bugs after service outages occur.

To steer the network toward a fully reliable and predictable state, instead of improving tools in each of the above categories, recently, there have been proposals [102] to make the verification and troubleshooting constantly work together in a continuous cycle. The cycle starts from verifying network-wide properties with latest network configurations to be pushed, then monitoring the runtime network state with these configurations deployed, locate the root cause once some network issue happens and change configurations in response. However, such a cycle, even equipped with the state-of-the-art tools, can not be used in real-world production networks. This is because these tools fail to take some important realistic challenges into account. For example, it is difficult to correctly model an enterprise-scale network's behavior in practice; however, all the existing network verification tools assume their network models are generated correctly. For another example, handling complex header transformation in real-world is hard, thus making the state-of-the-art network diagnosis efforts impractical to locate deep root causes.

On the other side, existing tools offers different domain specific languages or programming

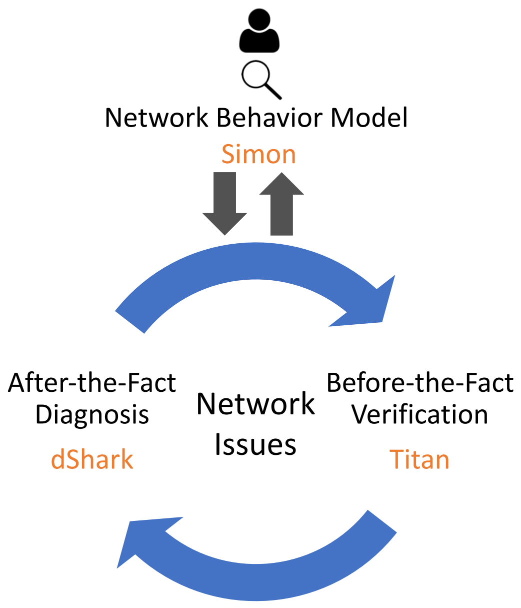


Figure 1.1: The overview of this diseertation.

models. It is challenging to deploy them in a practical environment since network operators are unlikely to manage all of them in a short time. In addition, some of these programming moedels are limited by the assumptions, environements and scopes they have. They fail to meet the diverse requirements from the network operators. For example, Sonata [37] only works in a P4 network.

1.2 Thesis Goals and Contributions

This thesis targets to make such cycle practical. Meanwhile, it aims to provide a simple and general model that makes network operators easier to understand their network and drive their network to a reliable and predictable state. We first presents two tools in this cycle by addressing practical challenges (as shown in Figure 1.1) that limit the real deployment of the state-of-the-art tools. This cycle consists of two individual systems: TITAN, a scalable and faithful BGP configuration verification tool and DSHARK, a general, easy to program and scalable framework for diagnosing network issues. With this cycle, network operators can first verify the network configurations with TITAN before they are really deployed, and then monitor and locate the root causes once some network issues occur with DSHARK. The network operators fix the errors in response and start the cycle over. With multiple iterations of the cycle, the network evolves towards a reliable and predictable state.

On top of the cycle, we offer an intuitive, general and flexible network behavior model. This model regards the running network as a stream of network events. By observing and manipulating this stream, network operators are able to monitor, analyze or even catch network violations. More specifically, network operators can probe this stream to refresh their understandings with the latest network events interactively like traditional software debugging tool (*e.g.*, GDB). Or they can compare current network behaviors with the one in their mind to find violations. Once found, they can write and compose scripts for repetitive tasks, monitoring of variables (*e.g.*, RIB, FIB) to locate the root causes.

1.2.1 Before-the-fact Network Verification

TITAN is a scalable and faithful network configuration verification tool which effectively addresses the scalability and faithfuoness challenges that the state-of-the-art tools facing. It achieves fast and correct verification with failure case coverage in large networks.

Scalability. TITAN offers checks on reachability related properties for both route updates and packets under arbitrary k (link or device) failures in one run. It is a simulation-based verification tool that is orders of magnitude faster than the state-of-the-art tools [12, 31]. It achieves this with a novel technique, *topology condition encoding* (§2.3.3). When simulating the propagation of route updates or packets, TITAN incrementally encodes, with a logical formula, the topologies under which a route update or packet can reach a device, or under which a rule exists in the RIB/FIB. TITAN leverages this encoding for its substantial scalability improvement in two ways. First, TITAN can simultaneously treat all no-more-than-k failure cases by checking if the propagated formula covers these cases, rather than simulating the network for combinatorial times (*e.g.*, Batfish), where most computations are redundant. Second, TITAN traces the process of route update or packet propagation, so that it can cut unnecessary propagation branches whose topology condition is impossible or out of consideration (larger than k failures). This pruning cannot be done by current logical-formula-based tools since the latter do not have the access of the intermediate states during route or packet propagation [12, 31]. Furthermore, because TITAN simulates the route functions rather than representing them as complex logical functions (*e.g.*, Minesweeper), its verification over a simple logical formula is much faster than solving an SMT problem with complex functions.

Faithfulness. To perform its simulation, TITAN needs models of the behavior of routers. Unfortunately, there is significant variation in how real routers implement various aspects of protocols, and, as we found, using a uniform, canonical model of router behavior can produce results which are substantially wrong. We approach this by iteratively specializing router models. TITAN builds a *behavior model tuner* for detecting the flaws of the behavior models in verification. There are two challenges to build such a tuner. First, it is infeasible to validate behavior models of TITAN against the actual device behaviors under arbitrary cases. Instead, our strategy is to validate behavior models under all cases that appear in production. For an existing device SKU, we find all places in production where the SKU sits and validate TITAN's model by comparing the reachability TITAN computes and the one obtained from the real network. For a new device SKU, we build testbeds or emulations [62] to cover all cases it will encounter in production. The second challenge is how to locate the root cause of a reachability mismatch to facilitate model repairs. With existing network monitoring methods, it is possible to localize the root cause to a wrong place because a VSB's impact can only be observed far away from the root cause location (§2.4.2). To discover differences between its model and the real device, TITAN combines all the attributes of a route relevant for routing into an extended RIB. Furthermore, it abstracts each device to a behavior model that contains three stages: "ingress policy", "route selector" and "egress policy". TITAN is able to locate the first place the mismatch happens in the granularity of these stages. As a result, TITAN usually accurately locate a VSB within $O(10)$ configuration lines. After that, developers can easily find the corresponding configuration block and produce patches to improve the verification accuracy.

1.2.2 After-the-fact Network Diagnosis

DSHARK is a scalable packet analyzer that allows for the analysis of in-network packet traces in near real-time and at scale. DSHARK provides a streaming abstraction with flexible and robust grouping of packets: all instances of a single packet at one or multiple hops, and all packets of an aggregate (*e.g.*, flow) at one or multiple hops. DSHARK is robust to, and hides the details of, compositions of packet transformations (encapsulation, tunneling, or NAT), and noise in the capture pipeline. DSHARK offers flexible and programmable parsing of packets to define packets and aggregates. Finally, a query (*e.g.,* is the last hop of a packet the same as expected?) can be made against these groups of packets in a completely parallel manner.

The design of DSHARK is inspired by an observation that a general programming model can describe all the typical types of analysis performed by our operators or summarized in prior work [103]. Programming DSHARK has two parts: a declarative part, in JSON, that specifies how packets are parsed, summarized, and grouped, and an imperative part in C++ to process groups of packets. DSHARK programs are concise, expressive, and in languages operators are familiar with. While the execution model is essentially a windowed streaming map-reduce computation, the specification of programs is at a higher level, with the 'map' phase being highly specialized to this context: DSHARK's parsing is designed to make it easy to handle multiple levels of header transformations, and the grouping is flexible to enable many different types of queries. As shown in §3.3, a typical analysis can be described in only tens of lines of code. DSHARK compiles this code, links it to DSHARK's scalable and high-performance engine and handles the execution. With DSHARK, the time it takes for operators to start a specific analysis can be shortened from hours to minutes.

DSHARK's programming model also enables us to heavily optimize the engine performance and ensures that the optimization benefits all analyses. Not using a standard runtime, such as Spark, allows DSHARK to integrate closely with the trace collection infrastructure, pushing filters and parsers very close to the trace source. We evaluate DSHARK on packet captures of production traffic, and show that on a set of commodity servers, with four cores per server, DSHARK can execute typical analyses in real time, even if all servers are capturing 1500B packets at 40Gbps line rate. When digesting faster capturing or offline trace files, the throughput can be further scaled up nearly linearly with more computing resources.

We summarize our contributions as follows: 1) DSHARK is the first general and scalable software framework for analyzing distributed packet captures. Operators can quickly express their intended analysis logic without worrying about the details of implementation and scaling. 2) We show that DSHARK can handle header transformations, different aggregations, and capture noise through a concise, yet expressive declarative interface for parsing, filtering, and grouping packets. 3) We show how DSHARK can express 18 diverse monitoring tasks, both novel and from previous work. We implement and demonstrate DSHARK at scale with real traces, achieving real-time analysis throughput.

1.2.3 Network Behavior Model

SIMON is a scriptable, interactive network monitor equipped with a simple, general and flexible network behavior model. SIMON has visibility into data-plane events (*e.g.*, packets arriving at and being forwarded by switches), control-plane events (*e.g.*, OpenFlow protocol messages), north-bound API messages (communication between the controller and other services;), and more, limited only by the monitored event sources. Since SIMON is interactive, users can use these events to iteratively refine their understanding of the system at SIMON 's debugging prompt, similar to using traditional debugging tools. Moreover, SIMON does not presume the user is knowledgeable about the intricacies of the controller in use.

The downside of an interactive debugger is that its use can often be repetitious. SIMON is thus scriptable, enabling the automation of repetitive tasks, monitoring of invariants, and much else. A hallmark of SIMON is its reactive scripting language, which is embedded into Scala (scala-lang.org). As we show in Chapter 4, reactivity enables both power and concision in debugging. Furthermore, having recourse to the full power of Scala means that SIMON enables kinds of stateful monitoring not supported in many other debuggers. In short, SIMON represents a fundamental shift in how we debug networks, by bringing ideas from software debugging and programming language design to bear on the problem.

Chapter 2

TITAN: Scalable and Faithful BGP Configuration Verification

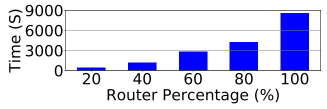
We propose TITAN, the first scalable and faithful BGP configuration verification tool that can be deployed in a production level global WAN. The WAN supports various types of online services. This is a joint work with a major online service provider.

2.1 Background and Motivation

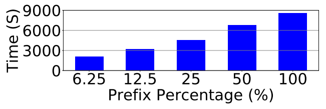
In this section, we first introduce our WAN and the motivations to use configuration verification. We then present the challenges we met when using existing tools on our WAN.

2.1.1 A brief introduction of our WAN

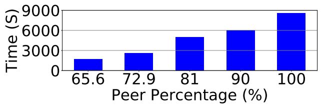
We have a global-scale WAN to support our various types of online services, including cloud services, web services, online video, searching, and so forth. This WAN connects multiple data centers and edge sites globally. The WAN has two tiers: local backbones and a global backbone. In the local tier, metropolitan area networks (MANs) are deployed to interconnect data centers and edge sites in the same city via eBGP; in the global tier, a backbone network is used to connect the MANs via eBGP. Different sites of the backbone network use iBGP on top of IS-IS for exchanging routing information



(a) The impact of #routers.



(b) The impact of #prefixes.



(c) The impact of #peers.

Figure 2.1: Turnaround time for verifying the reachability property with Minesweeper on a subset (6.5%) of our WAN.

and use eBGP to peer with external ISPs. In addition, for policy and security purposes, the MANs contain a large amount of ACL rules on data plane. With the fast growth of the business demands, the size of the WAN is almost doubled each year.

2.1.2 Our need of configuration verification

Correctly configuring a large scale WAN is, unsurprisingly, hard and error-prone. Due to the potential huge impacts on business if network incidents happen on WAN, making sure our configurations are correct before pushing them into production is one of the most important tasks for our network group. In the state-of-art, we have several options to check the network configurations:

Verification vs emulation. Network emulators like CrystalNet [62] enable the network operators to proactively validate network behaviors in a high-fidelity emulated environment. Compared with verification, network emulation has *two* issues which makes the configuration verification irreplaceable. First, network emulation needs vendors to provide their device firmwares within virtual machines or containers, while we found it is practically hard to get such support from all vendors at present. Second, running real switch software requires large amount of computing resources (*e.g.*, $100 per hour for emulating one data center [62]). Since routers in WAN typically have much more sophisticated firmware than data center switches, relying on emulation is extremely expensive. Therefore, we find that a more pragmatic and comprehensive way to validate a network is to first use emulation offline (in low frequency) to validate device firmware, and then use verification online (in high frequency) to validate logic in the configurations.

Configuration verification vs FIB verification. Existing work in network verification falls into two broad categories: configuration verification and FIB verification. Configuration verification checks whether the logic of network configurations meets the network operators' intended properties (*e.g.*, Minesweeper [12], Batfish [31], ERA [28], *etc*). Compared with FIB verification (*e.g.*, NoD [66], Anteater [13], Veriflow [52], HSA [48], *etc*) configuration verification not only offers more comprehensive verification results (*e.g.*, considering failures), but also *proactively* detects configuration errors, before adopting potentially buggy configurations in the network [12]. This 'before-the-fact' validation lead us to prefer the configuration verification approach.

2.1.3 Challenges in existing solutions

As our first attempt, we tried several existing mainstream configuration verification tools on our WAN. Nevertheless, we find that existing solutions can hardly work effectively due to two major practical challenges.

The first challenge is that existing solutions have poor scalability in our WAN. Taking Minesweeper [12] as an example, we ran it on a workstation equipped with 32 CPU cores and 256 GB memory and verified several subnets with different sizes of the WAN. Figure 2.1 shows the time spent to verify a single query on a subnet N1 (only about 6.5% of our entire WAN). As we can see in Figure 2.1a, the time increases exponentially as we increase the fraction of routers within N1, reaching 2.5 hours. Therefore, the total verification time on the entire WAN can easily become intractable, and, to make matters worse, there can be thousands of queries to fully verify a network. There are several specific reasons why Minesweeper is slow in our WAN. First, our WAN contains a large number of IP prefixes, which is expensive to handle in logical formulations [12]. For instance, as shown in Figure 2.1b, the verification time can be reduced if we remove prefixes in N1; second, our WAN has many BGP peering sessions due to the "one-peer-per-link" design strategy for fault tolerance, and Figure 2.1c shows that a large number of peering sessions can inflate the verification time on N1.

The second challenge is that it is hard to get faithful models of network devices due to VSBs, which, in turn, significantly undermines the correctness of the verification results. Specifically, our first attempt deployed the state-of-the-art verification tools in our WAN, but got many incorrect results. For example, Batfish and Minesweeper give incorrect verification results in the VSB case shown in Figure 2.6a. Before we consider the impacts from VSBs, our verification results also have

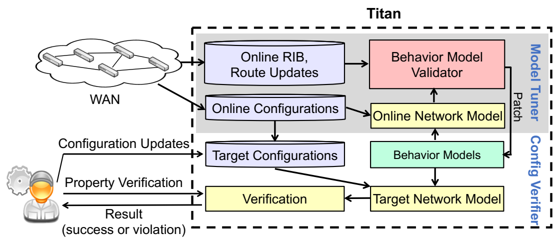


Figure 2.2: TITAN's architecture.

poor accuracy: 79% prefixes have less than 60% accuracy rate compared with the ground-truth (Figure 2.13). Thus, it is hard for a verification tool to get correct results in a complex network consisting of routers from diverse vendors without taking VSBs into account.

It is critical to address the preceding two challenges to deploy a configuration verification system in production, with correctness, failure coverage and scalability in computation.

2.2 Overview

TITAN provides services to network operators for verifying whether a set of configurations have violations on some particular reachability properties or not; meanwhile, it constantly compares the routes it computes from its current device behavior models and online configurations against the routes in real networks, to find flaws in its current behavior model and facilitate the operators to fix these flaws. Therefore, TITAN's architecture has two parts, as shown in Figure 2.2.

In the front-end, the *configuration verifier* (blocks outside the gray area) combines the current online configuration and the proposed configuration changes from operators to generate the target configuration to be verified. According to the SKU of each device of the network, it obtains each device's behavior model and feeds them with the target configuration to generate the target network model. After that, the verification block queries the target network model to answer operator's reachability questions (§2.3).

In the backend, the *behavior model tuner* (components in the gray box) continuously collects online configurations and network information, including routing table (RIB) and route updates. The tuner, on one hand, passes these configurations to each device's behavior model to generate the online

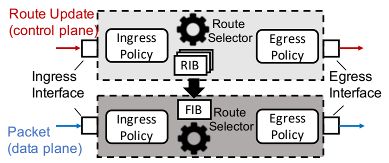


Figure 2.3: A device behavior model.

network model. On the other hand, it calls the validator to check whether there are any mismatches between the model it computes and the network information it collects. When a mismatch is found, the tuner localizes to a small configuration snippet, so that a human can easily understand. Network operators normally write a small patch to fix the flaw in corresponding device models (§2.4). Note that despite Figure 2.2 only showing the comparison against the production network, we augment this with comparisons against testbed and emulation environments.

Next, we explain two important concepts: device behavior model and network model.

2.2.1 Device behavior model

Generally, as shown in Figure 2.3, a device behavior model consists of two pipelines for processing route updates on control plane and packets on data plane respectively (we use the word "message" to refer to a route update or a packet). A concrete device behavior model is generated from the device configuration and the vendor specific behavior modeler of the device type. TITAN builds vendor specific configuration parsers and behavior modelers for all types of devices that could appear in production networks.

Each processing pipeline has three sequential components: ingress policy, route selector and egress policy, as shown in Figure 2.3. Ingress and egress policies are essentially match-action tables that define whether to forward or drop a message and/or how to modify a message based on the pattern of the message. For instance, on the control plane, a BGP router can have an ingress policy which drops route updates from a particular peer, and on the data plane, a router can have an egress ACL rule that drops UDP packets on a particular interface. The route selector encodes the core logic of routing protocols on the control plane or the forwarding logic on the data plane. For instance, in a BGP router, the route selector decides how to prioritize routes from different peers for the same subnet and sends the best route to the peers of this router; on the data plane, it matches a packet to a rule in the FIB based on longest prefix or other built-in logic and selects next hop(s) to forward the packet.

For different vendors and different device SKUs, the specific behavior of ingress and egress policies and the logic inside route selectors can be distinct. Therefore, TITAN must build device behavior models adaptively.

2.2.2 Network model

After obtaining all the behavior models of network devices, TITAN connects them according to the network topology. If two devices have a link between them, their ingress and egress modules in their behavior models are connected together in both directions. The resulting graph is called *network model*.

In a network model, TITAN first makes route announcements from the edges of the network model. Each route update is processed by the control plane pipeline it meets, so that it is propagated across the network model. The RIB of each device is established once the propagation process is done. The FIB then is derived from the RIB. With the complete RIB and FIB, the reachability of a given message can be easily evaluated by combining relevant logical rules along the pipelines the message encounters.

The process of verifying reachability on a network model is similar to existing simulation-based verification tools (*e.g.*, Batfish [31]). Nonetheless, TITAN's uniqueness lies in that it does not mechanically mimic the real networks, but encodes topology conditions along with the messages to construct RIBs and FIBs. This significantly boosts the scalability to verify reachability under failure cases, as we will discuss in next section.

2.3 Configuration Verifier

This section shows how TITAN performs reachability verifications and achieves the tremendous improvement in scalability.

2.3.1 Reachability on control- and data-planes

According to the requirements from operators, TITAN checks reachability of both routes and packets. On one hand, TITAN justifies whether the route to a given subnet can reach a group of network devices after the route propagation process. This is useful for debugging control plane configurations such as BGP peering, routing policies, *etc*. On the other hand, TITAN verifies whether packets towards a given subnet can start from a group of network devices and reach the gateway router of the subnet. Note that "the route of a subnet can reach device A" does not necessarily mean "a packet can reach the subnet's gateway router from A" because of multiple reasons such as data plane ACL rules and longest prefix matching.

2.3.2 Intuitive example of topology condition

We first use an intuitive example to explain the concept of *topology condition encoding*, which is the key to TITAN's good scalability in verification with failure cases.

Whether a device sends or receives a message depends on the up or down status of particular links. Figure 2.4 shows a simple BGP network. We use a tuple (Subnet, AS Path, Nexthop, TopoCond) to denote a BGP update with corresponding topology condition. A topology condition is a logic formula composed by binary link *aliveness* variables (a_n). For example, $a_1 = True$ means Link1 is up. At step ①, C receives a BGP update originated from A, (N, 100, A, a_1), if Link1 is alive. At this step, a_1 is the *topology condition* which must be true for C to get the route update from A. Similarly, at steps ② and ③, the route updates m_2 and m_3 also have their topology conditions: m_3's topology condition is $a_2 \wedge a_3$, as B must first receive the route under a_2 *and* then forward to C under a_3.

After receiving messages (m_1 and m_3), C writes them into its routing table (RIB) (step ④). Rules in the RIB are directly inherited from the topology conditions in the messages. C's RIB has two rules (r_1 and r_2), whose topology conditions are inherited from m_1 and m_3 respectively. Following BGP, C ranks all received routes and only forwards the best to its peer D (and B). In this case, C ranks r_1 higher because it has a shorter AS Path. If r_1 exists, C sends m_4 to D. Otherwise, it sends m_5 (step ⑤). Note m_5's topology condition is the negation of r_1's in conjunction with r_2's. This means under

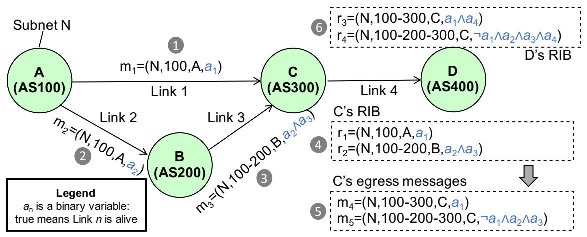

Subnet N
A (AS100)
B (AS200)
C (AS300)
D (AS400)
1
m$_1$=(N,100,A,a_1)
Link 1
Link 2
Link 3
Link 4
2
m$_2$=(N,100,A,a_2)
3
m$_3$=(N,100-200,B,$a_2 \land a_3$)
6
r$_3$=(N,100-300,C,$a_1 \land a_4$)
r$_4$=(N,100-200-300,C,$\neg a_1 \land a_2 \land a_3 \land a_4$)
D's RIB
C's RIB
4
r$_1$=(N,100,A,a_1)
r$_2$=(N,100-200,B,$a_2 \land a_3$)
C's egress messages
5
m$_4$=(N,100-300,C,a_1)
m$_5$=(N,100-200-300,C,$\neg a_1 \land a_2 \land a_3$)
Legend
a_n is a binary variable: true means Link n is alive

Figure 2.4: Route update & RIB with topology conditions.

the case that r_1 does not exist in C's RIB but r_2 does. Finally, D receives m_4 and m_5 and builds its RIB (r_3 and r_4) with the corresponding conditions (step ⑥).

After this route propagation process, it is straightforward to check the route reachability from A to D. For example, the topology condition for D to receive at least one route for Subnet N is r_3's topology condition or r_4's topology condition. It is also easy to find the failure case with the least link failures which causes unreachability from A to D. In this case, failure of Link 4 ($\neg a_4$) makes D unreachable from A.

Note that in this example TITAN has to simulate the propagation of more messages than would be propagated in the network, because of the multiple paths between the source and destinations of a path. In the worst case, there could be an exponential number of such paths. Luckily, in many practical topologies this doesn't happen, and in Section 2.3.6 we detail aggressive pruning rules that keep the problem tractable at the full scale of our WAN.

2.3.3 Topology condition encoding

Topology condition encoding in messages . Formally, given a route update m, we use a logical formula ($I(m, A)$) to decode the topology condition that message (m) reaches the ingress pipeline of device A. Similarly, we use $E(m, A)$ to refer to the topology condition that message (m) is sent to the egress pipeline of device A. $I(m, A)$ and $E(m, A)$ are composed by the link aliveness variables.

Topology condition encoding in RIB/FIB . In real networks, a device will select routes from what it receives to build up its RIB. From Figure 2.4 example, we have two observations: (i) the ranking of the routes only depends on the properties of the routes, independent with the topology conditions; and (ii) a lower ranked route will be selected if the ones in higher ranks are missing in the device. Therefore, we extend the real world RIB to have all received routes of a device with ranking and their corresponding topology condition.

In Figure 2.4, one implicit condition for inserting m_1 and m_3 into C's RIB is that they pass C's ingress policy. The ingress policy only considers the attributes of the route updates and is independent of topology conditions. Therefore, r_1 and r_2 can keep m_1 and m_3's topology conditions respectively if they survive after C's ingress filtering.

In general cases, a rule in RIB is directly derived from a route update. However, in practice, route aggregation, which merges several small subnet routes into a single larger subnet routes, is common. To handle route aggregation, TITAN reads the trigger conditions of route aggregation from configurations and separately handle the cases with and without route aggregation. For instance, if the configuration indicates that routes for "10.0.1.0/32" (with ingress topology condition I_1) and "10.0.1.1/32" (with ingress topology condition I_2) will be aggregated to a single route for "10.0.1.0/31" once both of them are received. We put following rules in the RIB:

$$\begin{aligned} r_{agg} &= (10.0.1.0/31, *, *, I_1 \wedge I_2) \\ r_{sub1} &= (10.0.1.0/32, *, *, I_1 \wedge \neg I_2) \\ r_{sub2} &= (10.0.1.1/32, *, *, \neg I_1 \wedge I_2) \end{aligned}$$

Note that r_{agg}, r_{sub1} and r_{sub2} are exclusive with each other in topology conditions. Despite in theory this method to handle route aggregation can lead exponential number of case combinations, we find it works well in practice because operators usually explicitly write a moderate number of aggregation triggering cases in the configuration and forbid automatic route aggregations for the controllability to routes.

2.3.4 The reachability of routes

The reachability of a route r to a network device D means that whether D can receive a route r ultimately when the control plane protocols have converged. The operators can specify a particular route, *e.g.*, ("10.0.1.0/31", 100-200-300, C), or a pattern representing a group of routes, *e.g.*, ("10.0.1.0/31",

*, *) which means any route to subnet "10.0.1.0/31" no matter the AS path or the next-hop, to verify the reachability.

In TITAN, the key step to verify route reachability under failures is to derive topology conditions for each route update and each rule in RIB/FIB.

Iteratively deriving topology conditions of routes . We use three simple rules to iteratively derive the topology conditions of all route updates, rules in RIB and FIB:

(i) *From RIB to egress*: Suppose r^i denotes a rule in RIB and $r^1, \ldots, r^{i-1}$ are rules to the same destination subnet with higher priority than r^i. The topology condition to send a route update (m^i) from r^i to egress policy pipeline is $\neg R(r^1) \wedge \ldots \wedge \neg R(r^{i-1}) \wedge R(r^i)$, in which $R(r)$ is the topology condition of rule r in the RIB. For example, in Figure 2.4, the topology condition of m_5 is computed according to this rule. Note that one implicit condition for the transition from a RIB rule to a route update is that the route selector decides to send a route to the egress interface.

(ii) *From egress to other side's ingress*: For a route update m with topology condition $E(m, S)$ in S's egress, the topology condition for m to reach D's ingress (D is a peer of S) is $I(m, D) = E(m, S) \wedge a_l$ where l is the link from S to D. For example, in Figure 2.4, the topology condition of m_3 is derived from m_2 based on this rule. Note that one implicit condition for the transition from egress to otherside's ingress is that m passes the egress policy of S.

(iii) *From ingress to RIB*: For a route update m with topology condition $I(m, D)$ in D's ingress, the route selector applies a route sorting algorithm to put m into RIB as rule r^i with topology condition $I(m, D)$ if no route aggregation is triggered. For instance, in Figure 2.4, r_1 and r_2's topology conditions are from m_1 and m_3. Otherwise, route aggregation will be applied and new rules will be available in RIB hereafter as described earlier. Again, one implicit condition for the transition from ingress to RIB is that m passes D's ingress policy.

With the preceding three simple rules, starting from the configured routes (with True topology condition) in the RIBs of the gateway routers of all subnets, TITAN can iteratively derive the route updates and the RIBs with their corresponding topology conditions. The whole route propagation process ends when no router has any new route updates to send.

During the route propagation process, one critical issue is to handle "late higher priority routes".

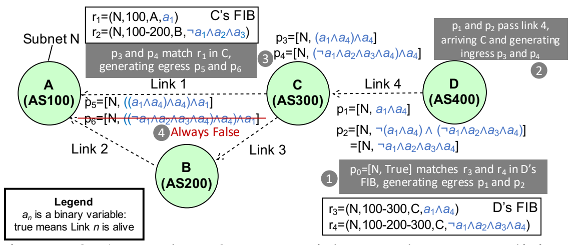

r_1=(N,100,A,a_1)
C's FIB
r_2=(N,100-200,B,$\neg a_1 \wedge a_2 \wedge a_3$)
Subnet N
p_3 and p_4 match r_1 in C, generating egress p_5 and p_6
3
p_3=[N, $(a_1 \wedge a_4) \wedge a_4$]
p_4=[N, $(\neg a_1 \wedge a_2 \wedge a_3 \wedge a_4) \wedge a_4$]
p_1 and p_2 pass link 4, arriving C and generating ingress p_3 and p_4
2
A (AS100)
Link 1
C (AS300)
Link 4
D (AS400)
p_5=[N, $((a_1 \wedge a_4) \wedge a_4) \wedge a_1$]
~~p_6=[N, $((\neg a_1 \wedge a_2 \wedge a_3 \wedge a_4) \wedge a_4) \wedge a_1$]~~
4 Always False
p_1=[N, $a_1 \wedge a_4$]
p_2=[N, $\neg(a_1 \wedge a_4) \wedge (\neg a_1 \wedge a_2 \wedge a_3 \wedge a_4)$]
=[N, $\neg a_1 \wedge a_2 \wedge a_3 \wedge a_4$]
Link 2
Link 3
B (AS200)
1
p_0=[N, True] matches r_3 and r_4 in D's FIB, generating egress p_1 and p_2
Legend
a_n is a binary variable: true means Link n is alive
r_3=(N,100-300,C,$a_1 \wedge a_4$)
D's FIB
r_4=(N,100-200-300,C,$\neg a_1 \wedge a_2 \wedge a_3 \wedge a_4$)

Figure 2.5: Packet & FIB with topology conditions.

Specifically, topology condition of a route update generated from a low priority RIB rule depends on high priority rules. Therefore, if a lower prioritized rule arrives at a device first, it is possible that it is announced with an incorrect topology condition. The solution to this issue is to track the propagation of each route and make an adjustment to topology condition of the lower priority route by "anding" the negation of the new, higher prioritized rule's topology condition. Then we announce the new rule with its new topology condition.

Computing route reachability under failures . Obviously, when the route propagation converges, it is easy to see whether a subnet exists in the RIB of a device. Moreover, because every RIB rule in TITAN has a topology condition, operators can also check whether there exists a failure case that makes routes unreachable to a device under k failure links. If there exists, we can conclude that the reachability is not resilient with up to k link failures. Therefore, given the candidate rules $r_1, \ldots, r_n$ and their topology conditions $R(r_1), \ldots, R(r_n)$, the topology condition which makes at least one rule exist is $V = R(r_1) \vee \ldots \vee R(r_n)$. By leveraging logic solver (*e.g.*, Z3), we can easily get the minimum number of False variables to make V be False [5, 6]. For instance, in Figure 2.4, the topology condition for D to receive at least one route to subnet N is $V = (a_1 \wedge a_4) \vee (\neg a_1 \wedge a_2 \wedge a_3 \wedge a_4)$. We can see that if a_4 is False, V will be False.

2.3.5 The reachability of packets

Operators also need to judge the reachability of a packet with a destination from a group of network devices to the gateway of the destination subnet. Although, as we mentioned, route reachability does not mean packet reachability, route reachability to a destination is the necessary condition of packet reachability. Hence, the first step to verify packet reachability is to finish the route propagation process and generate FIB from RIB in each device. As we presented in §2.3.3, each FIB rule takes the topology condition from its corresponding RIB rule. Figure 2.5 shows the packet propagation process with topology conditions. We omit packets from C to A through B due to space limit. As we can see, every device's FIB inherits its RIB's topology conditions as in Figure 2.4.

Iteratively deriving topology conditions of packets . Similar to a route update, a packet also has a topology condition to reach the ingress or the egress of a device behavior model's data plane pipeline. The topology condition of packets can also be derived iteratively with two simple rules:

(i) *From FIB to egress*: Suppose $r^1, \ldots, r^n$ are rules that match a packet p's destination, and they are ranked from high priority to low [1]. p will hit the highest ranked rule and get forwarded to the rule's nexthop – if it is an ECMP rule with multiple nexthops, each nexthop will have a copy of p. Suppose $I(p, S)$ is the topology condition for p to enter S's ingress, and $R(r^i, S)$ is the topology condition of r^i for existing in S, the topology condition for p to enter S's egress interface indicated by r^i is $E(p, S)^i = I(p, S) \wedge \neg(R(r^1, S) \vee \ldots \vee R(r^{i-1}, S)) \wedge R(r^i)$. For example, at Step ① in Figure 2.5, a packet p_0 from D to subnet N can hit two rules in D's FIB and p_1 and p_2 represent p_0 with different topology conditions by hitting different rules.

(ii) *From egress to otherside's ingress*: For a packet p with topology condition $E(p, S)$ in S's egress, the topology condition for p to reach nexthop D's ingress is $I(p, D) = E(p, S) \wedge a_l$ where l is the link from S to D. For example, at step ② in Figure 2.5, p_3 and p_4 are generated by this rule.

FIB's topology condition is fixed during the packet propagation, since packets, unlike route updates, cannot change RIB/FIB in routers. Hence, similar to Step ①, p_3 and p_4 hit r_1 in C, generating p_5 and p_6 at step ③. Note that in TITAN a packet can also be symbolic. The only difference is that topology conditions are attached to each packet branch during symbolic execution. We omit the details due to space limit.

Computing packet reachability under failures . After the packet propagation process, there can be multiple copies of the packet with different topology conditions reaching the gateway of the destination subnet. Similarly, we can combine all topology conditions and check whether there exists

[1]There can be rules with different subnet granularity matching the destination. Based on the longest prefix matching strategy, we first rank rules based on their subnet granularity and put smaller subnet granularity to higher priority. We keep the rank of the rules in the same subnet as what they are in the RIB.

a failure case with less than k failures which eliminates the reachability of the packet.

2.3.6 Optimizations for scalability

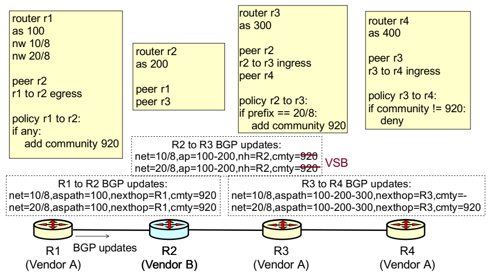
One potential concern is that tracing topology conditions in the route or packet propagation could grown exponentially, due to logic implications of multiple paths, route aggregations, or ECMP. However, we take the following strategies that make TITAN scale substantially. Section 2.6.1 shows the effectiveness of these strategies at the scale of our full WAN.

Dropping more-than-k-failure conditions : If the topology condition of a route has already contained more than k negation of link aliveness, the route will be dropped because we only care about the failure cases with no more than k link failures. Since k is typically small (*e.g.*, k=3) compared with the total number of links, this strategy can significantly reduce the number of propagation branches to explore.

Dropping impossible conditions : It is also easy to judge whether a formula is always False for further pruning. For example, in Figure 2.5, we stop considering p_6 at step ④ because its topology condition is always False.

Simplifying condition formulas : There are at most the number of links variables in a topology condition formula. In addition, a route update or a packet usually only pass through a small number of links. Hence, the number of independent variables in a topology condition is typically small, so that a topology condition can be simplified to a short formula. We can archive great memory efficiency from such simplification.

Another concern in the pruning strategies is the impact of “late high ranked rules”. Specifically, if we announce a low ranked rule first and make the amendment to its topology condition later, whether the pruning decisions we have already made before are still valid. Fortunately, the answer is yes: suppose r is the low ranked rule, and $R(r, D)$ is its old topology condition when it is first announced. Later, after the high ranked route r' comes, r’s topology condition will be modified to $R(r, D) \wedge \neg R(r', D)$. If r has already been dropped at D, $R(r, D)$ has either more than k negative variables or is always False. Modifying $R(r, D)$ to $R(r, D) \wedge \neg R(r', D)$ can neither reduce the number of negative variables nor make $R(r, D) \wedge \neg R(r', D)$ be True (if $R(r, D)$ is False). Therefore,



```
router r1
as 100
nw 10/8
nw 20/8

peer r2
r1 to r2 egress

policy r1 to r2:
if any:
   add community 920
```

```
router r2
as 200

peer r1
peer r3
```

```
router r3
as 300

peer r2
r2 to r3 ingress
peer r4

policy r2 to r3:
if prefix == 20/8:
   add community 920
```

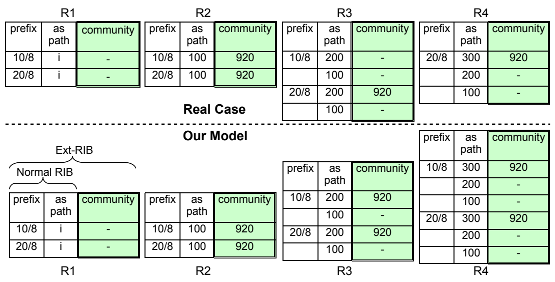
```
router r4
as 400

peer r3
r3 to r4 ingress

policy r3 to r4:
if community != 920:
   deny
```

```
R1 to R2 BGP updates:
net=10/8,aspath=100,nexthop=R1,cmty=920
net=20/8,aspath=100,nexthop=R1,cmty=920
```

R2 to R3 BGP updates:
net=10/8,ap=100-200,nh=R2,cmty=~~920~~
net=20/8,ap=100-200,nh=R2,cmty=~~920~~

```
R3 to R4 BGP updates:
net=10/8,aspath=100-200-300,nexthop=R3,cmty=-
net=20/8,aspath=100-200-300,nexthop=R3,cmty=920
```

(a) Routers and their configurations.



Real Case
Our Model
Ext-RIB
Normal RIB

Real Case

R1

prefix	as path	community
10/8	i	-
20/8	i	-

R2

prefix	as path	community
10/8	100	920
20/8	100	920

R3

prefix	as path	community
10/8	200	-
	100	-
20/8	200	920
	100	-

R4

prefix	as path	community
20/8	300	920
	200	-
	100	-

Our Model

R1

prefix	as path	community
10/8	i	-
20/8	i	-

R2

prefix	as path	community
10/8	100	920
20/8	100	920

R3

prefix	as path	community
10/8	200	920
	100	-
20/8	200	920
	100	-

R4

prefix	as path	community
10/8	300	920
	200	-
	100	-
20/8	300	920
	200	-
	100	-

(b) Comparing ext-RIB in real and our model.

Figure 2.6: A simplified real example of a latent VSB: Vendor A does not remove the community number from its BGP updates by default, while Vendor B does. Our model follows Vendor A's behavior.

the pruning decisions remains valid after topology condition amendments.

2.3.7 Link state protocols and redistribution

The preceding reachability verification process is designed for distance or path vector routing protocols like BGP. Besides BGP, the backbone network in our WAN is a single big AS, which uses IS-IS to distribute routes to loopback IPs of BGP routers to establish iBGP sessions. We use a graph based method like ARC [34] to first verify loopback IP reachability over IS-IS under k failures. If the IS-IS configurations fail to pass the verification, operators need to fix the errors in IS-IS first. Otherwise, all the route redistributions from IS-IS will be treated as static routes with initial topology condition set as True. Because IS-IS has already passed the reachability verification under k failures, we simply assume iBGP sessions are always up during the BGP verification process.

2.4 Behavior Model Tuner

In §2.3, we see the faithfulness of network models serves as the foundation of verification correctness: even a small logic flaw in network model is likely to cause incorrect results (see §2.5). In this section, we describe how TITAN leverages its device behavior model tuner to continuously detect flaws caused by VSBs and achieve high fidelity verification results.

2.4.1 Building faithful behavior models is hard

To build a faithful behavior model for a device, one should master the configuration language of the device's vendor and how the device's firmware implements the behaviors indicated in configurations. However, the second requirement is particularly hard due to several pragmatic challenges.

Standards can be ambiguous . RFC's specification usually defines protocol behaviors with ambiguous words like "should" or "may", causing different vendors to implement the corresponding configuration policy in diverse ways.

Standards can be incomplete . Ideally, RFCs should be comprehensive enough to cover all the cases of protocol execution; however, the fact is not. Due to the incompleteness of RFC specification, vendors define their own default behaviors based on their own safety concerns and understandings.

Vendor's implementation can be incomplete . Furthermore, despite RFCs define a large number of protocol behaviors, the majority of vendors only realize them partially, and different vendors might choose different parts in implementation.

Because of the preceding pragmatic challenges, we design the behavior model tuner in TITAN as a systematic and automatic way to build behavior models in a vendor-specific manner and find and fix logic flaws in the behavior models.

2.4.2 Behavior model tuner

We have to use black-box testing methodologies to compare the behaviors of our model with real devices, since vendors typically do not share the details of their implementation in all aspects and cases. The basic idea to find flaws in current device behavior models is to compare whether the model can have the same output with the same input as a real device in different network environments. Nevertheless, there are several critical challenges to realize this simple idea:

Unpredictable VSB areas . Despite from historical experiences, operators know some areas are likely to have VSBs like `remove-private-as`, new VSB areas are being discovered now and then in practice. Therefore, we cannot just make a list of all potential VSB areas to check proactively. Instead, we need to broadly check the behavior models and find new VSB areas. Hence, we try to create various environments to check our models rather than constructing several special cases for particular VSBs. We also try to localize the root cause if a mismatch between our model and real devices is found rather than assuming it is caused by any known VSBs.

Coverage of comparison cases . A device needs a context to perform a behavior. For instance, a BGP router needs to receive proper routes to perform route aggregation. Or it needs to get updates with private ASes in the path to perform `remove-private-as`. Even for a single type of device and a single route protocol, there are too many cases to cover in general to fully validate whether a device behavior model matches the actual behavior of real devices.

Our pragmatic strategy to address the coverage issue is to make sure we first cover all cases that a device faces in production. Therefore, the behavior model validator (Figure 2.2) keeps monitoring the online configuration and the route propagation of the production network and continuously compares the route propagation it computes from current behavior model with the real network. Alerts are raised if a difference is detected. Then the behavior model validator locates the root cause of this difference.

This strategy works surprisingly well in practice. One reason is that our WAN is complex, so that the major types of devices can have various deployment scenarios, which allows their behavior models to be validated with enough diversity over time. Besides, for new type of devices that have not been deployed in production yet, we rely on testbed or emulation to construct the deployment scenarios of the devices. Fortunately, the number of deployment scenarios for new device is typically small for the common "gradual upgrade" purpose.

Localization of root causes . First, it is possible to localize root causes at a wrong place if we merely use existing monitoring methods and data to detect mismatches. Because a VSB's impact might only be observed far from the real root cause place. For instance, in Figure 2.6a, R2 (Vendor B) by default removes communities from the BGP updates it sends, while other routers (Vendor A) do not. Figure 2.6b shows the RIBs of the four routers. Prefix 10/8 in R4's RIBs has a mismatch, but R3's are identical. Naturally, one might think the root cause is R3 or R4. However, it is R2, which can only be discovered if we look at the community of each route in R2 and R3's RIBs. In order to accurately locate the root cause, we create a concept called "extended RIB (ext-RIB)" which includes

VSB	Description	Affected dev.	# patch-lines
default ACL	Whether permitting data packets that match no explicit ACL.	87.5%	40
default route policy	Whether accepting route updates that match no explicit policy.	82.83%	39
(ext) community	Whether including (extended) communities in route updates by default.	63.91%	46
route redistribution	Whether redistributing default route (0.0.0.0/0).	13.26%	30
AS loop	Whether allowing AS number repetitions in AS Path.	8.63%	26
remove private AS	Whether removing all private ASes from AS path.	7.38%	66
self-next-hop	Whether using self as next-hop when announcing iBGP updates to VPN peers.	6.52%	13
local AS	Whether adding new AS into AS path during AS migration.	1.32%	17

Table 2.1: Detected VSBs and their impacts.

all attributes of each route that can make impacts in route selection. We compare ext-RIBs rather than real RIBs directly from devices for root cause localization.

Second, even with ext-RIBs, some VSBs are still latent. For example, in Figure 2.6b, ext-RIBs for prefix 20/8 are identical in all routers. If 10/8 does not exist, the only way to detect the VSB is in the route updates from R2 to R3 (Figure 2.6a); thus, besides ext-RIB, we also collect route updates received by the routers and use BGP monitoring protocol [98] to check the process details. This enables the tuner to precisely locate VSBs between ingress policy and route selector.[2] After that, our operators write patches embedded in corresponding device behavior models to make them faithful. All of these methodologies are necessary to accurately locate the root causes, since VSBs are prevalent in all steps of route processing.

Scalability of model validation . Comparing all IP prefixes' propagation process is not traceable in our network. We split configurations into blocks that each presents a single policy or behavior. We then build an automatic way to suggest moderate number of prefixes that can cover most configuration blocks, similar to the "equivalent class" idea in ATPG [118].

Deployment status . Now, we have deployed the behavior model tuner for BGP on control plane. We plan to apply similar ideas to other control plane protocols and data plane. Current deployment finds not only model flaws caused by VSBs, but also software bugs in the model development.

[2]The information between route selector and egress is still under development

2.5 Deployment Experience

TITAN has been deployed in production and used in daily basis on our WAN. In this section, we present VSBs and configuration errors TITAN found in real world.

2.5.1 Real world vendor-specific behaviors

Table 2.1 lists some major VSBs TITAN found in production which significantly influence the results of configuration verification. The column of "Affected dev." shows the fraction of devices that potentially face this VSB, and "# patch-lines" means the lines of code to patch the behavior model. We pick some example VSBs from Table 2.1 to explain their impact.

Default policy . "default ACL" and "default route policy" affect the most of devices in our WAN. Both of them refer to the vendor's default action (permit or deny). If a route update (or packet) does not match any explicit route policy or ACL rule in a device, permitting or denying this route update (or packet) is totally up to this device's vendor default action.

Community in BGP updates . The VSB example in Figure 2.6 is the "(ext) community" VSB. Some vendors drop the extended communities at default, while others keep. Unaware of this type of VSB, a verification tool may generate a RIB different from the real-world one, like R4's RIB in Figure 2.6.

AS migration . The configuration of "local AS" is designed for changing a router's AS number (a.k.a. AS migration). At the beginning of an AS migration, operators typically want to keep the old AS number to a router's existing peers for maintaining the BGP session. They configure the old AS number as "local AS". In the BGP updates of the router under migration, some vendors just put the old AS number in the BGP path, but others put both new and old. This affects the length of AS path which is used in many routers as a metric to select the best route. Therefore, this VSB ultimately impacts reachability judgment when verifying an AS migration plan.

iBGP peering via VPN . Normally, if a router A_1 learns a route with next-hop B from its iBGP peer A_2, A_1 uses B as the next-hop of the route in its RIB. However, if A_1 and A_2's iBGP session is on top of VPN, some vendor automatically enables "self-next-hop" on A_2, such that the next-hop A_1

learns is A_2 instead of B. As A_1 may have different control plane or data plane policies to A_2 versus to B, whether enabling the "self-next-hop" also affects the reachability verification results.

2.5.2 Real world configuration errors

TITAN has been used to continuously monitor the correctness of online configurations and verify the configuration updates on our WAN. We now share some representative configuration errors from the hundreds of errors we found.

Wrong attributes in BGP updates. Our operators attach BGP updates with various attributes (*e.g.*, med, community, *etc.*) during the propagation process to meet our network business requirements. For example, at some devices, we filter the community information to make sure they only accept and process the intended BGP routes.

Misconfiguration in changing these attributes (add, modify and remove) is common in our WAN before we develop TITAN. Missing attributes or incorrectly attached attributes may cause routers to make incorrect decisions. With TITAN, our operators significantly prevent such errors by reasoning about the reachability property in our WAN. For example, our operators expect a given router should talk to another, whereas in fact it doesn't. TITAN is able to show such an unreachability with detailed information that includes the propagation path of this route and where and why it is blocked.

Errors caused by incorrect BGP neighbor updates. In our network updates, operations like adding or deleting BGP neighbors are quite common but highly risky—any small error may cause disaster. For example, unwittingly or incorrectly removing a critical BGP node would cause a large amount of traffic to re-route, significantly degrading the service quality.

With TITAN, our operators can first simulate the target network before they apply the updates. Then, they could run TITAN to check whether the prior reachability properties still hold. By doing so, our operators can easily find this type of configuration errors ahead of time. We notice that after using TITAN, the errors resulting from incorrect BGP neighbor updates have been totally prevented.

Configuring device group incorrectly. To reduce the complexity of managing our WAN, many functionally-similar routers—*e.g.*, playing the same roles, locating in the same cluster, and serving for the same service—are configured into device groups. Our network operators push identical

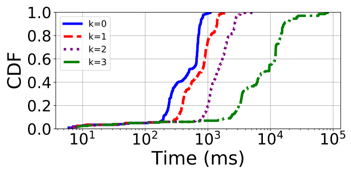


Figure 2.7: Time to simulate one IP prefix with different k.

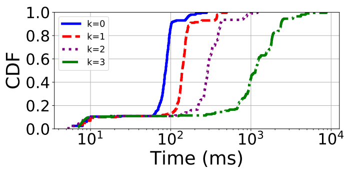


Figure 2.8: Time to verify one IP prefix with system overhead.

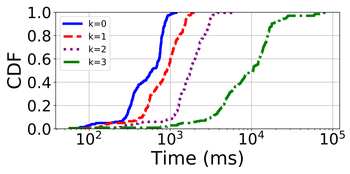


Figure 2.9: Turnaround time to verify one IP prefix.

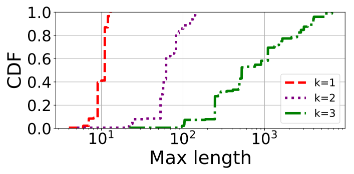


Figure 2.10: Maximum length of the topology condition formula of each prefix.

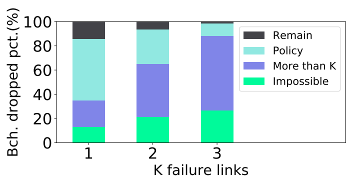


Figure 2.11: Effectiveness of pruning with different k.

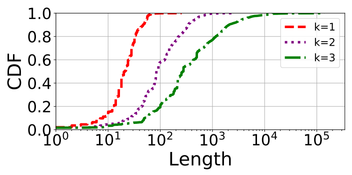


Figure 2.12: Formula length for reachability checking per prefix.

configurations to the routers within the same device group.

Unwittingly or incorrectly adding devices to an incorrect device group may cause severe problems, especially during failures. Our operators have used TITAN to find many “device grouping” errors in our WAN by checking the role equivalence: routers in the same device group should always receive the same route updates and have the same network state.

2.6 Performance Evaluation

In this section, we present some key performance numbers directly from the deployed system to unveil the real running status of TITAN. We also compare the verification performance of TITAN with two state-of-the-art verification tools in conducted experiments. We run all evaluations on a server with Intel(R) Xeon(R) CPU E5-2682 v4 @ 2.50GHz (32 logical cores), 256GB RAM and 1T SSD.

2.6.1 TITAN’s performance in the wild

Verification performance . Figure 2.7 shows the CDF of the time to simulate the propagation process of one IP prefix in TITAN over our entire WAN. Specifically, 98% IP prefixes can be done within one second. When k becomes large (*e.g.*, $k = 3$), the 90-percentile time cost increases to

Network properties		Route reachability	Packet reachability
Reachability	$k = 0$	$481s$	$245s$
	$k = 1$	$770s$	$304s$
	$k = 2$	$1523s$	$715s$
	$k = 3$	$10496s$	$3989s$
Role equivalence		$13s$	-

Table 2.2: Time to verify our entire WAN with TITAN

around 17 seconds.

Once the simulation is done for all prefixes, operators can verify whether network properties are held under normal and failure cases by solving the topology conditions corresponding to each route. Figure 2.8 shows the CDF of the time to verify route reachability queries by the solver. It includes the time to verify the logical formula by the solver and some system overhead (*e.g.*, RPC, data loading and parsing time). We observe TITAN can answer these queries in a quite light-weight way after the initial route propagation. The most complicated query under the failure case ($k = 3$) is answered within 8s. Figure 2.9 shows the end-to-end latency for verifying one IP prefix. Even for $k = 3$ the median is $<$ 10s.

Pruning (§2.3) keeps the logic formula simple in the topology encoding. Figure 2.10 shows the distribution of topology condition formula length. We observe that the maximum formula size in TITAN is only about $O(10,000)$. When $k = 3$, on average, only 2% of conditions survive during the propagation process, as shown in Figure 2.11. 61%, 27% and 10% of conditions are cut due to larger-than-k, impossible condition and policies respectively. As shown in Figure 2.12, once the propagation is done, the longest size of the composed reachability formula for $k = 3$, fed into the solver is 137,078.

In addition to the performance evaluation for individual IP prefixes, we also show the overall performance in real-life verification scenarios. We evaluate the end-to-end verification latency in two scenarios: (i) route reachability verification with network operators' expectations and packet reachability verification for all pairs of devices; (ii) equivalence verification of two devices. TITAN takes 30 seconds on average to load all topology and configuration information. Table 2.2 shows it can finish all-pair packet reachability verification around 4 hours even with $k = 3$. The device equivalence verification is super fast, which is about 13 seconds on average.

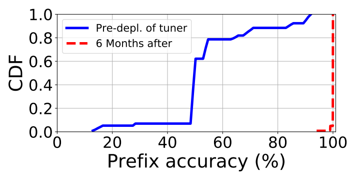


Figure 2.13: Verification accuracy.

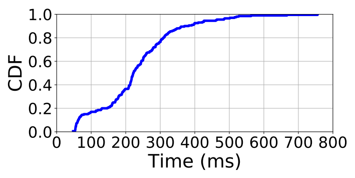


Figure 2.14: Ext-RIB loading time.

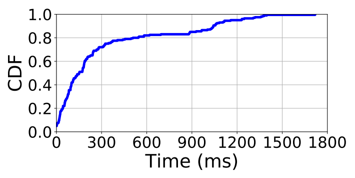


Figure 2.15: VSB localizing time.

Verification accuracy. We measure the verification accuracy by comparing TITAN's outputs with operator's expectation and figuring out the wrong side if a mismatch happens. We define "prefix accuracy" as the fraction of reachability properties (*e.g.*, reach or not reach a target) that is correctly computed by TITAN. Figure 2.13 shows the CDF of the prefix accuracy in our WAN. We observe that before we deploy the behavior model tuner, the accuracy of verification is fairly low: 50% of the prefixes have 50% accuracy or lower. The verification accuracy was significantly boosted 6 months later. After TITAN discovered and fixed many VSBs, 95% of our prefixes have reached 100% accuracy. The remaining inaccuracy is caused by data error or incomplete input fed by the external system, rather than the quality of TITAN's behavior model.

Behavior model tuner performance . To present the overhead of our behavior model tuner, we pick 200 IP prefixes in our WAN. These prefixes not only cover the most important services of our company, but also are active to serve for millions of users. We measure each prefix in details to show the performance and overhead of the behavior model tuner.

Figure 2.14 shows the CDF of the time to load ext-RIB from a real device. In this process, TITAN needs to contact the target router and pull the network state back. As we can see, the loading time is 222 ms and 382 ms in 50% and 90% percentile respectively. Even for the prefix with largest propagation scope, the loading time is less than 800 milliseconds.

We measure the memory cost of this process. Loading ext-RIB for one IP prefix from one device is around 1 KB. The total memory cost for all these 200 prefixes is around 840 MB.

After the ext-RIBs are loaded, TITAN checks and locates VSBs. We evaluate the time cost of this process in Figure 2.15. In most (90%) cases, TITAN takes less than 1 second.

Network properties		TITAN	Minesweeper	Batfish
Reachability	$k = 0$	$3s$	$1555s$	$28s$
	$k = 1$	$4s$	$3573s$	$6299s$
	$k = 2$	$5s$	$4733s$	$> 24h$
	$k = 3$	$14s$	$7430s$	$> 24h$
Role equivalence		$3s$	$203s$	-

Table 2.3: Time comparison in the small subnet.

Network properties		TITAN	Minesweeper	Batfish
Reachability	$k = 0$	$14s$	$84043s$	$683s$
	$k = 1$	$22s$	$> 24h$	$> 24h$
	$k = 2$	$43s$	$> 24h$	$> 24h$
	$k = 3$	$176s$	$> 24h$	$> 24h$
Role equivalence		$4s$	$> 24h$	-

Table 2.4: Time comparison in the medium subnet.

2.6.2 Comparing with existing tools

We compare TITAN with Batfish [31] and Minesweeper [12] on verification performance with conducted experiments.

Experiment setup . Because Batfish and Minesweeper cannot verify our entire WAN due to the scalability issue, we pick two subnets from our WAN—a small one with 20 routers and a medium one with 80 routers—and use the three tools to verify them for performance comparison. We perform two types of reachability verifications. The first is packet reachability. We verify packet reachability between each pair of routers in the network. We use 50 threads to perform verification of different device pairs in parallel for all three approaches. The second is device equivalence. We verify if two selected routers receive the same routes and build the same RIB. We use a single thread in this scenario.

Packet reachability. Table 2.3 shows the result in the small subnet. TITAN takes 3 seconds to finish the reachability verification whereas Minesweeper and Batfish need $1,555$s and 28s respectively. Without considering any link failures, TITAN's RIB and FIB generation process is similar to Batfish. The main reason why TITAN outperforms Batfish so much even when $k = 0$ is that Batfish consumes much time on the constraint solver (like NoD [66]) it relies on.

When considering failure cases with different k, time cost for TITAN are increasing slightly to 14s. The time cost for the other two works increases dramatically. For instance, Minesweeper needs thousands of seconds and Batfish takes more than 24 hours when $k = 3$.

In the medium subnet, TITAN performs orders of magnitude faster than Minesweeper and Batfish. Similar to the performance in the small subnet, TITAN takes less than 3 minutes whereas Batfish and Minesweeper need more than one day under different k as shown in Table 2.4.

We also evaluate the maximum length of the topology condition formula of each prefix in these two networks. Answering the reachability of a given route, in most cases, TITAN needs to solve logic formulas of size 242 and 543 when $k = 3$ in the small and medium subnets respectively while Minesweeper reaches 230,403 and 4,786,577.

The above evaluation results clearly demonstrate the huge advantage of TITAN in scalability to verify network configuration with failure cases.

Device equivalence . Given two devices, TITAN can verify whether they receive the same route updates and build the same RIB/FIB. The equivalence property needs not consider the failure case since verifying this property is not required to be valid under failures. Table 2.3 and Table 2.4 show TITAN takes less than 4s to verify this property whereas Minesweeper needs 203s and $>$ 24 hours respectively. We did not evaluate Batfish since its code does not have this feature.

2.7 Related Work

Early efforts (*e.g.*, rcc [29] and iMinerals [11]) focused on analyzing individual routing protocols' misbehavior (*e.g.*, BGP) to locate the configuration error.

Data plan verification. To verify diverse protocols' misconfiguration, data plane verification approaches were proposed (*e.g.*, NoD [66], Anteater [13], Veriflow [52], HSA [48], Delta-net [42], and SymNet [100]); however, data plane verification work can only check a given data plane snapshot, rather than proactively checking the entire control plane dynamics. Different from the above work, VMN [82] can verify networks with middleboxes; and P-Rex [44, 96] verifies MPLS networks.

Control plane verification. In recent years, configuration verification approaches were proposed to offer control plane-level checking [12, 28, 31, 34, 67, 86, 107, 109]. These solutions can be classified into three categories:

(i) *Simulation-based verification:* Batfish [31], C-BGP [86] and FastPlane [67] take as input network configuration and multiple given environments, simulate control plane, and finally generate the corresponding data planes. They, then, leverage the existing data plane work (mentioned above) to check misconfiguration. Simulation-based efforts suffer from scalability issue to consider failure cases in large networks.

(ii) *Graph-based verification:* Efforts like ARC [34] model configurations as a directed graph, and then use graph-based algorithms to verify the network properties. ARC is fast, but has limitations to encode complicated BGP routing policies.

(iii) *Logical-formula-based verification:* ERA [28] and Minesweeper [12] establish network models based on entire configuration, and then convert network properties verification into BDD and SMT problems, respectively. Bagpipe [109] shares the similar ideas, but only focuses on BGP configuration verification. Although TITAN does not do symbolic route verification, we only target the routes defined in the configurations. This difference is not the main reason that makes TITAN outperform. Compared with TITAN, the above tools struggle on solving large, complex logical formulas that are directly modeled from the control plane; TITAN merely traces the propagation of routes and packets, so that it can prune during the process and only solves an MinSAT problem on a simple and small logic formula.

Configuration synthesis and repair. Network configuration synthesis techniques are complementary to verification. Propane [13] and PropaneAT [14] are focused on BGP configuration synthesis. SyNET [26] and Zeppelin [101] synthesize configuration for general protocols and fault-tolerant router configurations, respectively. NetComplete [27] automatically completes partial configurations. HARC [33] repairs buggy configurations.

Chapter 3

DSHARK: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces

3.1 Motivation

DSHARK provides a scalable analyzer of distributed packet traces. In this section, we describe why such a system is needed to aid operators of today's networks.

3.1.1 Analysis of in-network packet traces

Prior work has shown the value of in-network packet traces for diagnosis [88**?**]. In-network packet captures are widely supported, even in production environments which contain heterogeneous and legacy switches. These traces can be described as the most detailed "logs" of a packet's journey through the network as they contain per-packet/per-switch information of what happened.

It is true that such traces can be heavyweight in practice. For this reason, researchers and practitioners have continuously searched for replacements to packet captures diagnosis, like flow records [21, 22], or tools that allow switches to "digest" traces earlier [38, 76, 103]. However, the former necessarily lose precision, for being aggregates, while the latter requires special hardware

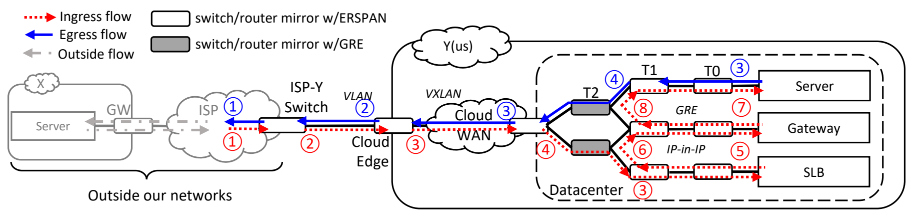


Figure 3.1: The example scenario. We collect per-hop traces in our network (Y and ISP-Y-switch) and do not have the traces outside our network except the ingress and egress of ISP-Y-switch. The packet format of each numbered network segment is listed in Table 3.1.

Number	Header Format										
	Headers Added after Mirroring			Mirrored Headers							
①	ETHERNET	IPV4	ERSPAN	ETHERNET						IPV4	TCP
②	ETHERNET	IPV4	ERSPAN	ETHERNET					802.1Q	IPV4	TCP
③	ETHERNET	IPV4	ERSPAN	ETHERNET		IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
④	ETHERNET	IPV4	GRE			IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑤	ETHERNET	IPV4	ERSPAN	ETHERNET	IPV4	IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑥	ETHERNET	IPV4	GRE		IPV4	IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑦	ETHERNET	IPV4	ERSPAN	ETHERNET		IPV4	GRE		ETHERNET	IPV4	TCP
⑧	ETHERNET	IPV4	GRE			IPV4	GRE		ETHERNET	IPV4	TCP

Table 3.1: The packet formats in the example scenario. Different switch models may add different headers before sending out the mirrored packets, which further complicates the captured packet formats.

support which in many networks is not yet available. Alternatively, a number of tools [7, 36, 93] have tackled diagnosis of specific problems, such as packet drops. However, these also fail at diagnosing the more general cases that occur in practice (§3.2), which means that the need for traces has yet to be eliminated.

Consequently, many production networks continue to employ in-network packet capturing systems [108**?**] and enable them on-demand for diagnosis. In theory, the operators, using packet traces, can reconstruct what happened in the network. However, we found that this is not simple in practice. Next, we illustrate this using a real example.

3.1.2 A motivating example

In 2017, a customer on our cloud platform reported an unexpected TCP performance degradation on transfers to/from another cloud provider. The customer is in the business of providing real-time video surveillance and analytics service, which relies on stable network throughput. However, every few hours, the measured throughput would drop from a few Gbps to a few Kbps, which would last for several minutes, and recover by itself. The interval of the throughput drops was non-deterministic. The customer did a basic diagnosis on their end hosts (VMs) and identified that the throughput drops were caused by packet drops.

This example is representative – it is very common for network traffic to go through multiple different components beyond a single data center, and for packets to be transformed multiple times on the way. Often times our operators do not control both ends of the connections.

In this specific case (Figure 3.1), the customer traffic leaves the other cloud provider, X's network, goes through the ISP and reaches one of our switches that peers with the ISP (①). To provide a private network with the customer, the traffic is first tagged with a customer-specific 802.1Q label (②). Then, it is forwarded in our backbone/WAN in a VXLAN tunnel (③). Once the traffic arrives at the destination datacenter border (④), it goes through a load balancer (SLB), which uses IP-in-IP encapsulation (⑤,⑥), and is redirected to a VPN gateway, which uses GRE encapsulation (⑦, ⑧), before reaching the destination server. Table 3.1 lists the corresponding captured packet formats. Note that beyond the differences in the encapsulation formats, different switches add different headers when mirroring packets (*e.g.*, ERSPAN vs GRE). On the return path, the traffic from the VMs on our servers is encapsulated with VXLAN, forwarded to the datacenter border, and routed back to X.

When our network operators are called up for help, they must answer two questions in a timely manner: 1) are the packets dropped in our network? If not, can they provide any pieces of evidence? 2) if yes, where do they drop? While packet drops seem to be an issue with many proposed solutions, the operators still find the diagnosis surprisingly hard in practice.

Problem 1: many existing tools fail because of their specific assumptions and limitations. As explained in §3.1.1, existing tools usually require 1) full access to the network including end hosts [7, 36]; 2) specific topology, like the Clos [93], or 3) special hardware features [38, 60, 76, 103]. In addition, operators often need evidence for "the problem is not because of" a certain part of the network (in this example, our network but not ISP or the other cloud network), for pruning the potential root causes. However, most of those tools are not designed to solve this challenge.

Since all these tools offer little help in this scenario, network operators have no choice but to enable in-network capturing and analyze the packet traces. Fortunately, we already deployed Everflow [**?**], and are able to capture per-hop traces of a portion of flows.

Problem 2: the basic trace analysis tools fall short for the complicated problems in practice. Even if network operators have complete per-hop traces, recovering what happened in the network is still a challenge. Records for the same packets spread across multiple distributed captures, and none of the well-known trace analyzers such as Wireshark [2] has the ability to join traces from multiple vantage points. Grouping them, even for the instances of a single packet across multiple hops, is surprisingly difficult, because each packet may be modified or encapsulated by middleboxes multiple times, in arbitrary combinations.

Packet capturing noise further complicates analysis. Mirrored packets can get dropped on their way to collectors or dropped by the collectors. If one just counts the packet occurrence on each hop, the real packet drops may be buried in mirrored packet drops and remain unidentified. Again, it is unclear how to address this with existing packet analyzers.

Because of these reasons, network operators resort to developing ad-hoc tools to handle specific cases, while still suffering from the capturing noise.

Problem 3: the ad-hoc solutions are inefficient and usually cannot be reused. It is clear that the above ad-hoc tools have limitations. First, because they are designed for specific cases, the header parsing and analysis logic will likely be specific. Second, since the design and implementation have to be swift (cloud customers are anxiously waiting for mitigation!), reusability, performance, and scalability will likely not be priorities. In this example, the tool developed was single threaded and thus had low throughput. Consequently, operators would capture several minutes worth of traffic and have to spend multiples of that to analyze it.

After observing these problems in a debugging session in production environment, we believe that a general, easy-to-program, scalable and high-performance in-network packet trace analyzer can bring significant benefits to network operators. It can help them understand, analyze and diagnose their network more efficiently.

Group pattern	Application	Analysis logic	In-nw ck. only	Header transf.	Query LOC
One packet on one hop	Loop-free detection[38] *Detect forwarding loop*	*Group:* same packet(ipv4[0].ipid, tcp[0].seq) on one hop *Query:* does the same packet appear multiple times on the same hop	No	No	8
	Overloop-free detection[123] *Detect forwarding loop involving tunnels*	*Group:* same packet(ipv4[0].ipid, tcp[0].seq) on tunnel endpoints *Query:* does the same packet appear multiple times on the same endpoint	Yes	Yes	8
One packet on multiple hops	Route detour checker *Check packet's route in failure case*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches *Query:* is valid detour in the recovered path(ipv4[:].ttl)	No	Yes*	49
	Route error *Detect wrong packet forwarding*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches *Query:* get last correct hop in the recovered path(ipv4[:].ttl)	No*	Yes*	49
	Netsight[38] *Log packet's in-network lifecycle*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches *Query:* recover path(ipv4[:].ttl)	No*	Yes*	47
	Hop counter[38] *Count packet's hop*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches *Query:* count record	No*	Yes*	6
Multiple packets on one hop	Traffic isolation checker[38] *Check whether hosts are allowed to talk*	*Group:* all packets at dst ToR(SWITCH=dst_ToR) *Query:* have prohibited host(ipv4[0].src)	No	No	11
	Middlebox(SLB, GW, etc) profiler *Check correctness/performance of middleboxes*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) pre/post middlebox *Query:* is middlebox correct(related fields)	Yes	Yes	18†
	Packet drops on middleboxes *Check packet drops in middleboxes*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) pre/post middlebox *Query:* exist ingress and egress trace	Yes	Yes	8
	Protocol bugs checker(BGP, RDMA, etc)[123] *Identify wrong implementation of protocols*	*Group:* all BGP packets at target switch(SWITCH=tar_SW) *Query:* correctness(related fields) of BGP(FLTR: tcp[-1].src|dst=179)	Yes	Yes*	23‡
	Incorrect packet modification[38] *Check packets' header modification*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) pre/post the modifier *Query:* is modification correct (related fields)	Yes	Yes*	4$^\diamond$
	Waypoint routing checker[38, 77] *Make sure packets (not) pass a waypoint*	*Group:* all packets at waypoint switch(SWITCH=waypoint) *Query:* contain flow(ipv4[-1].src+dst, tcp[-1].src+dst) should (not) pass	Yes	No	11
	DDoS diagnosis[77] *Localize DDoS attack based on statistics*	*Group:* all packets at victim's ToR(SWITCH=vic_ToR) *Query:* statistics of flow(ipv4[-1].src+dst, tcp[-1].src+dst)	No	Yes*	18
Multiple packets on multiple hops	Congested link diagestion[77] *Find flows using congested links*	*Group:* all packets(ipv4[-1].ipid, tcp[-1].seq) pass congested link *Query:* list of flows(ipv4[-1].src+dst, tcp[-1].src+dst)	No*	Yes*	14
	Silent black hole localizer[77, 123] *Localize switches that drop all packets*	*Group:* packets with duplicate TCP(ipv4[-1].ipid, tcp[-1].seq) *Query:* get dropped hop in the recovered path(ipv4[:].ttl)	No*	Yes*	52
	Silent packet drop localizer[123] *Localize random packet drops*	*Group:* packets with duplicate TCP(ipv4[-1].ipid, tcp[-1].seq) *Query:* get dropped hop in the recovered path(ipv4[:].ttl)	No*	Yes*	52
	ECMP profiler[123] *Profile flow distribution on ECMP paths*	*Group:* all packets at ECMP ingress switches(SWITCH in ECMP) *Query:* statistics of flow(ipv4[-1].src+dst, tcp[-1].src+dst)	No*	No	18
	Traffic matrix[77] *Traffic volume between given switch pairs*	*Group:* all packets at given two switches(SWITCH in tar_SW) *Query:* total volume of overlapped flow(ipv4[-1].src+dst, tcp[-1].src+dst)	No*	Yes*	21

Table 3.2: We implemented 18 typical diagnosis applications in DSHARK. "No*" in column "in-network checking only" means this application can also be done with end-host checking with some assumptions. "Yes*" in column "header transformation" needs to be robust to header transformation in our network, but, in other environments, it might not. "ipv4[:].ttl" in the analysis logic means DSHARK concatenates all ivp4's TTLs in the header. It preserves order information even with header transformation. Sorting it makes path recovery possible. †We profiled SLB. ‡We focused on BGP route filter. $^\diamond$We focused on packet encapsulation.

3.2 Design Goals

Motivated by many real-life examples like the one in §3.1.2, we derive three design goals that we must address in order to facilitate in-network trace analysis.

3.2.1 Broadly applicable for trace analysis

In-network packet traces are often used by operators to identify where network properties and invariants have been violated. To do so, operators typically search for abnormal behavior in the large volume of traces. For different diagnosis tasks, the logic is different.

Unfortunately, operators today rely on manual processing or ad-hoc scripts for most of the tasks. Operators must first parse the packet headers, *e.g.,* using Wireshark. After parsing, operators usually look for a few key fields, *e.g.,* 5-tuples, depending on the specific diagnosis tasks. Then they apply filters and aggregations on the key fields for deeper analysis. For example, if they want to check all the hops of a certain packet, they may filter based on the 5-tuple plus the IP id field. To check more instances and identify a consistent behavior, operators may apply similar filters many times with slightly different values, looking for abnormal behavior in each case. It is also hard to join instances of the same packet captured in different points of the network.

Except for the initial parsing, all the remaining steps vary from case to case. We find that there are four types of aggregations used by the operators. Depending on the scenario, operators may want to analyze 1) each single packet on a specific hop; 2) analyze the multi-hop trajectory of each single packet; 3) verify some packet distributions on a single switch or middlebox; or 4) analyze complicated tasks by correlating multiple packets on multiple hops. Table 3.2 lists diagnosis applications that are commonly used and supported by existing tools. We classify them into above four categories.

dSHARK must be broadly applicable for all these tasks – not only these four aggregation modes, but also support different analysis logic after grouping, *e.g.,* verifying routing properties or localizing packet drops.

3.2.2 Robust in the wild

dSHARK must be robust to practical artifacts in the wild, especially header transformations and packet capturing noise.

Packet header transformations. As shown in §3.1.2, these are very common in networks, due to the deployment of various middleboxes [87]. They become one of the main obstacles for existing tools [77, 103, 123] to perform all of the diagnosis logic (listed in Table 3.2) in one shot. As we can see from the table, some applications need to be robust to header transformations. Therefore, dSHARK must correctly group the packets as if there is no header transformation. While parsing the packet is not hard (indeed, tools like Wireshark can already do that), it is unclear how operators may specify the grouping logic across different header formats. In particular, today's filtering languages are often ambiguous. For example, the “ip.src == X” statement in Wireshark display filter may match different IP layers in a VXLAN-in-IP-in-IP packet and leads to incorrect grouping results. DSHARK addresses this by explicitly indexing multiple occurrences of the same header type (*e.g.*, IP-in-IP), and by adding support to address the innermost ([-1]), outermost ([0]), and all ([:]) occurrences of a header type.

Packet capturing noise. We find that it is challenging to localize packet drops when there is significant packet capturing noise. We define noise here as drops of mirrored packets in the network or in the collection pipeline. Naïvely, one may just look at all copies of a packet captured on all hops, check whether the packet appears on each hop as expected. However, 1% or even higher loss in the packet captures is quite common in reality, as explained in §3.1.2 as well as in [111]. With the naïve approach, every hop in the network will have 1% false positive drop rate in the trace. This makes localizing any real drop rate that is comparable or less than 1% challenging because of the high false positive rate.

Therefore, for DSHARK, we must design a programming interface that is flexible for handling arbitrary header transformations, yet can be made robust to packet capturing noise.

3.2.3 Fast and scalable

The volume of in-network trace is usually very large. DSHARK must be fast and scalable to analyze the trace. Below we list two performance goals for DSHARK.

Support real-time analysis when collocating on collectors. Recent efforts such as [**?**] and [88] have demonstrated that packets can be mirrored from the switches and forwarded to trace collectors. These collectors are usually commodity servers, connected via 10Gbps or 40Gbps links. Assuming each mirrored packet is 1500 bytes large, this means up to 3.33M packets per second (PPS). With high-performance network stacks [1, 91, 111], one CPU core is sufficient to capture at this rate. Ideally, DSHARK should co-locate with the collecting process, reuse the remaining CPU cores and be able to keep up with packet captures in real-time. Thus, we set this as the first performance goal – with a common CPU on a commodity server, DSHARK must be able to analyze *at least* 3.33 Mpps.

Be scalable. There are multiple scenarios that require higher performance from DSHARK: 1) there are smaller packets even though 1500 bytes is the most typical packet size in our production network. Given 40Gbps capturing rate, this means higher PPS; 2) there can be multiple trace collectors [**?**] and 3) for offline analysis, we hope that DSHARK can run faster than the packet timestamps. Therefore, DSHARK must horizontally scale up within one server, or scale out across multiple servers. Ideally, DSHARK should have near-linear speed up with more computing resources.

3.3 DSHARK Design

DSHARK is designed to allow for the analysis of distributed packet traces in near real time. Our goal in its design has been to allow for scalability, ease of use, generality, and robustness. In this section, we outline DSHARK's design and how it allows us to achieve these goals. At a high level, DSHARK provides a domain-specific language for expressing distributed network monitoring tasks, which runs atop a map-reduce-like infrastructure that is tightly coupled, for efficiency, with a packet capture infrastructure. The DSL primitives are designed to enable flexible filtering and grouping of packets across the network, while being robust to header transformations and capture noise that we observe in practice.

3.3.1 A concrete example

To diagnose a problem with DSHARK, an operator has to write two related pieces: a declarative set of trace specifications indicating relevant fields for grouping and summarizing packets; and an imperative callback function to process groups of packet summaries.

Here we show a basic example – detecting forwarding loops in the network with DSHARK. This means DSHARK must check whether or not any packets appear more than once at any switch. First, network operators can write the trace specifications as follows, in JSON:

```
{
  Summary: {
    Key: [SWITCH, ipId, seqNum],
    Additional: []
  },
  Name: {
    ipId:   ipv4[0].id,
    seqNum: tcp[0].seq
  },
  Filter: [
    [eth, ipv4, ipv4, tcp]: {    // IP-in-IP
      ipv4[0].srcIp: 10.0.0.1
    }
  ]
}
```

The first part, "Summary", specifies that the query will use three fields, *SWITCH*, *ipId* and *seqNum*. DSHARK builds a *packet summary* for each packet, using the variables specified in "Summary". DSHARK groups packets that have the same "key" fields, and shuffles them such that each group is processed by the same processor.

SWITCH, one of the only two predefined variables in DSHARK,[1] is the switch where the packet is captured. Transparent to operators, DSHARK extracts this information from the additional header/metadata (as shown in Table 3.1) added by packet capturing pipelines [108**?**].

Any other variable must be specified in the "Name" part, so that DSHARK knows how to extract the values. Note the explicit index "[0]" – this is the key for making DSHARK robust to header transformations. We will elaborate this in §3.3.3.

In addition, operators can constrain certain fields to a given value/range. In this example, we specify that if the packet is an IP-in-IP packet, we will ignore it unless its outermost source IP address is 10.0.0.1.

In our network, we assume that *ipId* and *seqNum* can identify a unique TCP packet without specifying any of the 5-tuple fields.[2] Operators can choose to specify additional fields. However, we recommend using only necessary fields for better system efficiency and being more robust to middleboxes. For example, by avoiding using 5-tuple fields, the query is robust to any NAT that does not alter *ipId*.

The other piece is a query function, in C++:

[1]The other predefined variable is *TIME*, the timestamp of packet capture.

[2]In our network and common implementation, IP ID is chosen independently from TCP sequence number. This may not always be true [105].

```
map<string, int> query(const vector<Group>& groups) {
  map<string, int> r = {{"loop", 0}, {"normal", 0}};
  for (const Group& group : groups) {
    group.size() > 1 ?
      (r["loop"]++) : (r["normal"]++);
  }
  return r;
}
```

The query function is written as a callback function, taking an array of groups and returning an arbitrary type: in this case, a map of string keys to integer values. This is flexible for operators – they can define custom counters like in this example, get probability distribution by counting in predefined bins, or pick out abnormal packets by adding entries into the dictionary. In the end, DSHARK will merge these key-value pairs from all query processor instances by unionizing all keys and summing the values of the same keys. Operators will get a human-readable output of the final key-value pairs.

In this example, the query logic is simple. Since each packet group contains all copies of a packet captured/mirrored by the same switch, if there exist two packet summaries in one group, a loop exists in the network. The query can optionally refer to any field defined in the summary format. We also implemented 18 typical queries from the literature and based on our experience in production networks. As shown in Table 3.2, even the most complicated one is only 52 lines long. For similar diagnosis tasks, operators can directly reuse or extend these query functions.

3.3.2 Architecture

The architecture of DSHARK is inspired by both how operators manually process the traces as explained in 3.2.1, and distributed computing engines like MapReduce [23]. Under that light, DSHARK can be seen as a streaming data flow system specialized for processing distributed network traces. We provide a general and easy-to-use programming model so that operators only need to focus on analysis logic without worrying about implementation or scaling.

DSHARK's runtime consists of three main steps: *parse*, *group* and *query* (Figure 3.2). Three system components handle each of the three steps above, respectively. Namely,

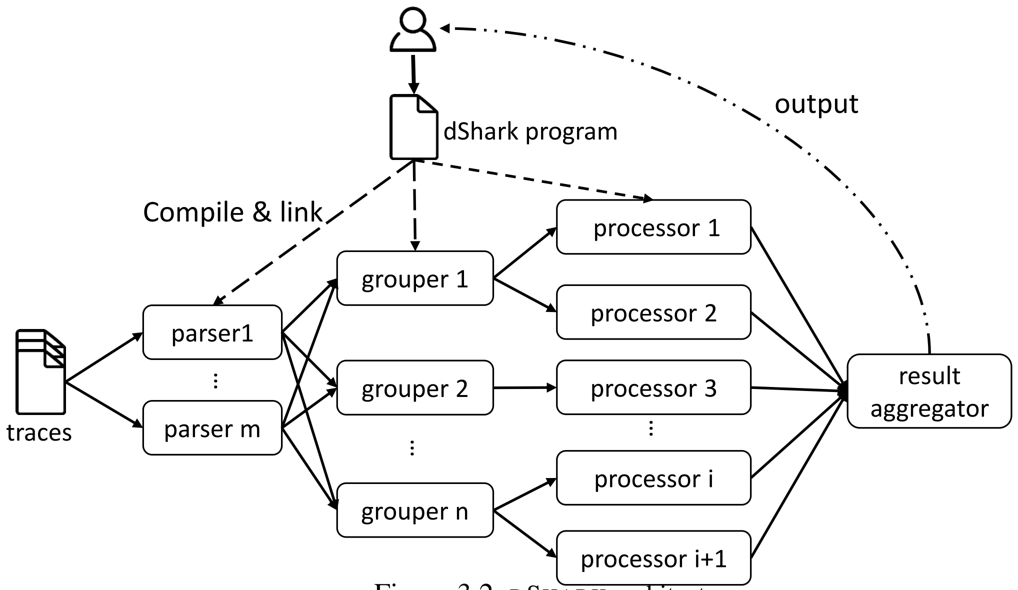


Figure 3.2: DSHARK architecture.

- *Parser:* DSHARK consumes network packet traces and extracts user-defined key header fields based on different user-defined header formats. Parsers send these key fields as packet summaries to groupers. The DSHARK parsers include recursive parsers for common network protocols, and custom ones can be easily defined.
- *Grouper:* DSHARK groups packet summaries that have the same values in user-defined fields. Groupers receive summaries from all parsers and create batches per group based on time windows. The resulting packet groups are then passed to the query processors.
- *Query processor:* DSHARK executes the query provided by users and outputs the result for final aggregation.

DSHARK pipeline works with two cascading MapReduce-like stages: 1) first, packet traces are (mapped to be) parsed in parallel and shuffled (or reduced) into groups; 2) query processors run analysis logic for each group (map) and finally aggregate the results (reduce). In particular, the *parser* must handle header transformations as described in §3.2.2, and the *grouper* must support all possible packet groupings (§3.2.1). All three components are optimized for high performance and can run in a highly parallel manner.

Input and output to the DSHARK pipeline. DSHARK ingests packet traces and outputs aggregated analysis results to operators. DSHARK assumes that there is a system in place to collect traces from the network, similar to [**?**]. It can work with live traces when collocating with trace collectors, or run anywhere with pre-recorded traces. When trace files are used, a simple coordinator (§3.4.4) monitors the progress and feeds the traces to the parser in chunks based on packet timestamps. The final aggregator generates human-readable outputs as the query processors work. It creates a union of the key-value pairs and sums up values output by the processors (§3.4).

Programming with DSHARK**.** Operators describe their analysis logic with the programming interface provided by DSHARK, as explained below (§3.3.3). DSHARK compiles operators' programs into a dynamic-linked library. All parsers, groupers and query processors load it when they start, though they link to different symbols in the library. DSHARK chooses this architecture over script-based implementation (*e.g.,* Python or Perl) for better CPU efficiency.

3.3.3 DSHARK **programming model**

As shown in the above example, the DSHARK programming interface consists of two parts: 1) declarative packet trace specifications in JSON, and 2) imperative query functions (in C++). We design the specifications to be declarative to make common operations like *select*, *filter* and *group* fields in the packet headers straightforward to the operators. On the other hand, we make the query functions imperative to offer enough degrees of freedom for the operators to *define* different diagnosis logic. This approach is similar to the traditional approach in databases of embedding imperative user-defined functions in declarative SQL queries. Below we elaborate on our design rationale and on details not shown in the example above.

"Summary" in specifications. A packet summary is a byte array containing only a few key fields of a packet. We introduce packet summary for two main goals: 1) to let DSHARK compress the packets right after parsing while retaining the necessary information for query functions. This greatly benefits DSHARK's efficiency by reducing the shuffling overhead and memory usage; 2) to let groupers know which fields should be used for grouping. Thus, the description of a packet summary format consists of two lists. The first contains the fields that will be used for grouping and the second of header fields that are not used as grouping keys but are required by the query functions. The variables in both lists must be defined in the "Name" section, specifying where they are in the headers.

"Name" in specifications. Different from existing languages like Wireshark filter or BPF, DSHARK requires an explicit index when referencing a header, *e.g.,* "ipv4[0]" instead of simply "ipv4". This means the first IPv4 header in the packet. This is for avoiding ambiguity, since in practice a packet can have multiple layers of the same header type due to tunneling. We also adopt the Python syntax, *i.e.,* "ipv4[-1]" to mean the last (or innermost) IPv4 header, "ipv4[-2]" to mean the last but one IPv4 header, *etc*.

With such header indexes, the specifications are both robust to header transformations and explicit enough. Since the headers are essentially a stack (LIFO), using negative indexes would allow operators to focus on the end-to-end path of a packet or a specific tunnel regardless of any additional header transformation. Since network switches operate based on outer headers, using 0 or positive indexes (especially 0) allows operators to analyze switch behaviors, like routing.

"Filter" in specifications. Filters allow operators to prune the traces. This can largely improves the system efficiency if used properly. We design DSHARK language to support adding constraints for different types of packets. This is inspired by our observation in real life cases that operators often want to diagnose packets that are towards/from a specific middlebox. For instance, when diagnosing a specific IP-in-IP tunnel endpoint, *e.g.*, $10.0.0.1$, we only care IP-in-IP packets whose source IP is $10.0.0.1$ (packets after encapsulation), and common IP packets whose destination IP is $10.0.0.1$ (packets before encapsulation). For convenience, DSHARK supports "*" as a wildcard to match any headers.

Query functions. An operator can write the query functions as a callback function that defines the analysis logic to be performed against a batch of groups. To be generally applicable for various analysis tasks, we choose to prefer language flexibility over high-level descriptive languages. Therefore, we allow operators to program any logic using the native C++ language, having as input an array of packet groups, and as output an arbitrary type. The query function is invoked at the end of time windows, with the guarantee that all packets with the same key will be processed by the same processor (the same semantics of a shuffle in MapReduce).

In the query functions, each *Group* is a vector containing a number of summaries. Within each summary, operators can directly refer the values of fields in the packet summary, *e.g.,* $summary.ipId$

is *ipId* specified in JSON. In addition, since it is in C++, operators can easily query our internal service REST APIs and get control plane metadata to help analysis, *e.g.,* getting the topology of a certain network. Of course, this should only be done per a large batch of batches to avoid a performance hit. This is a reason why we design query functions to take *a batch of* groups as input.

3.3.4 Support for various groupings

To show that our programming model is general and easy to use, we demonstrate how operators can easily specify the four different aggregation types, which we extend to *grouping* in DSHARK, listed in §3.2.1.

Single-packet single-hop grouping. This is the most basic grouping, which is used in the example (§4.2). In packet summary format, operators simply specify the "key" as a set of fields that can uniquely identify a packet, and from which switch (*SWITCH*) the packet is collected.

Multi-packet single-hop grouping. This grouping is helpful for diagnosing middlebox behaviors. For example, in our data center, most software-based middleboxes are running on a server under a ToR switch. All packets which go into and out of the middleboxes must pass through that ToR. In this case, operators can specify the "key" as *SWITCH* and some middlebox/flow identifying fields (instead of identifying each packet in the single-packet grouping) like 5-tuple. We give more details in §3.5.1.

Single-packet multi-hop grouping. This can show the full path of each packet in the network. This is particularly useful for misrouting analysis, *e.g.,* does the traffic with a private destination IP range that is supposed to stay within data centers leak to WAN? For this, operators can just set packet identifying fields as the key, without *SWITCH*, and use the [-1] indexing for the innermost IP/TCP header fields. DSHARK will group all hops of each packet so that the query function checks whether each packet violates routing policies. The query function may have access to extra information, such as the topology, to properly verify path invariants.

Multi-packet multi-hop grouping. As explained in §3.2.2, loss of capture packets may impact the results of localizing packet drops, by introducing false positives. In such scenarios DSHARK *should* be used with multi-packet multi-hop groupings, which uses the 5-tuple and the sequence numbers as the

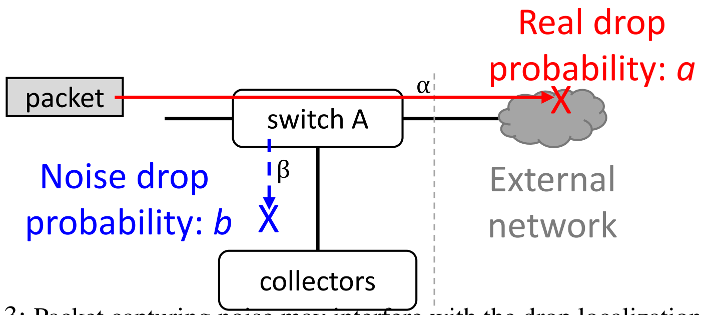


Figure 3.3: Packet capturing noise may interfere with the drop localization analysis.

Case	Probability	w/o E2E info	w/ E2E info
No drop	$(1-a)(1-b)$	Correct	Correct
Real drop	$a(1-b)$	Correct	Correct
Noise drop	$(1-a)b$	Incorrect	Correct
Real + Noise drop	ab	Incorrect	Incorrect

Table 3.3: The correctness of localizing packet drops. The two types of drops are independent because the paths are disjoint after A.

grouping keys, without *ipId*. This has the effect of grouping together transport-level retransmissions. We next explain the rationale for this requirement.

3.3.5 Addressing packet capture noise

To localize where packets are dropped, in theory, one could just group all hops of each packet, and then check where in the network the packet disappears from the packet captures on the way to its destination. In practice, however, we find that the noise caused by data loss in the captures themselves, *e.g.,* drops on the collectors and/or drops in the network on the way to the collector, will impact the validity of such analysis.

We elaborate this problem using the example in Figure 3.3 and Table 3.3. For ease of explanation we will refer the to paths of the mirrored packets from each switch to the collector as β type paths and the normal path of the packet as α type paths. Assume switch A is at the border of our network and the ground truth is that drop happens after A. As operators, we want to identify whether the drop happens within our network. Unfortunately, due to the noise drop, we will find A is dropping packets with probability b in the trace. If the real drop probability a is less than b, we will misblame A. This problem, however, can be avoided if we correlate individual packets across different hops in the network as opposed to relying on simple packet counts.

Specifically, we propose two mechanisms to help DSHARK avoid miss-detecting where the packet was dropped:

Verifying using the next hop(s). If the β type path dropping packets is that from a switch in the middle of the α path, assuming that the probability that *the same* packet's mirror is dropped on two β paths is small, one can find the packet traces from the next hop(s) to verify whether A is really the point of packet drop or not. However, this mechanism would fail in the "last hop" case, where there is no next hop in the trace. The "last hop" case is either 1) the specific switch is indeed the last on the α path, however, the packets may be dropped by the receiver host, or 2) the specific switch is the last hop before the packet goes to external networks that do not capture packet traces. Figure 3.3 is such a case.

Leveraging information in end-to-end transport. To address the "last hop" issue, we leverage the information provided by end-to-end transport protocols. For example, for TCP flows, we can verify a packet was dropped by counting the number of retransmissions seen for each TCP sequence number. In DSHARK, we can just group all packets with the same TCP sequence number across all hops together. If there is indeed a drop after A, the original packet and retransmitted TCP packets (captured at all hops in the internal network) will show up in the group as packets with different IP IDs, which eliminates the possibility that the duplicate sequence number is due to a routing loop. Otherwise, it is a noise drop on the β path.

This process could have false positives as the packet could be dropped both on the β and α path. This occurs with probability of only $a \times b$ – in the "last hop" cases like Figure 3.3, the drops on β and α path are likely to be independent since the two paths are disjoint after A. In practice, the capture noise b is $\ll 100\%$. Thus any a can be detected robustly.

Above, we focused on describing the process for TCP traffic as TCP is the most prominent protocol used in data center networks [8]. However, the same approach can be applied to any other reliable protocols as well. For example, QUIC [58] also adds its own sequence number in the packet header. For general UDP traffic, DSHARK's language also allows the operators to specify similar identifiers (if exist) based on byte offset from the start of the payload.

3.4 DSHARK Components and Implementation

We implemented DSHARK, including parsers, groupers and query processors, in >4K lines of C++ code. We have designed each instance of them to run in a single thread, and can easily scale out by adding more instances.

3.4.1 Parser

Parsers recursively identify the header stack and, if the header stack matches any in the Filter section, check the constraints on header fields. If there is no constraint found or all constraints are met, the fields in the Summary and Name sections are extracted and serialized in the form of a byte array. To reduce I/O overhead, the packet summaries are sent to the groupers in batches.

Shuffling between multiple parsers and groupers: When working with multiple groupers, to ensure grouping correctness, all parsers will have to send packet summaries that belong to the same groups to the same grouper. Therefore, parsers and groupers *shuffle* packet summaries using a consistent hashing of the “key” fields. This may result in increased network usage when the parsers and groupers are deployed across different machines. Fortunately, the amount of bandwidth required is typically very small – as shown in Table 3.2, common summaries are only around 10B, more than 100$\times$ smaller than an original 1500B packet.

For analyzing live captures, we closely integrate parsers with trace collectors. The raw packets are handed over to parsers via memory pointers without additional copying.

3.4.2 Grouper

DSHARK then groups summaries that have the same keys. Since the grouper does not know in advance whether or not it is safe to close its current group (groupings might be very long-lived or even perpetual), we adopt a tumbling window approach. Sizing the window presents trade-offs. For query correctness, we would like to have all the relevant summaries in the same window. However, too large of a window increases the memory requirements.

DSHARK uses a 3-second window – once three seconds (in packet timestamps) passed since the creation of a group, this group can be wrapped up. This is because, in our network, packets that may be grouped are typically captured within three seconds.[3] In practice, to be robust to the noise in packet capture timestamps, we use the number of packets arriving thereafter as the window size. Within three seconds, a parser with 40Gbps connection receives no more than 240M packets even if all packets are as small as 64B. Assuming that the number of groupers is the same as or more than parsers, we can use a window of 240M (or slightly more) packet summaries. This only requires several GB of memory given that most packet summaries are around 10B large (Table 3.2).

3.4.3 Query processor

The summary groups are then sent to the query processors in large batches.

Collocating groupers and query processors: To minimize the communication overhead between groupers and query processors, in our implementation processors and groupers are threads in the same process, and the summary groups are passed via memory pointers.

This is feasible because the programming model of DSHARK guarantees that each summary group can be processed independently, *i.e.,* the query functions can be executed completely in parallel. In our implementation, query processors are child threads spawned by groupers whenever groupers have a large enough batch of summary groups. This mitigates thread spawning overhead, compared with processing one group at one time. The analysis results of this batch of packet groups are in the form of a key-value dictionary and are sent to the result aggregator via a TCP socket. Finally, the query process thread terminates itself.

3.4.4 Supporting components in practice

Below, we elaborate some implementation details that are important for running DSHARK in practice.

DSHARK **compiler.** Before initiating its runtime, DSHARK compiles the user program. DSHARK generates C++ meta code from the JSON specification. Specifically, a definition of *struct Summary* will be generated based on the fields in the summary format, so that the query function has access to the value of a field by referring to *Summary.variable_name*. The template of a callback function

[3]The time for finishing TCP retransmission plus the propagation delay should still fall in three seconds.

that extracts fields will be populated using the Name section. The function will be called after the parsers identify the header stack and the pointers to the beginning of each header. The Filter section is compiled similarly. Finally, this piece of C++ code and the query function code will compile together by a standard C++ compiler and generate a dynamic link library. DSHARK pushes this library to all parsers, groupers and query processors.

Result aggregator. A result aggregator gathers the output from the query processors. It receives the key-value dictionaries sent by query processors and combines them by unionizing the keys and summing the values of the same keys. It then generates human-readable output for operators.

Coordinate parsers. DSHARK parsers consume partitioned network packet traces in parallel. In practice, this brings a synchronization problem when they process *offline* traces. If a fast parser processes packets of a few seconds ahead of a slower parser (in terms of when the packets are captured), the packets from the slower parser may fall out of grouper moving window (§3.4.2), leading to incorrect grouping.

To address this, we implemented a coordinator to simulate live capturing. The coordinator periodically tells all parsers until which timestamp they should continue processing packets. The parsers will report their progress once they reach the target timestamp and wait for the next instruction. Once all parsers report completion, the coordinator sends out the next target timestamp. This guarantees that the progress of different parsers will never differ too much. To avoid stragglers, the coordinator may drop parsers that are consistently slower.

Over-provision the number of instances. Although it may be hard to accurately estimate the minimum number of instances needed (see §3.5) due to the different CPU overhead of various packet headers and queries, we use conservative estimation and over-provision instances. It only wastes negligible CPU cycles because we implement all components to spend CPU cycles only on demand.

3.5 DSHARK Evaluation

We used DSHARK for analyzing the in-network traces collected from our production networks[4]. In

[4]All the traces we use in evaluation are from clusters running internal services. We do not analyze our cloud customers traffic without permission.

this section, we first present a few examples where we use DSHARK to check some typical network properties and invariants. Then, we evaluate the performance of DSHARK.

3.5.1 Case study

We implement 18 typical analysis tasks using DSHARK (Table 3.2). We explain three of them in detail below.

Loop detection. To show the correctness of DSHARK, we perform a controlled experiment using loop detection analysis as an example. We first collected in-network packet traces (more than 10M packets) from one of our networks and verified that there is no looping packet in the trace. Then, we developed a script to inject looping packets by repeating some of the original packets with different TTLs. The script can inject with different probabilities.

We use the same code as in §4.2. Figure 3.4 illustrates the number of looping packets that are injected and the number of packets caught by DSHARK. DSHARK has zero false negative or false positive in this controlled experiment.

Profiling load balancers. In our data center, layer-4 software load balancers (SLB) are widely deployed under ToR switches. They receive packets with a virtual IP (VIP) as the destination and forward them to different servers (called DIP) using IP-in-IP encapsulation, based on flow-level hashing. Traffic distribution analysis of SLBs is handy for network operators to check whether the traffic is indeed balanced.

To demonstrate that DSHARK can easily provide this, we randomly picked a ToR switch that has an SLB under it. We deployed a rule on that switch that mirrors *all* packets that go towards a specific VIP and come out. In one hour, our collectors captured more than 30M packets in total.[5]

Our query function generates both flow counters and packet counters of each DIP. Figure 3.5 shows the result – among the total six DIPs, DIP5 receives the least packets whereas DIP6 gets the most. Flow-level counters show a similar distribution. After discussing with operators, we conclude that for this VIP, load imbalance does exist due to imbalanced hashing, while it is still in an acceptable range.

[5]An SLB is responsible for multiple VIPs. The traffic volume can vary a lot across different VIPs.

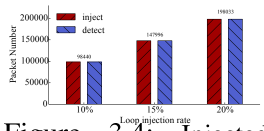


Figure 3.4: Injected loops are all detected.

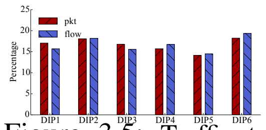


Figure 3.5: Traffic to an SLB VIP has been distributed to destination IPs.

Packet drop localizer. Noise can affect the packet drop localizer. This example shows the effectiveness of using transport-level retransmission information to reduce false positives (§3.3.5). We implemented the packet drop localizer as shown in Table 3.2, and used the noise mitigation mechanism described in §3.3.5. In a production data center, we deployed a mirroring rule on *all* switches to mirror *all* packets that originate from or go towards a small set of servers, and fed the captured packets to DSHARK. We first compare our approach, which takes into account gaps in the sequence of switches, and uses retransmissions as evidence of actual drops, with a naïve approach, that just looks at the whether the last captured hop is the expected hop. Since the naïve approach does not work for drops at the last switch, including ToR and the data center boundary (Tier-2 spine) switches, for this comparison we only considered packets whose last recorded switch were leaf (Tier-1) switches. The naïve approach reports 5599 suspected drops while DSHARK detects 7. The reason that makes the two numbers different is that the trace itself contains capturing noise. The drops detected by DSHARK are real, because they generated retransmissions with the same TCP sequence number.

To assess the effectiveness of our noise mitigation, we replayed the trace while randomly dropping capture packets with increasing probabilities. DSHARK reports 5802, 5801, 5801 and 5784 packet drops under 0%, 1%, 2% and 5% probabilities respectively. We admit there is still a possibility that we miss the retransmitted packet, but, from the result, it is very low (0.3%).

3.5.2 DSHARK component performance

Next, we evaluate the performance of DSHARK components individually. For stress tests, we feed offline traces to DSHARK as fast as possible. To represent commodity servers, we use eight VMs from our public cloud platform, each has a Xeon 16-core 2.4GHz vCPU, 56GB memory and 10Gbps virtual network. Each experiment is repeated for at least five times and we report the average. We

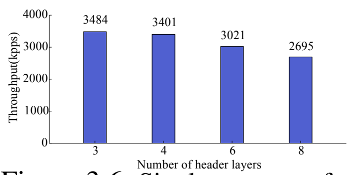


Figure 3.6: Single parser performance with different packet headers.

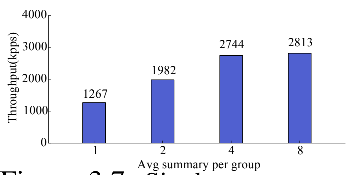


Figure 3.7: Single grouper performance with different average group sizes.

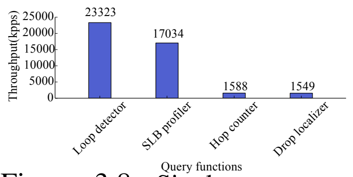


Figure 3.8: Single query processor performance with different query functions.

verify the speed difference between the fastest run and slowest run is within 5%.

Parser. The overhead of the parser varies based on the layers of headers in the packets: the more layers, the longer it takes to identify the whole header stack. The number of fields being extracted and filter constraints do not matter as much.

To get the throughput of a parser, we designed a controlled evaluation. Based on the packet formats in Table 3.1, we generated random packet traces and fed them to parsers. Each trace has 80M packets of a given number of header layers. Common TCP packets have the fewest header layers (three – Ethernet, IPv4, and TCP). The most complicated one has eight headers, *i.e.,* ⑤ in Table 3.1.

Figure 3.6 shows that in the best case (parsing a common TCP packet), the parser can reach nearly 3.5 Mpps. The throughput decreases when the packets have more header layers. However, even in the most complicated case, a single-thread parser still achieves 2.6 Mpps throughput.

Grouper. For groupers, we find that the average number of summaries in each group is the most impacting factor to grouper performance. To show this, we test different traces in which each group will have one, two, four, or eight packets, respectively. Each trace has 80M packets.

Figure 3.7 shows that the grouper throughput increases when each group has more packets. This is because the grouper uses a hash table to store the groups in the moving window (§3.4.2). The more packets in each group, the less group entry inserts and hash collisions. In the worst case (each packet is a group by itself), the throughput of one grouper thread can still reach more than 1.2 Mpps.

Query processor. The query processor performs the query function written by network operators against each summary group. Of course, the query overhead can vary significantly depending on the operators' needs. We evaluate four typical queries that represent two main types of analysis: 1) loop detection and SLB profiler only check the size of each group (§4.2); 2) the misrouting analysis and drop localization must examine every packet in a group.

Figure 3.8 shows that the query throughput of the first type can reach 17 or 23 Mpps. The second type is significantly slower – the processor runs at 1.5 Mpps per thread.

3.5.3 End-to-end performance

We evaluate the end-to-end performance of DSHARK by using a real trace with more than 640M packets collected from production networks. Unless otherwise specified, we run the loop detection example shown in §4.2.

Our first target is the throughput requirement in §3.2: 3.33 Mpps per server. Based on the component throughput, we start two parser instances and three grouper instances on one VM. Groupers spawn query processor threads on demand. Figure 3.9 shows DSHARK achieves 3.5 Mpps throughput. This is around three times a grouper performance (Figure 3.7), which means groupers run in parallel nicely. The CPU overhead is merely four CPU cores. Among them, three cores are used by groupers and query processors, while the remaining core is used by parsers. The total memory usage is around 15 GB.

On the same setup, the drop localizer query gets 3.6 Mpps with similar CPU overhead. This is because, though the query function for drop localizer is heavier, its grouping has more packets per group, leading to lighter overhead (Figure 3.7).

We further push the limit of DSHARK on a single 16-core server. We start 6 parsers and 9 groupers, and achieve 10.6 Mpps throughput with 12 out of 16 CPU cores fully occupied. This means that even if the captured traffic is comprised of 70% 64B small packets and 30% 1500B packets, DSHARK can still keep up with 40Gbps live capturing.

Finally, DSHARK must scale out across different servers. Compared to running on a single server, the additional overhead is that the shuffling phase between parsers and groupers will involve networking I/O. We find that this overhead has little impact on the performance – Figure 3.9 shows that when running two parsers and three groupers on each server, DSHARK achieves 13.2 Mpps on four servers and 26.4 Mpps on eight servers. This is close to the numbers of perfectly linear speedup 14 Mpps and 28 Mpps, respectively. On a network with full bisection bandwidth, where traffic is limited by the host access links, this is explained because we add parsers and groupers in the same

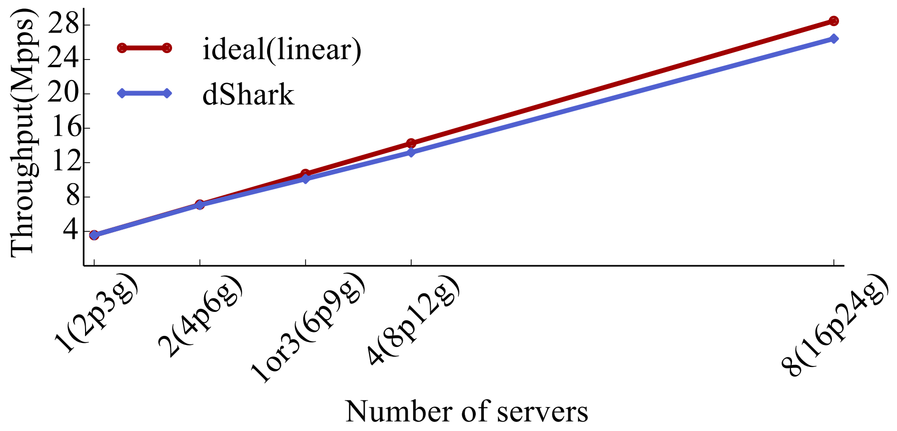


Figure 3.9: DSHARK performance scales near linearly.

proportion, and the hashing in the shuffle achieves an even distribution of traffic among them.

3.6 Discussion and Limitations

Complicated mappings in multi-hop packet traces. In multi-hop analysis, DSHARK assumes that at any switch or middlebox, there exist 1:1 mappings between input and output packets, if the packet is not dropped. This is true in most parts of our networks. However, some layer 7 middleboxes may violate this assumption. Also, IP fragmentation can also make troubles – some fragments may not carry the TCP header and break analysis that relies on TCP sequence number. Fortunately, IP fragmentation is not common in our networks because most servers use standard 1500B MTU while our switches are configured with larger MTU.

We would like to point out that it is not a unique problem of DSHARK. Most, if not all, state-of-art packet-based diagnosis tools are impacted by the same problem. Addressing this challenge is an interesting future direction.

Alternative implementation choices. We recognize that there are existing distributed frameworks [20, 23, 117] designed for big data processing and may be used for analyzing packet traces. However, we decided to implement a clean-slate design that is specifically optimized for packet trace analysis. Examples include the zero-copy data passing via pointers between parsers and trace collectors, and between groupers and query processors. Also, existing frameworks are in general heavyweight since they have unnecessary functionalities for us. That said, others may implement

DSHARK language and programming model with less lines of code using existing frameworks, if performance is not the top priority.

Offloading to programmable hardware. Programmable hardware like P4 switches and smart NICs may offload DSHARK from CPU for better performance. However, DSHARK already delivers sufficient throughput for analyzing 40Gbps online packet captures per server (§3.5) in a practical setting. Meanwhile, DSHARK, as a pure software solution, is more flexible, has lower hardware cost, and provides operators a programming interface they are familiar with. Thus, we believe that DSHARK satisfies the current demand of our operators. That said, in an environment that is fully deployed with highly programmable switches,[6] it is promising to explore hardware-based trace analysis like Marple [76].

3.7 Related Work

DSHARK, to the best of our knowledge, is the first framework that allows for the analysis of distributed packet traces in the face of noise, complex packet transformations, and large network traces. Perhaps the closest to DSHARK are PathDump [103] and SwitchPointer [104]. They diagnose problems by adding metadata to packets at each switch and analyzing them at the destination. However, this requires switch hardware modification that is not widely available in today's networks. Also, in-band data shares fate with the packets, making it hard to diagnose problems where packets do not reach the destination.

Other related work that has been devoted to detection and diagnosis of network failures includes:

Switch hardware design for telemetry [38, 54, 60, 65, 76]. While effective, these work require infrastructure changes that are challenging or even not possible due to various practical reasons. Therefore, until these capabilities are mainstream, the need to for distributed packet traces remains. Our summaries may resemble NetSight's postcards [38], but postcards are fixed, while our summaries are flexible, can handle transformations, and are tailored to the queries they serve.

Algorithms based on inference [10, 35, 36, 40, 46, 69, 73, 93? , 94]. A number of works use

[6]Unfortunately, this can take some time before happening. In some environments, it may never happen.

anomaly detection to find the source of failures within networks. Some attempt to cover the full topology using periodic probes [36]. However, such probing results in loss of information that often complicates detecting certain types of problems which could be easily detected using packet traces from the network itself. Other such approaches, *e.g.,* [69, 73, 93, 94], either rely on the packet arriving endpoints and thus cannot localize packet drops, or assume specific topology. Work such as EverFlow [**?**] is complementary to DSHARK. Specifically, DSHARK's goal is to analyze distributed packet captures fed by Everflow. Finally, [9] can only identify the general type of a problem (network, client, server) rather than the responsible device.

Work on detecting packet drops. [19, 24, 25, 41, 43, 45, 55, 61, 68, 72, 74, 79, 110, 115, 120–122] While these work are often effective at identifying the cause of packet drops, they cannot identify other types of problems that often arise in practice *e.g.,* load imbalance. Moreover, as they lack full visibility into the network (and the application) they often are unable to identify the cause of problems for specific applications [8].

Failure resilience and prevention [4, 17, 18, 32, 49, 56, 63, 64, 81, 85, 89, 95, 114] target resilience or prevention to failures via new network architectures, protocols, and network verification. DSHARK is complementary to these works. While they help avoid problematic areas in the network, DSHARK identifies where these problems occur and their speedy resolution.

Chapter 4

SIMON: Scriptable Interactive Monitoring for Networks

4.1 Background and Overview

Software-Defined Networking greatly increases the power that operators have over their networks. Unfortunately, this power comes at a cost. While logically centralized controller programs do simplify network control and configuration, the network itself remains an irrevocably distributed system. Bugs in network behavior are not eliminated in an SDN, but are merely lifted into controller programs. SDNs also introduce new classes of bugs that do not exist in traditional networks, such as flow-table consistency issues [83, 90].

Even in a relatively simple environment such as Mininet [59], it can be frustrating to understand SDN controller behavior. Simple errors can be deceptively subtle to test and debug. For instance, if an application sometimes unnecessarily floods traffic via packetOut messages, the network's performance can suffer even though connectivity is preserved. Similarly, Perešíni et al. [83] note a class of bugs that result in a glut of unnecessary rules being installed on switches, without changing the underlying forwarding semantics. Efforts that focus only on connectivity, such as testing via ping, can disguise these and other problems.

Fortunately, SDN also offers opportunities for improving how we test, debug, and verify networks.

Invariant checking tools such as VeriFlow [53] and NetPlumber [47] are a good first step. However these tools, while powerful, are limited in scope to the network's forwarding information base, and errors involving more (such as the above unnecessary-flooding error) will escape their notice. Even total knowledge of the flow-table rules installed does not suffice to fully predict network behavior; as Kuźniar, et al. [57] show, different switches have varying foibles when it comes to implementing OpenFlow, with some even occasionally disregarding barrier requests and installing rules in (reordered) batches. Thus, operators need tools that can inspect more than just forwarding tables and can determine whether the network (on all planes) respects their goals.

SIMON (Scriptable Interactive Monitoring) is a next step in that direction. SIMON is an interactive debugger for SDNs; its architecture is shown in Figure 4.1. SIMON has visibility into data-plane events (e.g., packets arriving at and being forwarded by switches), control-plane events (e.g., OpenFlow protocol messages), northbound API messages (communication between the controller and other services; e.g., see Section 4.5), and more, limited only by the monitored event sources (Section 4.4). Since SIMON is interactive, users can use these events to iteratively refine their understanding of the system at SIMON's debugging prompt, similar to using traditional debugging tools. Moreover, SIMON does not presume the user is knowledgeable about the intricacies of the controller in use.

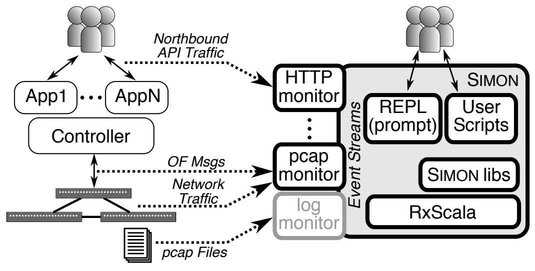


Figure 4.1: SIMON's workflow. Events are captured from the network by monitors, which feed into Simon's interactive debugging process.

The downside of an interactive debugger is that its use can often be repetitious. SIMON is thus *scriptable*, enabling the automation of repetitive tasks, monitoring of invariants, and much else. A hallmark of SIMON is its reactive scripting language, which is embedded into Scala (`scala-lang.org`). As we show in Section 4.2, reactivity enables both power and concision in debugging. Furthermore, having recourse to the full power of Scala means that SIMON enables kinds of *stateful* monitoring not supported in many other debuggers. In short, SIMON represents a fundamental shift in how we debug networks, by bringing ideas from software debugging and programming language design to bear on the problem.

4.2 Simon in Action

For illustrative purposes, we show SIMON in action on an intentionally simplistic example: a small stateful firewall application. (We discuss more complex applications in Section 4.5.) A stateful firewall should allow all traffic from internal to external ports, but deny traffic arriving at external ports unless it involves hosts that have previously communicated.

Debugging With Events

Consider a basic implementation that performs three separate tasks when an OpenFlow packetIn message is received on an internal port:

1. It installs an OpenFlow rule to forward internally-arriving traffic with this packet's source and destination;
2. It installs an OpenFlow rule forwarding replies between those addresses arriving at the external port; and
3. It replies to the original packetIn with a packetOut message so that this packet will be properly forwarded.

If the programmer forgets (3), packetOuts are never sent, and packets that arrive before FlowMod installation will be dropped. Since the OpenFlow rules are installed correctly, this bug will not be caught by flow-table invariant checkers like VeriFlow. Connections eventually work, but the bug is noticeable when pinging external hosts. Faced with this issue, an operator may ask: *What happened to the initial ping?* To investigate, we first enter the following at SIMON's prompt:

```
val ICMPStream=Simon.nwEvents().filter(isICMP);
showEvents(ICMPStream);
```

The first line reactively *filters* SIMON's stream of network events (`Simon.nwEvents()`) to remove everything but ICMP packet arrivals and departures. All streams produced are constantly updated by SIMON to maintain consistency, and so new ICMP packet arrivals will automatically be directed to `ICMPStream`. We might also achieve this effect with callbacks (Section 4.3), but at the cost of far more verbose code where we most want concision: an interactive prompt. Although the `filter` operation here is analogous to a pcap filter, we will show that SIMON provides far more flexibility.

In the second line, the `showEvents` function spawns a new window that prints events arriving on the stream it is given, allowing further study without cluttering the prompt. The window displays the following[1] when `h1` pings `h2` twice on a linear, 1-switch, 2-host topology with `h1` and `eth1` internal:

```
ICMP packet from h1 to h2 arrives at s1 on eth1
ICMP packet from h1 to h2 arrives at s1 on eth1
ICMP packet from h1 to h2 departs from s1 on eth2
```

We conclude that the initial packet is dropped by the firewall (or, at least, delayed until well after the second ping is received). The next question is: *Did the program send the appropriate OpenFlow messages to configure the firewall?*

To answer this question, we need to examine OpenFlow events that are *related* to ICMP packets. To do this, we use SIMON's powerful `cpRelatedTo` function ("control-plane related to"). It takes a packet and produces a stream of all future OpenFlow messages *related* to that packet: PacketIn and PacketOut messages containing the packet, as well as FlowMod messages whose match condition the packet satisfies. We also use the `flatMap` stream operator; here it invokes `cpRelatedTo` on each ICMP packet and merges the results into a single stream (Figure 4.2 illustrates this operation on a separate, more general example). We write:

```
showEvents(ICMPStream.flatMap(cpRelatedTo))
```

[1]Edited for clarity; SIMON displays events as JSON by default.

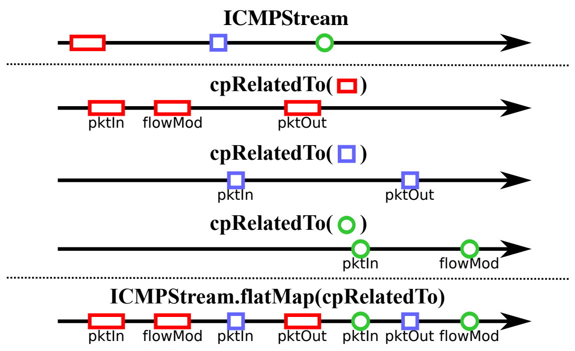


Figure 4.2: General illustration of finding related packets with `cpRelatedTo` and using `flatMap` to combine the resulting streams. Red rectangle, blue square, and green oval represent incoming ICMP packets. Each has related PacketIn, PacketOut, and/or FlowMod messages (indicated by text under the original symbol). `cpRelatedTo` places these messages into a separate stream for each original packet. `flatMap` then merges the streams into one stream of ICMP-related OpenFlow messages, keeping their relative ordering intact.

SIMON shows that, while the expected pair of FlowMods are installed for the initial packet, no corresponding packetOut is received. This explains the incorrect behavior.

Debugging via Ideal Models

The operator could have also detected the bug by describing what a stateful firewall *should* do, independent of implementation, and having SIMON monitor the network and alert them if their assumptions are violated. There are three high-level *expectations* about the forwarding behavior of a stateful firewall:

1. It allows packets arriving at internal ports;
2. It allows packets with source S and destination D arriving at external ports if traffic from D to S has been previously allowed; and
3. It drops other packets arriving at external ports.

Note that none of these expectations are couched in terms of packetOut events or flow tables; rather they describe actual forwarding behavior that users expect to see. Also, they are stateful, in that the second and third expectations refer to the history of packets previously seen.

We can implement monitors for these expectations in SIMON. To keep track of the firewall's state, we create a mutable set of pairs of addresses called `allowed`. We then use SIMON's built-in `rememberInSet` operator to keep the set up-to-date:

```
Simon.rememberInSet(ICMPStream, allowed,
  {e: NetworkEvent =>
    if(isInInt(e))
      Some(new Tuple2(e.pkt.eth.dl_src,
                      e.pkt.eth.dl_dst))
    else None});
```

We pass in a stream of events (here, ICMP packet events), the set to be mutated (`allowed`), and a function that says what, if anything, to add to the set for each event in the stream. Now, as ICMP packets arrive on the internal interface, their source-destination pairs will be automatically added to the set. The `isInInt` ("is incoming on internal") function is just a helper defined in SIMON that returns true on packets arriving on internal interfaces. `{e: NetworkEvent => ...}` is Scala syntax for defining an anonymous function over network events.

We are now free to write the three expectations using SIMON's `expectNot` function, which takes a source stream (here, the ICMP stream), along with a function that says whether or not an event violates the expectation, and a duration to wait. It then returns a stream that emits an `ExpectViolation` if a violating event is seen before the duration is up and otherwise emits an `ExpectSucceed` event. The third expectation is most interesting. First we define a helper that recognizes external packets whose destination and source are not in `allowed`[2]:

```
def isInExtNotAllow(e: NetworkEvent): Boolean = {
  e.direction == NetworkEventDirection.IN &&
  e.sw == fwswitchid && fwexternals(e.interf) &&
  !allowed((e.pkt.eth.dl_dst, e.pkt.eth.dl_src))
}
```

`fwswitchid` and `fwexternals` are configurable parameters that indicate which switch acts as a firewall and which interfaces are considered external. Now we create a stream for this expectation, saying that whenever a packet should be dropped, we expect not to see it exiting the switch:

```
val e3 = ICMPStream.filter(isInExtNotAllow).flatMap(
  e => Simon.expectNot(ICMPStream, isOutSame(e),
         Duration(100, "milliseconds")));
```

[2]SIMON's event field names (e.g., `pkt.eth.dl_dst`) follow the standard pcap format (`www.tcpdump.org`).

We elide the first and second expectations for space, but they are similar. The `isOutSame` function accepts an *incoming* packet event and produces a new function that returns true on events that are *outgoing* versions of the same packet.

After defining all three expectation streams, we merge them into a single stream that emits events whenever any expectation is violated. As before, we can call `showEvents` on this stream. Other code can also use the stream to modify state, trigger actions on the network, or even feed into new event streams. Moreover, as we discuss in Section 4.5, once expectations have been written they can be re-used to check multiple applications.

4.3 Why Reactive Programming?

We now explain the advantages of reactive programming for network monitoring in more detail. Recall this expression from earlier:

```
showEvents(ICMPStream.flatMap(cpRelatedTo))
```

Now consider how one might implement it without reactivity. A natural solution is to use synchronous calls:

```
val seen = new Set();
while(true) {
  val e = getEvent();
  if(isICMP(e))
    seen += e;
  for(orig: seen) {
    if(relatedTo(orig, e))
      println(e);
  }
}
```

Not only is this much more involved, it also quickly becomes untenable because synchronous calls will *block*. Instead, asynchronous communication is needed. Callbacks are a standard way to implement asynchrony, but even with library support for callback registration, we get something like:

```
def eventCallback(e: Event) {
  if(isICMP(e))
```

```
    Simon.nwEvents().subscribe(
      makeRelatedToCallback(e));
}
def makeRelatedToCallback(e: Event): Event => Unit {
  e2 => if(relatedTo(e, e2)) println(e2);
}
Simon.nwEvents().subscribe(eventCallback);
```

In general, this approach can produce a tangle of callbacks registering even more callbacks ad nauseum. We could avoid this with careful maintenance of state throughout a single callback. However, that only moves the complexity into the callback's internal state, which harms reusability, compositionality, and clarity of the debugging script.

Callbacks also force an inversion of a program's usual control logic: external events *push* to internal callbacks; this can be confusing, especially when integrating with existing code. Instead, reactive programs maintain the perspective that the program *pulls* events. The necessary callbacks to maintain this abstraction are handled automatically by the language, and consistency between streams is maintained without any programmer involvement. (The canonical example of this is the way a spreadsheet program automatically updates a cell if other cells it depends on change. The same is true of streams in reactive programs.) In effect, reactive programming lets the programmer structure their program *as if* they were writing with synchronous calls that return values subsequent computations can consume, leading to a compositional programming style (which has already been used successfully in SDN controllers, most notably by Voellmy, et al. [106]).

Moreover, the ability to name streams, compose them with other streams, and re-use them as values is not something callbacks can easily provide. For these reasons, reactive programs tend to be concise, which makes monitoring—with the full power of a programming language behind it—brief enough to use at the debugging prompt.

There are at least two other applications of SIMON's streams that are worth mentioning:

Interacting with the Network As we alluded to in Section 4.2, streams can be easily fed into arbitrary code. For instance, suppose we wanted to write a monitor that sends continuous pings to host `10.0.0.1`. We create a stream that detects ICMP traffic exiting the network, and feed that stream into code that sends the next ping via `subscribe`:

```
ICMPStream.filter(
  e => e.sw == fwswitchid &&
  e.direction == NetworkEventDirection.OUT)
    .subscribe( _ => "ping 10.0.0.1" ! );
```

(The `!` operation, from Scala's `process` library, executes the preceding string as a shell command.)

Caching Stream History By default, SIMON's event streams do not remember events they have already dealt with, but sometimes visibility into the past is important. Users can apply the `cache` operator to obtain a stream that caches events as they are broadcast, so that new subscribers will see them. While this incurs a space overhead, it also enables *omniscient* debugging scripts that can see into the past of the network, before they were run.

filter	Applies a function to every event in the stream, keeping only events on which the function returns true
map	Applies a function to every event in the stream, replacing that event with the function's result.
flatMap	Like map, the function given returns a stream for each event, which flatMap then merges.
cache	Cache events as they are broadcast, allowing subscribers to access event history.
timer	Emit an event after a specified delay has passed.
merge	Interleave events on multiple streams, producing a unified stream.
takeUntil	Propagate events in a stream until a given condition is met, then stop.
subscribe	Calls a function (built-in or user-defined) whenever a stream emits an event.

expect	Accepts a stream to watch, a delay, and a function that returns true or false on events. Produces a stream that will generate exactly one event: an ExpectViolation or the first event on which the function returned true.
expectNot	Similar to expect, but the function describes events that violate the expectation.
cpRelatedTo	Accepts a packet arrival event and returns the stream of future PacketIns and PacketOuts that contain the packet, as well as FlowMods whose match condition the packet would pass.
showEvents	Accepts a stream and spawns a new window that displays every event in the stream.
isICMP	Filtering function, recognizes ICMP traffic. (Similar functions exist for other traffic types.)
isOutSame	Accepts an incoming-packet event and produces a function that returns true for outgoing-packet events with the same header fields.

Figure 4.3: Selection of Reactive Operators and built-in SIMON helper functions. The first table contains commonly-used operators provided by Reactive Scala (`reactivex.io/documentation/operators.html`). The second table contains selected SIMON helper functions we constructed from reactive operators to support the examples shown in Sections 4.2 and 4.5.

4.4 A Simon Prototype

As depicted in Figure 4.1, SIMON's architecture separates the sources of network events from event processing. We implement a fully functional prototype of SIMON's event processing component, and in this paper evaluate it by monitoring a Mininet-emulated SDN running different networks and controller applications. The current sources of events in our prototype rely on the visibility into the network afforded by Mininet, but the event processing framework and scripts do not. While an immediate contribution is the ability to monitor and debug arbitrary SDN environments running in Mininet—our original motivation was the frustration of doing just this—in Section 4.7 we discuss other potential sources of events that would enable SIMON in real networks.

We implement SIMON event processing atop Scala, using Scala's ReactiveX library (`reactivex.io`) to manage streams and events. SIMON's debugging prompt is the Scala command-line interface[3] plus a suite of additional functions we wrote for processing and reacting to events. Users can either use the prompt by itself, or load external scripts from the REPL.

Figure 4.3 contains a selection of ReactiveX operators (top), as well as a selection of built-in helper functions for network debugging and monitoring (bottom). These functions are built from ReactiveX operators and are sufficient for the examples of Sections 4.2 and 4.5. If needed, additional functions can be written the same way; SIMON's helper functions are themselves scriptable.

Prototype Monitors SIMON's event processing is independent of the types of events it receives, but of course the scripts and debugging power depend on the specific input event streams. SIMON receives events from monitor components through a JSON interface.

Our current prototype uses two types of monitors: a pcap monitor that captures both data plane and OpenFlow events, and an HTTP monitor that we use to capture REST API calls to a firewall running atop the Ryu controller (Section 4.5). Both monitors use JnetPcap 1.4 (`jnetpcap.com`), a Java wrapper for libpcap, and exploit the fact that we can capture all packets from Mininet. We use the

[3]Also called a REPL, short for "read-eval-print loop". A REPL is an interactive prompt where expressions can be evaluated and programs run. Though sometimes called an "interpreter" loop, it can equally well interface to a compiler, as it does here.

APIs provided by JnetPcap and Floodlight (`www.projectfloodlight.org/floodlight/`) to deserialize data-plane data and control-plane data. The HTTP monitor currently assumes that the API calls will be contained in the first data packet of the TCP connection, which holds true for our tests.

The monitors use multiple threads to capture packets, which are timestamped by the kernel when captured. Due to the scheduling order of the threads, however, events may become accessible to SIMON out of order, so we implement a holding buffer to allow reordering of the packets. Empirically, a buffer of 50ms is enough to provide more than 96% of the packets in order. Because of the way the buffer is implemented, we change the interarrival time distribution of the packets seen by SIMON slightly, which has (minor) implications to reactive operators that depend on timeouts, such as `expect` and `expectNot`. In Section 4.7 we discuss a more general handling of time needed to apply SIMON to real networks.

4.5 Additional Case-Studies

We evaluate SIMON's utility by applying it to three real controller programs: two that implement shortest-path routing and a firewall application with real-time configurable rules. The firewall and one of the shortest-path applications are third-party implementations that are used in real networks. All are necessarily more complex than the basic stateful firewall of Section 4.2. However, unlike the previous example, none of these applications sends data-plane packets to the controller. Rather, the controller responds to northbound API events, network topology changes, etc.

Shortest-Path Routing We first examine a pair of shortest-path routing controllers. The first was the final project for a networking course designed by two of the authors. The second is RouteFlow [92], a complex application that creates a virtual Quagga (`quagga.net`) instance for each switch and emulates distributed routing protocols in an SDN.

The shortest-path ideal model in SIMON keeps up-to-date knowledge of the network's topology and runs an all-pairs-shortest-path algorithm to determine the ideal path length for each route. When the user sends a probe, the model starts a hop-counter appropriate to the probe's source and destination, and decrements the counter as the probe traverses the network. A non-zero counter at the final hop indicates a path of unexpected length, and the ideal model issues a warning. Note that the ideal model does not determine a *distinct* shortest path that packets must follow. Rather, the model is tolerant of variations in the exact paths computed by each application, so long as they are indeed shortest (by hop count). The shortest-path computation takes roughly 100 lines of code; the remaining model uses under 80 lines.

This example highlights an advantage of SIMON's approach: we were able to re-use the same ideal model for both implementations; once a model is written, its assumptions can be applied to multiple programs. Also, creating the ideal model did not require knowledge of RouteFlow or Quagga, but merely a sense of what a shortest-path routing engine should do. Of course, our model is a proof of concept, and assumes a single-area application of OSPF with known weights.

Ryu Firewall We also created an ideal model for the firewall module released with the Ryu controller platform (`osrg.github.io/ryu`). This module accepts packet-filtering rules via HTTP messages, which it then enforces with corresponding OpenFlow rules on firewall switches. These OpenFlow rules are installed proactively (i.e., the application installs them without waiting for packets to appear), but the rule-set is modified as new messages arrive.

To capture these rule-addition and -deletion messages, we took advantage of SIMON's general monitor interface to add a second event source, one that listens for HTTP messages to the controller. By creating a model aware of management messages, rather than depending on the OpenFlow messages created by the program, our SIMON model was able to check whether traffic-filtering respected the current firewall ruleset.

4.6 Related Work

We relate SIMON to other work along the four axes that characterize it: *interactivity*, *scriptability*, *reactivity*, and *visibility into network events*.

Scriptable Debugging Scriptable debuggers are not new; many have been proposed, starting with Dalek [80], which can automate repetitive tasks in gdb. Dalek's scripts are event-based, as in SIMON, but Dalek is callback-centric rather than reactive. MzTake [71] brought reactive programming to scriptable debugging. SIMON's use of the Scala command-line interface is partly inspired by MzTake's use of the DrScheme interface. Expositor [51] adds *time-travel* features to scriptable debugging, i.e., it allows users to view (and act on) events that have occurred in the past. SIMON has the capability to do the same, but we have not yet fully explored this direction. Expositor is also interactive, with a reactive programming style. All of these tools are designed for traditional program debugging, and so their notion of event visibility is different from SIMON's (e.g. method entrance and exit rather than network events).

Data-Plane Invariant Checking There has been significant work on invariant checking for the data plane. Anteater [70] provides off-line invariant checking about a network's forwarding information base. VeriFlow [53] extends the ideas of Anteater with specialized algorithms to allow *real-time* verification, ensuring invariants are respected as the rules in a live network changes. Beckett et al. [15] use annotations in controller programs to enable dynamic invariants in VeriFlow. Although powerful, these tools are limited to checking invariants about the rules installed on the network. For instance, the example of Section 4.2 would not be expressible in these tools, since the forwarding rules installed by the buggy program violate no invariants. SIMON does not require knowledge or annotation of controller program code to function, and its visibility is not limited to flow-tables.

Batfish [30] checks the overall behavior of network configurations, including routing and other factors that change over time. Like the above tools, it uses data-plane checking techniques, but the invariants it checks are not limited only to the data-plane. Batfish provides off-line configuration analysis, rather than on-line monitoring and debugging as SIMON does.

Network Monitoring and Debugging The NetSight [39] suite of tools has several goals closely aligned with SIMON. Chief among these tools is an interactive network debugger, `ndb`, which provides detailed information on the fate of packets matching user-provided filters on packet history. `ndb` is not scriptable, however, and its filters are limited to describing data-plane behavior, although control-plane context is attached to packet histories it reports. NetSight's postcard system allows it to differentiate between packets based on payload hashes, rather than using only packet header information (as our prototype monitor does); this means it provides more fine-grained packet history information than SIMON currently can. The matching invariant checker, `netwatch`, contains a library of invariants, such as traffic isolation and forwarding loop-freedom, and raises an alarm if those invariants are violated, along with the packet-history context of the violation. These invariants are limited in general to data-plane behavior; in contrast, SIMON provides visibility into all planes of the network.

Narayana, et al. [75] accept regular expressions (“path-queries”) over packet behavior and encode them as rules on switches, avoiding the collection of packets that are not interesting to the user. This is in contrast to both NetSight and SIMON, which process all packets. However, the path-queries tool is neither interactive nor scriptable, and has visibility only into data-plane behavior.

OFRewind [113] is a powerful, lightweight network monitor that records key control plane events and client data packets and allows later replay of complete subsets of network traffic when problems are detected. While it is neither scriptable nor interactive to the level SIMON provides, its replay can be an excellent source of events for analysis with SIMON.

FortNOX [84] monitors flow-rule updates to prevent and resolve rule conflicts. It provides no visibility into other types of events, is not scriptable and has no interactive interface.

Y! [112] is an innovative tool that can explain why desired network behavior did *not* occur. Such explanations take the form of a branching backtrace, where every possible cause of the desired behavior is refuted. Obtaining such explanations requires program-analysis as well as monitoring, whereas SIMON has utility even if the controller is treated as a black box. Y! does not provide interactivity or scriptability.

Other Network Debugging Tools A number of other tools provide useful debugging information without monitoring. Automated test packet generation [119] produces test-cases guaranteed to fully exercise a network’s forwarding information base. SDN traceroute [3] uses probes to discover how hypothetical packets would be forwarded through an SDN. The tool functions similarly to traditional traceroute, although it is more powerful since it allows arbitrary packet-headers to be tested. These also lack either an interactive environment or scriptability, and none leverage reactive programming.

SIMON's ideal-model description bears some resemblance to the example-based program synthesis approach of NetEgg [116]. However, NetEgg synthesizes applications from individual examples of correct behavior; ideal models fully describe the shape of correctness.

STS [97] analyzes network logs to find *minimal* event-sequences that trigger bugs. Although logs may include OpenFlow messages and other events, STS's notion of invariant violation is limited to forwarding state. Thus SIMON is capable of expressing richer invariants, although it does not attempt to minimize the events that led to undesired behavior. As STS focuses on log analysis, it provides neither scriptability nor interactive debugging.

4.7 Discussion

Reactive Programming Section 4.3 discusses how reactive programming is a natural fit to deal with the inherent streaming and concurrent nature of network events. However, not all programmers and operators will feel comfortable with it. SIMON is flexible in this regard and allows any observable to invoke event-processing callbacks at any point in a script, and progressively incorporate reactive features.

SIMON beyond Mininet SIMON as presented is agnostic to, but only as useful as, the source of events that it sees as input. In this paper we prototyped SIMON using omniscient packet capturing enabled by Mininet. Given that Mininet allows the faithful reproduction of many networking environments and SDN applications, this is already valuable.

There are, however, many other potential sources of events that can make SIMON applicable to real networks. On a live network, port-mirroring solutions such as Big Switch's Big Tap [16] can serve as sources of events, and an OpenFlow proxy like FlowVisor [99] can intercept OpenFlow messages. It is also straightforward to feed SIMON with events from logs, such as pcap traces, which are routinely captured in test and production networks. NetSight's [39] packet history files can also be used a source of events to SIMON. Finally, SIMON's interactivity and scriptability offer an excellent complement to OFRewind [113]'s replay capabilities, which offer a hybrid between online and offline monitoring.

SIMON can also compile portions of debugging scripts that involve flow-table invariants to existing checkers such as VeriFlow [53] or HSA [50].

Narayana et al. [75] make a distinction between two types of monitors: “neat freaks,” which record a narrow range of events but support correspondingly narrow functionality, and “hoarders,” which record all events available. Our prototype monitor is a hoarder; it captures all network events, which are then filtered at the prompt or in a script. While this is practical in a prototype deployment it is less so in a real network under load. A solution would be to “neaten” SIMON by analyzing how scripts process event streams and, where possible, proactively circumscribing what traffic must be captured.

Incomplete Information Some sources of events will not provide all packets in the network. Mirroring ingress and egress ports only, for example, allows for end-to-end checks in SIMON programs, but not hop-by-hop. Sampling (*e.g.* sFlow [108] and Planck [88]) makes it infeasible to witness the same packet be forwarded by different switches along a path, as sampling is uncoordinated. Incorporating these with SIMON (*e.g.* via inference) is an interesting future challenge.

Dealing with Time Most networks can synchronize clocks to acceptable accuracy, but SIMON has to be prepared to deal with occasional timing inconsistencies. Our prototype naïvely orders packet events by their timestamps after a small reordering buffer, but in real networks we will need to extend SIMON’s notion of time. Some scripts are only concerned with logical time; for these, SIMON only needs to potentially reorder events to be consistent with causal order. For these, SIMON has to maintain an internal notion of time, driven by the timestamps in the input streams, but properly corrected to be consistent with causal ordering. By observing pairs of causally related events in both directions among two sources, SIMON can compute correction factors and bounds for the different time sources in the network.

Other Application Areas SIMON applies to a wide range of situations beyond the illustrative examples seen here. For instance, SIMON could monitor a load-balancing application, sending events on a warning stream whenever balancing failed.

More broadly, networks that are not entirely controlled by a logically centralized program—e.g., networks with middleboxes—cry out for black-box methods that are nevertheless stateful. SIMON can even be used to debug problems in a non-SDN network, although it may be harder to pinpoint the cause of observed anomalies. SIMON also allows stateful debugging at the border between networks, even when one or more are not SDNs. Because it does not assume that flow-tables suffice to fully predict behavior, it can also be useful in detecting consistency errors [83, 90] or switch behavior variation [57].

Chapter 5

Conclusion

5.1 Network Verification

We introduce TITAN, a scalable and faithful BGP configuration verification system for our global-scale WAN. TITAN verifies the reachability related properties under arbitrary k failures in one run, while it is orders of magnitude more efficient than the state-of-the-art. In addition, TITAN discovers vendor-specific behaviors (VSBs) in a real-time way, which tunes the faithfulness of our network behavior model. Our network operators have used TITAN on a daily basis to check network configuration errors on our global-scale WAN.

5.2 Network Troubleshooting

We present DSHARK, a general and scalable framework for analyzing packet traces collected from distributed devices in the network. DSHARK provides a programming model for operators to specify trace analysis logic. With this programming model, DSHARK can easily address complicated artifacts in real world traces, including header transformations and packet capturing noise. Our experience in implementing 18 typical diagnosis tasks shows that DSHARK is general and easy to use. DSHARK can analyze line rate packet captures and scale out to multiple servers with near-linear speedup.

Bibliography

[1] Data plane development kit (DPDK). `http://dpdk.org/`, 2018. Accessed on 2018-01-25.

[2] Wireshark. `http://www.wireshark.org/`, 2018. Accessed on 2018-01-25.

[3] Kanak Agarwal, Eric Rozner, Colin Dixon, and John Carter. SDN traceroute: Tracing SDN forwarding without changing network behavior. In *Workshop on Hot Topics in Software Defined Networking*, 2014.

[4] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. CONGA: Distributed congestion-aware load balancing for datacenters. *ACM SIGCOMM Computer Communication Review*, 44(4):503–514, 2014.

[5] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. Solving (weighted) partial maxsat through satisfiability testing. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 427–440, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[6] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. A new algorithm for weighted partial maxsat. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pages 3–8. AAAI Press, 2010.

[7] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Liu, Jitu Padhye, Geoff Outhred, and Boon Thau Loo. Closing the network diagnostics gap with vigil. In *Proceedings*

of the SIGCOMM Posters and Demos, SIGCOMM Posters and Demos '17, pages 40–42, New York, NY, USA, 2017. ACM.

[8] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Liu, Jitu Padhye, Geoff Outhred, and Boon Thau Loo. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, 2018. USENIX Association.

[9] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 440–453. ACM, 2016.

[10] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review*, 37(4):13–24, 2007.

[11] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):2:1–2:28, 2011.

[12] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *ACM SIGCOMM (SIGCOMM '17)*, 2017.

[13] 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 *ACM SIGCOMM (SIGCOMM'16)*, 2016.

[14] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*, 2017.

[15] Ryan Beckett, X. Kelvin Zou, Shuyuan Zhang, Sharad Malik, Jennifer Rexford, and David

Walker. An assertion language for debugging SDN applications. In *Workshop on Hot Topics in Software Defined Networking*, 2014.

[16] BigSwitch Big Tap Monitoring Fabric. `http://www.bigswitch.com/products/big-tap-monitoring-fabric`.

[17] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In *ACM SIGCOMM*, pages 431–442, 2012.

[18] Guo Chen, Yuanwei Lu, Yuan Meng, Bojie Li, Kun Tan, Dan Pei, Peng Cheng, Layong Larry Luo, Yongqiang Xiong, Xiaoliang Wang, et al. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *USENIX ATC*, 2016.

[19] Yan Chen, David Bindel, Hanhee Song, and Randy H Katz. An algebraic approach to practical and scalable overlay network monitoring. *ACM SIGCOMM Computer Communication Review*, 34(4):55–66, 2004.

[20] Zaheer Chothia, John Liagouris, Desislava Dimitrova, and Timothy Roscoe. Online reconstruction of structural information from datacenter logs. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 344–358, New York, NY, USA, 2017. ACM.

[21] B. Claise, B. Trammell, and P. Aitken. RFC7011: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. `https://tools.ietf.org/html/rfc7011`, September 2013.

[22] Ed. Claise, B. RFC3954: Cisco Systems NetFlow Services Export Version 9. `https://tools.ietf.org/html/rfc3954`, October 2004.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[24] Nick Duffield. Network tomography of binary network performance characteristics. *IEEE Transactions on Information Theory*, 52(12):5373–5388, 2006.

[25] Nick G. Duffield, Vijay Arya, Rémy Bellino, Timur Friedman, Joseph Horowitz, D. Towsley, and Thierry Turletti. Network tomography from aggregate loss reports. *Performance Evaluation*, 62(1):147–163, 2005.

[26] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-wide configuration synthesis. In *29th International Conference on Computer Aided Verification(CAV'17)*, 2017.

[27] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. NetComplete: Practical network-wide configuration synthesis with autocmpleteion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, 2018.

[28] 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 *12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[29] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 43–56, Berkeley, CA, USA, 2005. USENIX Association.

[30] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Networked Systems Design and Implementation*, 2015.

[31] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*, 2015.

[32] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *NSDI, 12th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2015.

[33] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[34] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM (SIGCOMM'16)*, 2016.

[35] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. RINC: Real-time Inference-based Network diagnosis in the Cloud. Technical report, Princeton University, 2015. `https://www.cs.princeton.edu/research/techreps/TR-975-14`.

[36] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*, pages 139–152, 2015.

[37] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 357–371, New York, NY, USA, 2018. ACM.

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

[39] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Networked Systems Design and Implementation*, 2014.

[40] Brandon Heller, Colin Scott, Nick McKeown, Scott Shenker, Andreas Wundsam, Hongyi Zeng, Sam Whitlock, Vimalkumar Jeyakumar, Nikhil Handigol, James McCauley, et al. Leveraging SDN layering to systematically troubleshoot networks. In *ACM SIGCOMM HotSDN*, pages 37–42, 2013.

[41] Herodotos Herodotou, Bolin Ding, Shobana Balakrishnan, Geoff Outhred, and Percy Fitter. Scalable near real-time failure localization of data center networks. In *ACM KDD*, pages 1689–1698, 2014.

[42] Alex Horn, Ali Kheradmand, and Mukul R. Prasad. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2017.

[43] Yiyi Huang, Nick Feamster, and Renata Teixeira. Practical issues with using network tomography for fault diagnosis. *ACM SIGCOMM Computer Communication Review*, 38(5):53–58, 2008.

[44] Jesper Stenbjerg Jensen, Troels Beck Krøgh, Jonas Sand Madsen, Stefan Schmid, Jiří Srba, and Marc Tom Thorgersen. P-rex: Fast verification of mpls networks with multiple link failures. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 217–227, New York, NY, USA, 2018. ACM.

[45] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. Shrink: A tool for failure diagnosis in IP networks. In *ACM SIGCOMM MineNet*, pages 173–178, 2005.

[46] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. *SIGCOMM Comput. Commun. Rev.*, 39(4):243–254, 2009.

[47] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Networked Systems Design and Implementation*, 2013.

[48] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, 2012.

[49] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, volume 12, pages 113–126, 2012.

[50] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Networked Systems Design and Implementation*, 2012.

[51] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. Expositor: Scriptable time-travel debugging with first class traces. In *International Conference on Software Engineering*, May 2013.

[52] Ahmed Khurshid, Xuan Zhou, Whenxuan Zhou, Matthew Caesar, and Philip Brighten Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*, 2013.

[53] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *Networked Systems Design and Implementation*, April 2013.

[54] Changhoon Kim, Ed Doe Parag Bhide, Hugh Holbrook, Anoop Ghanwani, Dan Daly, Mukesh Hira, and Bruce Davie. In-band Network Telemetry (INT). `https://p4.org/assets/INT-current-spec.pdf`, June 2016.

[55] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C. Snoeren. IP fault localization via risk modeling. In *USENIX NSDI*, pages 57–70, 2005.

[56] Maciej Kuźniar, Peter Perešíni, Nedeljko Vasić, Marco Canini, and Dejan Kostić. Automatic failure recovery for software-defined networks. In *ACM SIGCOMM HotSDN*, pages 159–160, 2013.

[57] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What you need to know about SDN flow tables. In *Passive and Active Measurement*, 2015.

[58] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 183–196, New York, NY, USA, 2017. ACM.

[59] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Workshop on Hot Topics in Networks*, 2010.

[60] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *USENIX NSDI*, pages 311–324, 2016.

[61] Chang Liu, Ting He, Ananthram Swami, Don Towsley, Theodoros Salonidis, and Kin K Leung. Measurement design framework for network tomography using fisher information. *ITA AFM*, 2013.

[62] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *26th Symposium on Operating Systems Principles (SOSP'17)*, 2017.

[63] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613. ACM, 2017.

[64] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring connectivity via data plane mechanisms. In *USENIX NSDI*, pages 113–126, 2013.

[65] Yuliang Liú, Rui Miao, Changhoon Kim, and Minlan Yuú. LossRadar: Fast detection of lost packets in data center networks. In *ACM CoNEXT*, pages 481–495, 2016.

[66] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked System Design and Implementation (NSDI'15)*, 2015.

[67] Nuno P. Lopes and Andrey Rybalchenko. Fast bgp simulation of large datacenters. In *20th International Conference on Verification, Model Checking, and Abstract Interpretation(VMCAI)*, 2019.

[68] Liang Ma, Ting He, Ananthram Swami, Don Towsley, Kin K Leung, and Jessica Lowe. Node failure localization via network tomography. In *ACM SIGCOMM IMC*, pages 195–208, 2014.

[69] Ratul Mahajan, Neil Spring, David Wetherall, and Thomas Anderson. User-level internet path diagnosis. *ACM SIGOPS Operating Systems Review*, 37(5):106–119, 2003.

[70] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel T. King. Debugging the data plane with Anteater. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2011.

[71] Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engineering Journal*, 2006.

[72] Matt Mathis, John Heffner, Peter O'Neil, and Pete Siemsen. Pathdiag: Automated TCP diagnosis. In *PAM*, pages 152–161, 2008.

[73] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 129–143, New York, NY, USA, 2016. ACM.

[74] Radhika Niranjan Mysore, Ratul Mahajan, Amin Vahdat, and George Varghese. Gestalt: Fast, unified fault localization for networked systems. In *USENIX ATC*, pages 255–267, 2014.

[75] Srinivas Narayana, Jennifer Rexford, and David Walker. Compiling path queries in software-defined networks. In *Workshop on Hot Topics in Software Defined Networking*, 2014.

[76] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98. ACM, 2017.

[77] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. Compiling path queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 207–222, Santa Clara, CA, 2016. USENIX Association.

[78] Lily Hay Newman. How a tiny error shut off the internet for parts of the us. *Wired*, Nov 2017. Accessed Jan 1st, 2018.

[79] Nagao Ogino, Takeshi Kitahara, Shin'ichi Arakawa, Go Hasegawa, and Masayuki Murata. Decentralized boolean network tomography based on network partitioning. In *IEEE/IFIP NOMS*, pages 162–170, 2016.

[80] Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho. Dalek: A GNU, improved programmable debugger. In *Usenix Technical Conference*, 1990.

[81] Christoph Paasch and Olivier Bonaventure. Multipath TCP. *Communications of the ACM*, 57(4):51–57, 2014.

[82] Aurojit Panda, Ori Lahav, Katerina J. Argyraki, Mooly Sagiv, and Scott Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[83] Peter Perešíni, Maciej Kuźniar, Nedeljko Vasić, Marco Canini, and Dejan Kostić. OF.CPP: Consistent packet processing for OpenFlow. In *Workshop on Hot Topics in Software Defined Networking*, 2013.

[84] Phillip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofi Gu. A security enforcement kernel for OpenFlow networks. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[85] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *USENIX NSDI*, pages 43–57, 2015.

[86] B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with c-bgp. *IEEE Network*, 19(6):12–19, Nov 2005.

[87] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 399–412, San Jose, CA, 2012. USENIX Association.

[88] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 407–418, New York, NY, USA, 2014. ACM.

[89] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. Fattire: Declarative fault tolerance for software-defined networks. In *ACM SIGCOMM HotSDN*, pages 109–114, 2013.

[90] Mark Reitblatt, Nate Foster, Jen Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2012.

[91] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *USENIX ATC*, 2012.

[92] Christian Esteve Rothenberg, Marcelo Ribeiro Nascimento, Marcos Rogerio Salvador, Carlos Nilton Araujo Corrêa, Sidney Cunha de Lucena, and Robert Raszuk. Revisiting routing control

platforms with the eyes and muscles of software-defined networking. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[93] Arjun Roy, Jasmeet Bagga, Hongyi Zeng, and Alex Sneoren. Passive realtime datacenter fault detection. *ACM NSDI*, 2017.

[94] Arjun Roy, Jasmeet Bagga, Hongyi Zeng, and Alex Sneoren. Passive realtime datacenter fault detection. In *ACM NSDI*, 2017.

[95] Liron Schiff, Stefan Schmid, and Marco Canini. Ground control to major faults: Towards a fault tolerant and adaptive SDN control network. In *IEEE/IFIP DSN*, pages 90–96, 2016.

[96] S. Schmid and J. Srba. Polynomial-time what-if analysis for prefix-manipulating mpls networks. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 1799–1807, April 2018.

[97] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, Hrishikesh B. Acharya, Kyriakos Zarifis, and Scott Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2014.

[98] J. Scudder, R. Fernando, and S. Stuart. BGP Monitoring Protocol (BMP). RFC 7854, IETF, June 2016.

[99] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martín Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *Operating Systems Design and Implementation*, 2010.

[100] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM (SIGCOMM)*, August 2016.

[101] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Synthesis of fault-tolerant distributed router configurations. *POMACS*, 2(1):22:1–22:26, 2018.

[102] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A network-state management service. *SIGCOMM Comput. Commun. Rev.*, 44(4):563–574, August 2014.

[103] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *OSDI*, pages 233–248, 2016.

[104] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 453–456, Renton, WA, 2018. USENIX Association.

[105] J. Touch. RFC6864: Updated Specification of the IPv4 ID Field. `https://tools.ietf.org/html/rfc6864`, February 2013.

[106] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[107] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn L. Talcott. FSR: formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Network (ToN)*, 20(6):1814–1827, 2012.

[108] Mea Wang, Baochun Li Li, and Zongpeng Li. sFlow: Towards resource-efficient and agile service federation in service overlay networks. In *IEEE ICDCS*, pages 628–635, 2004.

[109] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.

[110] Chathuranga Widanapathirana, Jonathan Li, Y Ahmet Sekercioglu, Milosh Ivanovich, and Paul Fitzpatrick. Intelligent automated diagnosis of client device bottlenecks in private clouds. In *IEEE UCC*, pages 261–266, 2011.

[111] Wenji Wu and Phil DeMar. Wirecap: A novel packet capture engine for commodity nics in high-speed networks. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 395–406, New York, NY, USA, 2014. ACM.

[112] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2014.

[113] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference*, 2011.

[114] Andreas Wundsam, Amir Mehmood, Anja Feldmann, and Olaf Maennel. Network troubleshooting with mirror VNets. In *IEEE GLOBECOM*, pages 283–287, 2010.

[115] Minlan Yu, Albert G Greenberg, David A Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *USENIX NSDI*, 2011.

[116] Yifei Yuan, Rajeev Alur, and Boon Thau Loo. NetEgg: Programming network policies by examples. In *Workshop on Hot Topics in Networks*, 2014.

[117] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[118] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2012.

[119] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic Test

Packet Generation. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[120] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of internet flow rates. *ACM SIGCOMM Computer Communication Review*, 32(4):309–322, 2002.

[121] Yin Zhang, Matthew Roughan, Walter Willinger, and Lili Qiu. Spatio-temporal compressive sensing and internet traffic matrices. *ACM SIGCOMM Computer Communication Review*, 39(4):267–278, 2009.

[122] Yao Zhao, Yan Chen, and David Bindel. Towards unbiased end-to-end network diagnosis. *ACM SIGCOMM Computer Communication Review*, 36(4):219–230, 2006.

[123] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 479–491, New York, NY, USA, 2015. ACM.