Towards a Reliable and Predictable Network

Thesis Proposal

Da Yu

November 26th, 2018

Contents

Abstract of "Towards a Reliable and Predictable Network"

Today's tools to improve the reliability and manageability of networks can be generally classified into three different classes: before-the-fact network verification, during-the-fact network monitoring and after-the-fact network troubleshooting. Unfortunately, none of the above classes can individually make the network fully reliable and predictable due to fundamental limitations. For example, before-the-fact network verification prevents service downtime by proactively checking the correctness of network configuration (rather than firmware and software bugs); during-the-fact network monitoring targets on collecting network events at scale with limited analysis abstracts and capabilities, and after-the-fact network troubleshooting is mainly responsible for locating root causes like bugs after service outages occur.

To drive the network toward a fully reliable and predictable state, instead of improving tools in each of the above categories, recently, there have been proposals to make the verification, monitoring 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 run-time network state with these configurations deployed, pin-point 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.

In this work, I propose three practical tools that fit into this cycle and address important limitations of previous works. The first, in our preliminary work, addresses monitoring, called SIMON. SIMON is a Scriptable Interactive network Monitoring system. With SIMON, operators can probe the network behavior with queries interactively. They can also compose scripts to enable automation of repetitive tasks, monitoring of invariants, and much else.

Our second preliminary work DSHARK mainly targets on 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 propose work, HOYAN, focuses on verification. HOYAN is a tool to help the network operators know whether they correctly configure their network before some network issues occur. In practice, correctly configuring a production scale network organized with complex BGP policies is important but extremely hard. The state-of-the-art verifications tools have two main limitations: 1) they fail to guarantee the faithfulness of their network behavior model due to vendor specific behaviors (VSBs), and 2) they poorly scale to the global WAN.

To make HOYAN breakthrough these two limitations, we build HOYAN generating a network behavior model by simulating route propagation. To ensure the faithfulness of the network model, HOYAN needs to have a behavior model

trainer responsible of detecting the difference between the current network model and real network behaviors in a real-time way, warning the operators once a VSB is detected. To make it scale to the global WAN, during the model generation, it dynamically encodes and aggressively simplifies the constraints of route propagation to significantly improve the verification efficiency.

Chapter 1

Introduction

Thesis Statement Existing network-reliability tools can be categorized into before-the-fact verification tools, during-the-fact network monitors and after-the-fact network diagnosis tools. Each of the above categories is not able to solely make the network highly available and reliable due to *their goals and scopes*. Integrating these categories into a closed-cycle to fully leverage each category's power can drive the network toward a reliable and predictable state: this cycle starts from verifying network-wide configuration properties, then monitors the run-time network states with these configurations, and finally diagnoses the root causes after network issues occur. Once the issues get fixed, this cycle can start over. With multiple iterations of the cycle, the network evolves towards a reliable and predictable state.

However, such a cycle, even equipped with the state-of-the-art tools, still fail to be deployed into production directly. These tools perform poorly in practice because they don't take some significant practical challenges into consideration. In this thesis, I propose three tools that fit into this cycle and address the practical limitations of previous works.

1.1 Overview

High reliability has been accepted as an explicit need for today's network management. An ISP outage [68], 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 three classes: before-the-fact network verification tools [9, 24], during-the-fact network monitors [30, 77, 106] and after-the-fact network issue debuggers [89, 90]:

- **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.

- **During-the-fact network monitors** are used to help network operators better understand the network issues triggered in runtime. These monitors deploy rules on devices to collect network events and states of interest; however, since this type tools mainly focus on capturing the most recent network observations, they cannot analyze or locate the root causes of observed issues.

- **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); during-the-fact network monitoring targets on collecting network events at scale with limited analysis abstracts and capabilities; 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 [87] to make the verification, monitoring 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 run-time network state with these configurations deployed, pin-point 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.

This proposal presents three tools in a cycle that makes the network reliable and predictable by addressing these practical challenges (as shown in Figure 1.1). This cycle consists of three individual systems: HOYAN (propose work), a scalable and faithful BGP configuration verification tool, SIMON (preliminary work), a scriptable and interactive monitor

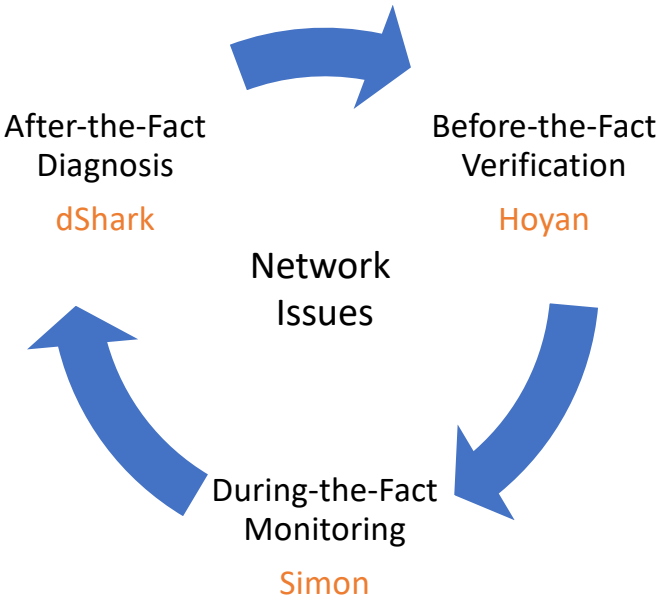


Figure 1.1: The overview of the closed-cycle to make the network reliable and predictable. The cycle starts from verifying network-wide configuration properties, then monitors the run-time network states with these configurations, and finally diagnoses the root causes after network issues occur. Once the issues get fixed, this cycle can start over.

for networks and DSHARK (preliminary work), a general, easy to program and scalable framework for diagnosing network issues. With this circle, the network operators can first verify the network configurations with HOYAN, and then monitor the network state with SIMON once the configurations are deployed. Once network issues occur, DSHARK can efficiently help the operators locate the root causes. Once the issues get fixed, this cycle can start over. With multiple iterations of the cycle, the network evolves towards a reliable and predictable state.

1.2 Preliminary Work

We started to design and build the cycle with our preliminary work in monitoring a lab-scale environment called SIMON. Then, we expand the scope to real world enterprise datacenter to troubleshoot network issues happened in the production with another preliminary work called DSHARK. We introduce these two preliminary work respectively in detail in this section.

1.2.1 During-the-Fact: SIMON: Scriptable Interactive Monitoring for Networks

In this preliminary work, I present SIMON, a scriptable, interactive monitor for networks. 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 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.

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

In this preliminary work, I 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.

1.3 Propose Work

Our preliminary work, SIMON, provides an network monitoring model to the network operators to help them better understand the latest network state. DSHARK gives the network operators the ability to fast pinpoint the root cause once some network issue happen. These two works are during-the-fact and after-the-fact. Sometimes, network operators are curious about the correctness of their network configurations. They hope to detect configuration errors before hand instead of the outages really occur. Hence, we propose to build a configuration verification tool called HOYAN, that can answer the network-operators queries (*e.g.*, traffic isolations, packets reachability) to meet their daily network management requirements.

1.3.1 Before-the-Fact: HOYAN: Scalable and Faithful BGP Configuration Verification

The correctness of configurations on network devices is vital to the connectivity, performance and reliability of today's enterprise networks [88]. To prevent incidents resulting from network configuration errors, numerous tools have been proposed recently to proactively verify the correctness of network configuration [9, 23, 25, 27]. These efforts, 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.

After trying the state-of-the-art network verification tools (*e.g.*, Minesweeper [9]), we found they can hardly be effective to verify BGP configurations in a WAN owned by a online service provider that contains tens of thousands of network devices connected via complex BGP configuration policies. Specifically, existing efforts are facing two pragmatic challenges that have not been extensively considered or studied yet:

- **Scalability.** First, the state-of-the-art tools scale poorly to the networks which have massive size but lack of topology symmetry (like WAN). On one hand, verification approaches (*e.g.*, Batfish [25]) simulate the entire process of routing propagation and convergence on control plane, and generates forwarding tables (FIBs) on data plane for a given network topology. Therefore, they have to run $\binom{n}{k}$ times to verify a configuration under arbitrary k failures out of n links. For a large-scale network with tens of thousands of links, these tools are far from efficient to fully verify the network under the $k \geq 3$ case. On the other hand, approaches, like Minesweeper [9], offer verification capable of reasoning about arbitrary k failures, but it is also not scalable, because it represents the entire control plane as a single but big logical formula; checking such a formula (even with a modern SMT solver). is very time-consuming. In addition, although recent efforts aim to improve the scalability of existing tools by taking advantage of datacenter topology's symmetry [10, 73], they are not suitable to the WAN environment.
- **Faithfulness.** Second, the correctness of the behavior model of network devices is hard to guarantee in practice, due to vendor diversity. Namely, different vendors may realize some parts of a protocol in various ways. For example, the BGP policy `remove-private-AS` in Vendor A's implementation means "removing all the private AS numbers", but means "removes only private AS numbers until the first non-private one" in Vendor B's implementation. These vendor-specific behaviors (VSB) significantly influence the faithfulness of network model, thus fundamentally invalidating the verification results. We have found none of the state-of-the-art efforts takes into account the network behavior model's faithfulness.

Our approach: HOYAN**.** We propose HOYAN—a scalable and faithful BGP configuration verification tool—which effectively addresses the preceding two challenges and can been deployed in a global-scale WAN.

In general, HOYAN can offer checks on reachability related properties under arbitrary k failures in one run, while it is orders of magnitude more efficient than the state of the art. Furthermore, HOYAN can propose a behavior model trainer for detecting the difference between current network model and actual network behaviors in a real-time way,

thus enabling network operators to easily "correct" the network behavior model accordingly. Therefore, the accuracy of HOYAN should significantly outperforms than existing tools in production networks.

Specifically, HOYAN can address the scalability challenge by leveraging the advantages of control plane simulation and logical formulas. On one hand, while logical formula-based verification approaches, like Minesweeper, are flexible to reason about the entire control plane (even under failures) in a single formula, they suffer from prolong computation time to check even just one reachability query (§4.1). This is because modeling a large-scale network would produce a huge formula. On the other hand, simulation based verification tools, like Batfish, merely obtain a single data plane model for a given topology, losing the information and behavior of control plane during failures. HOYAN aims to combine the advantages of logical formula and simulation. It simulates the process of route propagation on control plane, but the key design insight of HOYAN is to dynamically encode and aggressively simplify the constraints during the route propagation of each router. At the end, HOYAN can establish a small logical formula for each router to verify whether an IP prefix can reach the router or not under arbitrary k failures. Computing smaller logical formulas in parallel can significantly boost the speed of verification, while we can also remove unnecessary route updates during the simulation For example, when a BGP update needs $k + 1$ failures to reach a router, it can be omitted because we only consider topologies with up to k failures.

HOYAN can achieve the model faithfulness by introducing a component, called behavior model trainer, to detect the difference between current network model and real network behaviors in a real-time way. The trainer timely compares the routing tables it computes against a ground truth either from real networks or emulated networks [52]. Once the trainer finds any VSB, it traces back towards the source of this difference along the propagation path of the problematic route and finally locates the root cause within a small configuration block. Operators can, therefore, produce patches to improve the faithfulness of our network behavior model.

Chapter 2

During-the-Fact: SIMON: Scriptable Interactive Monitoring for Networks

2.1 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 [72, 79].

Even in a relatively simple environment such as Mininet [49], 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. [72] 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 [43] and NetPlumber [37] 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. [47] 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 2.1. 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 2.5), and more, limited only by the monitored event sources (Section 2.4). 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.

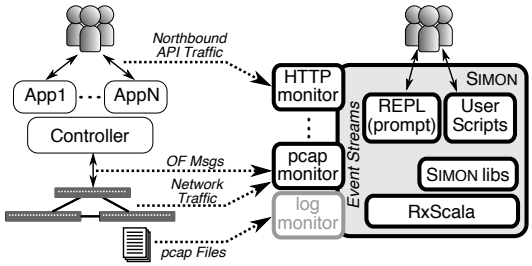


Figure 2.1: SIMON's workflow. Events are captured from the network by monitors, which feed into Simon's interactive debugging process.

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 3.3.1, 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.

2.2 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 2.5.) 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:

```
1 val ICMPStream=Simon.nwEvents().filter(isICMP);
2 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

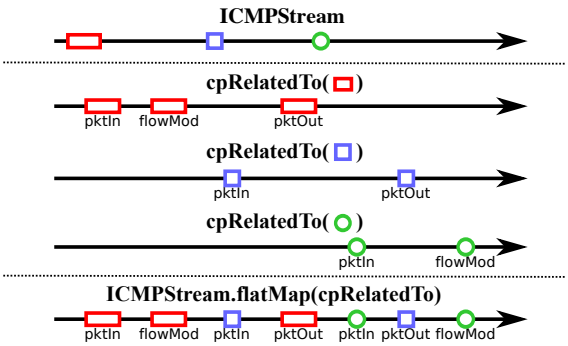


Figure 2.2: General illustration of finding related packets with `cpRelatedTo` and using `flatMap` to combine the resulting streams. Red rectangle, blue square, and green oval represent incoming ICMP packets. Each has related PacketIn, PacketOut, and/or FlowMod messages (indicated by text under the original 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.

(Section 2.3), 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:

```
1 ICMP packet from h1 to h2 arrives at s1 on eth1
2 ICMP packet from h1 to h2 arrives at s1 on eth1
3 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 2.2 illustrates this operation on a separate, more general example). We write:

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

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.

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

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 &&
```

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

```
  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")));
```

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 2.5, once expectations have been written they can be re-used to check multiple applications.

2.3 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. [92]).

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 3.3.1, 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 2.3: 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 Section 3.3.1 and section 2.5.

2.4 A Simon Prototype

As depicted in Figure 2.1, 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 2.7 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 2.3 contains a selection of ReactiveX operators (top), as well as a selection of built-in helper functions for

[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.

network debugging and monitoring (bottom). These functions are built from ReactiveX operators and are sufficient for the examples of Section 3.3.1 and section 2.5. 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 2.5). 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 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 2.7 we discuss a more general handling of time needed to apply SIMON to real networks.

2.5 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 3.3.1. 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 [81], 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.

2.6 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 [70], 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 [60] 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 [41] 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 [59] provides off-line invariant checking about a network's forwarding information base. VeriFlow [43] 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. [12] 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 3.3.1 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 [24] 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 [31] 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. [64] 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 [97] 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 [74] 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! [96] 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 [102] 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 [100]. However, NetEgg synthesizes applications from individual examples of correct behavior; ideal models fully describe the shape of correctness.

STS [85] 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.

2.7 Discussion

Reactive Programming Section 2.3 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 [13] can serve as sources of events, and an OpenFlow proxy like FlowVisor [86] 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 [31] 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 [97]'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 [43] or HSA [38].

Narayana et al. [64] 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 [93] and Planck [77]) 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 [72, 79] or switch behavior variation [47].

Chapter 3

After-the-Fact: 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 will 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 [77, 106]. 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 [18, 19], or tools that allow switches to "digest" traces earlier [30, 65, 89]. However, the former necessarily lose precision, for being aggregates, while the latter requires special hardware support which in many networks is not yet available. Alternatively, a number of tools [5, 29, 82] have tackled diagnosis of specific problems, such as packet drops. However, these too 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 [93, 106] and enable them on-demand for diagnosis. In theory, the operators, using packet traces, can reconstruct what happened in

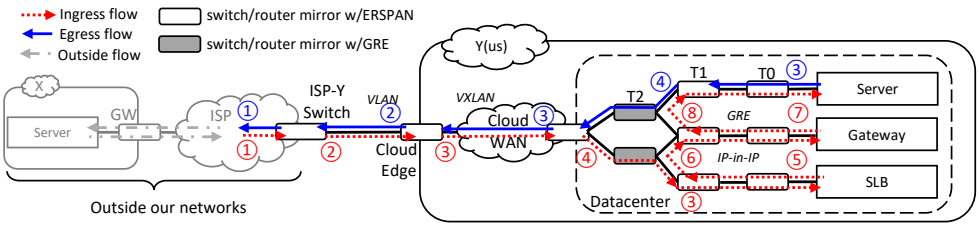


Figure 3.1: The example scenario. We collect per-hop traces in our network (Y and ISP-Y-switch) and do not have the traces outside our network except the ingress and egress of ISP-Y-switch. The packet format of each numbered network segment is listed in Table 3.1.

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.

the network. However, we found that this is not as 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.

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? Though 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 §4.1, existing tools usually require 1) full access to the network including end hosts [5, 29]; 2) specific topology, like the Clos [82], or 3) special hardware features [30, 50, 65, 89]. 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 for this.

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 a system that is similar to to [106], 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, to recover 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] have facilities 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 worked with a single thread 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 much more efficiently.

Group pattern	Application	Analysis logic	In-nw ck. only	Header transf.	Query LOC
One packet on one hop	Loop-free detection[30] *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[106] *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[30] *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[30] *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[30] *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)[106] *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[30] *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°
	Waypoint routing checker[30, 66] *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[66] *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[66] *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[66, 106] *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[106] *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[106] *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[66] *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. °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 manually 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 [76]. They become one of the main obstacles for existing tools [66, 89, 106] 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 [95]. 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 analyzing the trace. Below we list two performance goals for DSHARK.

Support real-time analysis when collocating on collectors. Recent work such as [106] and [77] 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, 80, 95], 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 [106]; 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 [93, 106].

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++:

```
map<string, int> query(const vector<Group>& groups) {
  map<string, int> r = {{"loop", 0}, {"normal", 0}};
  for (const Group& group : groups) {
```

[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 [91].

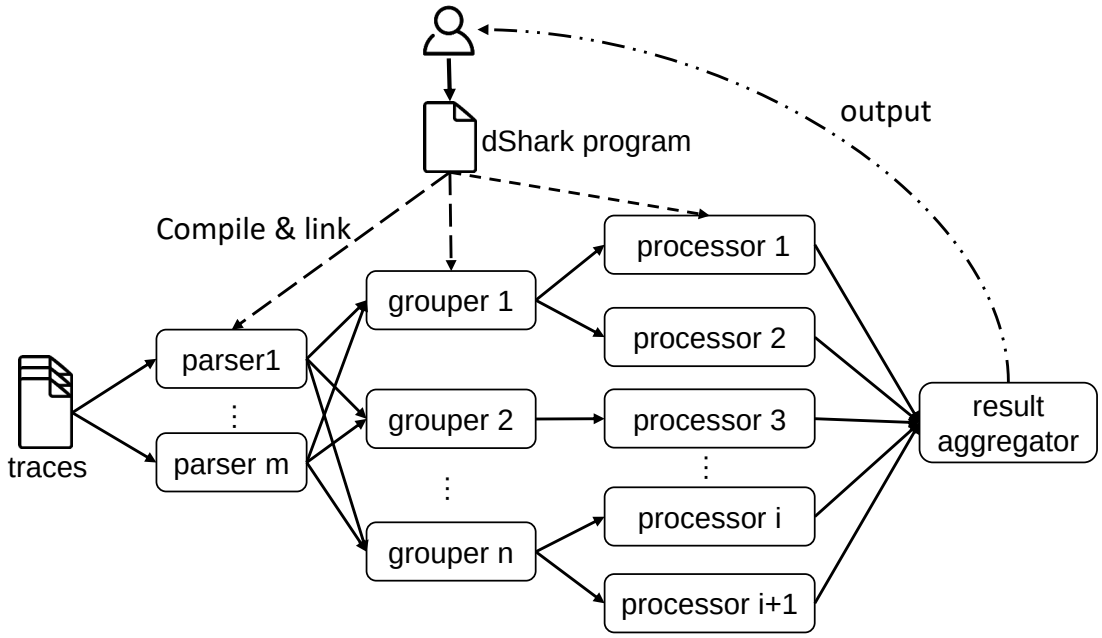


Figure 3.2: DSHARK architecture.

```
4     group.size() > 1 ?
5       (r["loop"]++) : (r["normal"]++);
6   }
7   return r;
8 }
```

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 [20]. 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 can be only focused 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,

- *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 [106]. 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 (§4.2).

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 we can see from the above example, the DSHARK programming interface consists of two parts: packet trace specifications in JSON, and query functions (the C++ code). Below we explain their design rationale and details that are not shown in the above example.

"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.

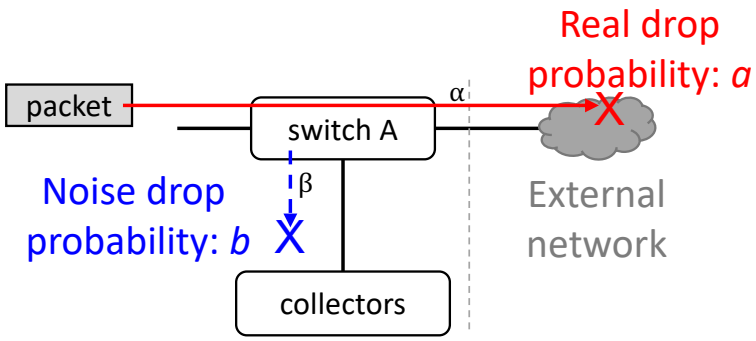


Figure 3.3: Packet capturing noise may interfere with the drop localization analysis.

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 (§3.3.1). 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 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

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.

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 that are supported by DSHARK to help 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 $<< 100\%$. 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 [6]. However, the same approach can be applied to any other reliable protocols as well. For example, QUIC [48] 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 with more than 4K lines of C++ code. It consists of three major components: parsers, groupers, and query processors. 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 identifies the header stack and, if the header stack matches any in the Filter section, checks 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

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

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 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 be compiled 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 deployed DSHARK for analyzing the in-network traces collected from our production networks[4]. In this section, we will first present a few examples where we use DSHARK to check some typical network properties and invariants. Then, we will 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 §3.3.1. 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 –

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

[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.

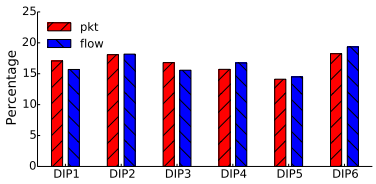


Figure 3.5: Traffic to an SLB VIP has been distributed to destination IPs.

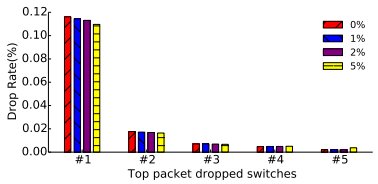


Figure 3.6: Five switches with the largest drop rates and 0% to 5% capture noise.

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.

Packet drop localizer. As explained in §3.3.5, noise can impact 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 cluster, we deployed a mirroring rule on *all* switches to mirror *all* packets that originate from or go towards a small set of servers. Then we fed the stream of real-time captures to DSHARK. Figure 3.6 shows top five switches with highest drop rate after running DSHARK for a day. Though we don't have ground truth of exactly how many packets are dropped on each switch,[6] we verified that the first switch is a ToR switch and the servers under it did report performance issues in the same time period.

Next, to verify that DSHARK is noise-resistant, we replay the same trace and randomly drop captured packets with different probabilities. In figure 3.6, we can see that packet drop localization results remain roughly the same. With 5% capture noise, the detected packet drop rate changes very little, even though the drop rate is way below 5%.

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 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. This is because the more layers, the longer it takes to identify the whole header stack. We find that the number of fields being extracted and filter constraints does 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

[6] Switch drop counters are usually not reliable, as literature shows [89, 106].

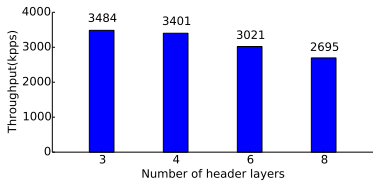


Figure 3.7: Single parser performance with different packet headers.

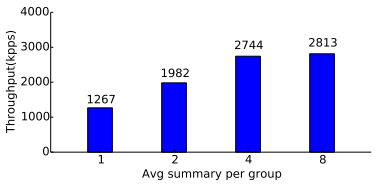


Figure 3.8: Single grouper performance with different average group sizes.

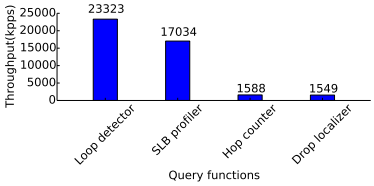


Figure 3.9: Single query processor performance with different query functions.

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 3.7 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 3.8 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 (§3.3.1); 2) the misrouting analysis and drop localization must examine every packet in a group.

Figure3.9 shows that the query throughput of the first type can easily reach more 17 or 23 Mpps. The second type is significantly slower – the query process runs at around 1.5 Mpps per thread.

3.5.3 End-to-End Performance

Finally, we evaluate the end-to-end performance of DSHARK. We use a real trace that has more than 640M packets collected from production networks. Unless otherwise specified, we run the loop detection example shown in §3.3.1.

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. Query processor threads are spawned by groupers on demand. Figure 3.10 shows DSHARK achieves 3.5 Mpps throughput. This is around three times a grouper performance (Figure 3.8), 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.

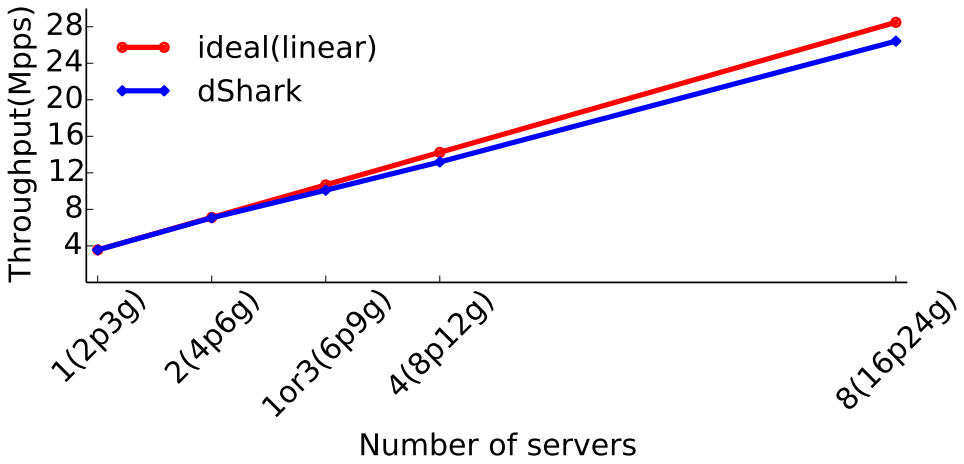


Figure 3.10: DSHARK performance scales near linearly.

We also run drop localizer using the same setup. We get 3.6 Mpps throughput with similar CPU overhead. This is because, though the query function for drop localizer is heavier, the grouping for drop localizer has more packets in each group, leading to lighter overhead (Figure 3.8). These two factors compensate each other.

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 3.10 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.

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 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 [17, 20, 101]

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,[7] it is promising to explore hardware-based trace analysis like Marple [65].

3.7 Related Work

DSHARK is designed to allow for the analysis of distributed packet traces in the face of noise, complex packet transformations, and large network traces. To the best of our knowledge, in this respect, DSHARK is the first of its kind. Perhaps the most relevant to our work is PathDump [89] and SwitchPointer [90]. They attempt to aid operators in diagnosing network problems by adding metadata to the packet 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, which makes 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 [30, 44, 50, 55, 65]. 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 [30], but postcards are fixed, while our summaries are flexible, can handle transformations, and are tailored to the queries they serve.

Algorithms based on inference [8, 28, 29, 32, 36, 58, 62, 67, 82, 83, 106]. A number of works use anomaly detection to find the source of failures within networks. Some attempt to cover the full topology using periodic probes [29]. 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 work, e.g. [58, 62, 82, 83] either rely on

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

the packet arriving endpoints and thus cannot localize packet drops, or assume specific topology. Work such as [106] is complementary to DSHARK as DSHARK provides a means of analyzing the distributed traces they provide. Finally, [7] can only identify the general type of a problem (network, client, server) rather than the responsible device.

Work on detecting packet drops. [16, 21, 22, 33–35, 45, 51, 57, 61, 63, 69, 94, 99, 103–105] 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 [6].

Failure resilience and prevention [4, 14, 15, 26, 39, 46, 53, 54, 71, 75, 78, 84, 98] target resilience to failures or failure prevention via new network architectures, protocols, and network verification. DSHARK is complementary to these works. While they help avoid problematic areas in the network, DSHARK allows for identifying where these problems occur and their speedy resolution.

Chapter 4

Propose work: Before-the-Fact: HOYAN: Scalable and Faithful BGP Configuration Verification

We propose HOYAN, 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 majon online service provider.

4.1 Background and Motivation

This section first motivates the need of BGP configuration verification on the wide area network (WAN) in a majoy online service provider (§4.1.1), and then presents the challenges in existing efforts in §4.1.2. Finally, we clarify the goal of HOYAN in §4.1.3.

4.1.1 Need of BGP Configuration Verification

The global-scale WAN is used to support the various types of online services, including cloud services, online video, search, *etc*. To meet the complex business requirements and network management goals, BGP route policies are widely used in the WAN, which coordinates multiple data centers and edge sites globally. Data centers and edge sites in the same city are connected by metropolitan area networks (MANs) via eBGP. Then, these MANs are connected by two backbone networks each of which is a single AS running iBGP internally. External ISPs are peered via eBGP at the

edge sites. With the fast growth of the business demands, the size of the whole network is almost doubled each year.

Correctly configuring the WAN is, unsurprisingly, hard and error-prone. Based on the root cause survey of the network problems occurred in recent two years (2016 and 2017) before we deploy HOYAN, for example, incidents resulting from BGP configuration errors account for the largest proportion among other types (*e.g.*, device bugs, human errors and transient errors). Therefore, reasoning about the network configurations errors (especially for BGP configuration due to the complexity) has been one of the most important tasks for the network group.

Existing work in control-plane validation falls into two broad categories: verification and emulation. Control-plane verification checks whether the logic of network configurations meets the network operators' intended properties (*e.g.*, Minesweeper [9], Batfish [25], ERA [23], *etc*). Compared with data-plane verification (*e.g.*, NoD [56], HSA [40], Anteater [11], and Veriflow [42]) control-plane verification not only offers more comprehensive verification results, but also *proactively* detects configuration errors, before adopting potentially buggy configurations in the network [9].

Network emulators like CrystalNet [52] enable the network operators to proactively validate network behaviors in a high-fidelity emulated environment. Compared with verification, network emulation has *two* issues which make the configuration verification indispensable. First, the network emulation needs vendors to provide their device firmware within a virtual machine or a container, while we found it is practically hard to get such support from all vendors at present. The second limitation is the cost: running real switch software requires large amount of computing resources (*e.g.*, $100 per hour for emulating one data center [52]). Since routers on WAN typically have much more sophisticated firmware than data center switches, relying on emulation will be extremely expensive for us. A more pragmatic way of network validation is to use verification to detect configuration issues as much as possible, and merely use emulation to detect software bugs.

4.1.2 Challenges in Existing Solutions

As our first attempt, we tried existing control-plane verification solutions to validate the BGP configurations in the WAN. Nevertheless, our founding is that the existing solutions can hardly work effectively due to two major practical challenges.

The first challenge is that existing solutions have poor scalability in the WAN. Taking Minesweeper [9] as an example, we run it on a workstation equipped with 32 CPU cores and 256 GB memory and verify several subnets with difference sizes of the WAN. The time spent to verify a single query increases with the network size exponentially and reaches about 2.5 hours when the subnet is only about 6.5% of the entire WAN. Therefore, the total verification time on the WAN can easily become intractable, and, what makes it even worse, there can be thousands of queries to fully verify a network. There are several specific reasons why Minesweeper is slow in the WAN. First, the WAN contains a huge number of IP prefixes which is expensive to handle in logical formulations [9]. Second, the WAN has numerous BGP

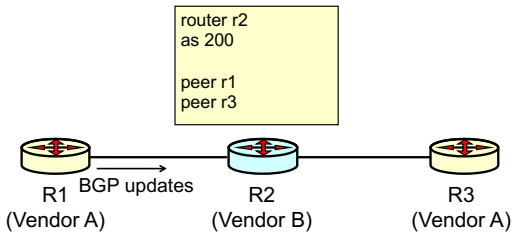


Figure 4.1: An existing vendor-specific behaviors (VSBs) from real-world configurations: Minesweeper and Batfish report R1 can talk to R3. However, in fact, BGP routes from R1 to R3 are dropped by R2. This is because Vendor B denies all BGP updates but Vendor A allows if no route policy is specified. Batfish and Minesweeper are single model (follow Vendor A) that don't handle such difference. Also, they don't have the ability to detect.

peering sessions due to the "one peer per-link" design strategy for fault tolerance. Hence a large number of peering sessions can inflate the verification time. Finally, the WAN lacks of topology symmetry, so that many acceleration strategies [10, 73] do not help.

The second challenge is that the behavior model of network devices is hard to get right due to vendor specific protocol realization, which significantly undermines the faithfulness of the verification results. In practice, we found several essential vendor specific behavior implementations which can impact the results of verifications. For example, the default behaviors of different vendors are quite different because RFC never explicitly specified those default behaviors. Figure 4.1 shows a real example in the WAN. Minesweeper and Batfish fail to get correct results, since their network models are generated incorrectly due to VSBs. In our investigation, none of existing control-plane verification solutions considers such behavior diversity. As a production WAN, using multiple vendors is an important strategy for both business and technical reasons. Therefore, it is critical to consider VSBs to ensure the faithfulness of the verification results.

4.1.3 The Goal of HOYAN

In this paper, our goal is to build a practical BGP configuration verification system which can proactively detect the most potential reachability issues due to configurations with good scalability and high faithfulness under both normal case and failure cases with up to k failures.

4.2 HOYAN Architecture

HOYAN has been used to verify BGP configuration on the global WAN. This section presents the architecture overview of HOYAN. As shown in Figure 4.2, HOYAN offers the operator two functions: 1) vendor-specific behavior (VSB) detection and 2) network property verification. The network information database, as shown in Figure 4.2, is an internal database, which stores the information, including network configuration and topology information, needed for the

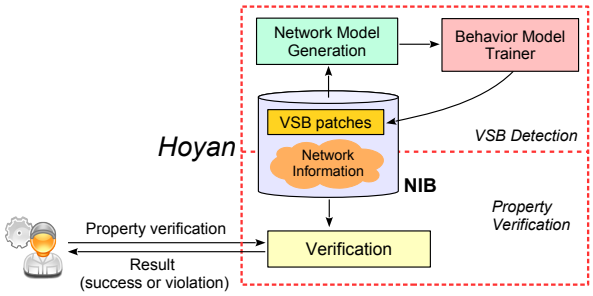


Figure 4.2: High-level workflow of HOYAN. HOYAN provides the operator two functions: network model difference detector and property verification. The former one aims to ensure that the network model correctly and completely captures network behaviors, while the latter one verifies the network property of interest.

network model generation. Intuitively, the upper part in Figure 4.2 aims to guarantee the network model correctly captures network behaviors; The property verification module, the bottom part in Figure 4.2, checks whether the current BGP configuration meets the operator's property expectation.

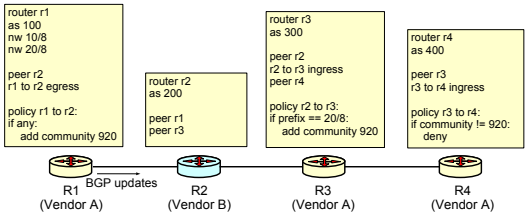
Network model. HOYAN relies on a simple network model which represents the BGP route propagation (as shown in Figure 4.4). To generate this network model, the operator needs to take the network information database—which provides 1) network configurations, 2) network topology, and 3) IP assignments, a list of IP prefixes that allocated to different services—as input. Our network model aims to offer an abstraction representing the real BGP behaviors of all the routers in the WAN.

Behavior model trainer. The behavior model trainer runs in a real-time. As shown in Figure 4.2, it reads the current network model, and compares this model with the network state and data collected from real devices. Once some diversity is detected, the trainer traces back towards the source of this difference along the propagation path of the problematic route, and finally locates the VSB within an $O(10)$ configuration block. With such a result in hand, the operators can easily fix the issue, and sends a patch to the network information database, as shown in Figure 4.2.

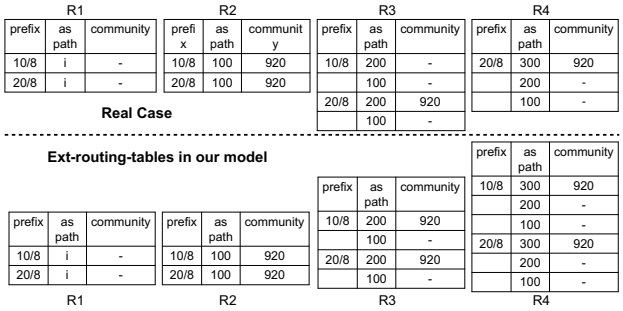
Property verification. By given a property verification query, HOYAN can check whether the current model (loaded from the network information database) meets the operator's intended properties; thus, the verification module takes the operator's intention (*e.g.*, reachability query) as input, and dynamically explores whether the specified properties are violated or not, during the process of generating network model, finally returning a verification result (*e.g.*, violation or success) to the operator. We detail our scalable verification algorithm in §4.4.

4.3 Network Model and Model Trainer

Our network model serves as the base for our verification; an incorrect network model would fundamentally undermine the verification results. In this section, we first describe how to generate a network model and then present how to detect vendor-specific behaviors (VSBs) via our behavior model trainer.



(a) Routers and their configurations.



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

Figure 4.3: A real example for vendor-specific behavior difference. By default, Vendor B's device removes the community number from its advertised BGP update; on the contrary, Vendor A's default policy does not remove the community number.

4.3.1 Network Model Generation

HOYAN can automatically generate a network model by taking the network information database (NIB) as input. The generation process mainly contains the following steps.

Initiation. HOYAN first extracts all the routers' configuration and topology information from NIB. With the information in hand, HOYAN can generate: 1) the entity nodes and their links in the physical layer, and 2) all the BGP protocol instance nodes. This is because BGP protocol information has been specified in their routers' configurations. So far, we obtain a network model with BGP nodes (but without interaction links) and their underlying physical links.

BGP node interaction. To generate the interaction among BGP nodes, HOYAN injects all the IP prefixes to the above generated network model. Our key insight is to perform the same behaviors as the propagation process of BGP updates for each IP prefix, thus generating the interactions among BGP nodes. Specifically, we take all the IP prefixes as input, and assign a queue to each IP prefix.

- For each IP prefix p, we look it as the origin BGP update and push it in its queue Q_p.
- We pop the first BGP update from Q_p. Without loss of generality, we define the popped BGP update as $U_{i\to j}$, where i and j denote the route's source and destination, respectively. We check BGP protocol instance j's ingress policy, run j's route selection algorithm, update j's routing table, select the best route, check j's egress policy, and finally advertise BGP updates to j's peers (say, m and n). Next, we push $U_{j\to m}$ and $U_{j\to n}$ in Q_p, respectively. We repeat the above operations until Q_p becomes empty.

By far, the routing tables of BGP nodes and the interactions among BGP nodes have been generated.

4.3.2 Why building a faithful model is hard?

While we can generate a network behavior model in an automatic way, the faithfulness of such a model is questionable. The key reasons include: 1) RFC specifications are ambiguous, so different vendors implement quite different behaviors in their routers for the same configuration policy; and 2) RFC specifications are not complete—*i.e.*, many corner cases are not defined, so different vendors develop their own *default* behaviors for these corner cases. We therefore call the above issues as vendor-specific behaviors (VSBs).

Because VSBs are implicit to the configuration, building a faithful behavior model in a multi-vendor network is very hard. In the rest of §4.3.2, we show more detailed examples we met in our experience.

RFC is ambiguous. RFC's specification defines protocol behaviors and processing logic in English. Words like "should", "may", *etc.* in RFC is quite vague, causing different vendors to implement the corresponding configuration policy in quite diverse ways. For example, the `remove-private-AS` policy in Vendor B's router removes all the private AS numbers, while the same policy in Vendor A's router removes only private AS numbers until the first non-private one. In this example, the configuration policy `remove-private-AS` is a VSB. Directly parsing or encoding the configuration policy (like Minesweeper and Batfish did) cannot capture these implicit behaviors. Furthermore, RFCs keep evolving—people continuously propose new RFCs or improve the old version, amplifying the ambiguity. For example, there exist about 30 RFCs specifying the BGP protocol's behaviors; the majority of vendors only implement a part of them.

RFC is 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. For example, there is no specification about what the default behavior should be if a BGP update should be propagate to a BGP node without any routing policies specified; thus, Vendor B's router denies this BGP update whereas Vendor C permits. Again, these behaviors are also not explicitly defined in the configuration policy, so parsing or encoding configuration cannot capture these behaviors.

4.3.3 Behavior Model Trainer

HOYAN proposes a behavior model trainer, capable of 1) efficiently discovering which device causes the network model to be incorrect, and 2) accurately locating VSBs within a small snippet of configuration policy. Such a trainer significantly reduces the workload of the operators on discovering VSBs.

Figure 4.3 presents a real-world example where a VSB hidden in the BGP configuration in the WAN badly affects the faithfulness of network models generated by prior work; we use it in this section to clarify the design of behavior

model trainer. Figure 4.3a shows a four-router BGP-layer link, and each router's configuration. $R1$, $R3$, and $R4$ are Vendor A routers, and $R2$ is a Vendor B's router. By default, Vendor B's device removes the community number from its advertised BGP update; on the contrary, Vendor A's default policy does not remove the community number. As described in §4.3.1, HOYAN generates a network model based on the explicit configuration policy. The four routing tables in the lower half of Figure 4.3b show the routing tables of the nodes in our network model. Because our network model is generated based on explicit configuration policy, the prefix 10/8's community number in node $R3$ and $R4$ is 920, due to the configuration policy `if any: add community 920` in $R1$.

Discovering differential pathlet. The behavior model trainer runs multiple agents to collect the *ext-routing-tables* of routers in the real network in real time; these agents compare the acquired ext-routing-tables with their corresponding ext-routing-tables in our network model (as shown in Figure 4.3b). Thus, we can build a path including all the routers with the differential ext-routing-tables. We call this path as a *differential pathlet*. For example, $R3 \rightarrow R4$ in Figure 4.3 is a differential pathlet.

Ext-routing-table is the extended routing table, which not only contains the information like prefix, as-path and next-hop, but also includes parameters for routing policy (*e.g.*, community) and route selection (*e.g.*, origin and med). We use ext-routing-table, because the information (like community) in the ext-routing-table can be leveraged to discover the differential pathlet. For example in Figure 4.3, the community number is the key information for us to find the differential pathlet. Extracting ext-routing-table is trivial in practice.

Backtracing. We backtrace each differential prefix along with the differential pathlet, P, until the first node p_1.[1] The VSB is either in p_1 or in p_1's predecessor node p_0. For example in Figure 4.3a, the first node in the differential pathlet $R3 \rightarrow R4$ is $R3$. VSB is, therefore, either in $R3$ or in $R2$, which is the predecessor of $R3$.

Locating the VSB. We check p_0's egress policy and p_1's ingress policy with the prefix, thus eventually locating the VSB. In Figure 4.3 example, we find the VSB should exist in $R2$'s egress policy. However, in $R2$'s configuration, we cannot find actions applied to the prefix 10/8. In this case, the VSB is $R2$'s default policy.

Tuning the network model. With the VSB in hand, the operators can easily write a patch to tune our network model. The patch will be submitted to NIB, thus refining the data source for our network model generation. In HOYAN, patches consists of two parts: a small piece of logic embedded in the network model generation process and a small data structure consisting of meta-data, environment variables, and a list of affected devices in the NIB. At this point, if HOYAN generates the network model again, say, for the verification purpose, the patch would be automatically loaded and the generated network model no longer has the VSB. Note that such a patch affects all the devices from the same vendor.

[1] BGP guarantees no loop exists.

Latent VSBs

A more tricky case for VSB is some routing policy may override the effects of vendor-specific behaviors. For example in Figure 4.3, as a Vendor B's router, $R2$ actually drops the community number of both 10/8 and 20/8. However, the configuration policy in $R3$—*i.e.*, `if prefix == 20/8: add community 920`—adds the community number 920 back to 20/8, which overrides the effect made by $R2$'s vendor-specific behavior. Suppose if we only have prefix 20/8 in Figure 4.3 example, our agents would not launch the model tuning process to find VSB, because there is no differential ext-routing-table. Thus, we call a VSB whose effect is override and thus "bypasses" the detection as *latent VSB*.

The most intuitive solution on detecting latent VSBs is to directly compare all routes before and after each routing policy in the network model with the real routers. However, such an "exhaustively-checking" solution is not feasible and scalable in practice. In our network, for example, the number of receiving and sending BGP updates on one router can be easily above hundreds of thousands.

We propose a scalable solution, which classifies routes into equivalence classes based on each routing policy's match condition. We can find the latent VSBs by checking the equivalence classes. Note that we only need to divide equivalence classes on each hop independently, because our ext-routing-table can successfully locate VSB in one hop. In practice, we observe routes like BGP updates are classified into at most 1295 equivalence classes on one hop.

4.3.4 Vendor and Device Feature Upgrading

In practice, the operators may 1) introduce new-version devices, 2) enable new features or policy in existing devices, or 3) use a totally new vendor to instead of some old devices. The above operations potentially create the difference between our network model and real network. In order to proactively find the potential VSBs before faults occur, we propose an approach that detects VSBs before the above updates are enabled in the real network.

We equip HOYAN with a CrystalNet-like emulator system [52], named NetMatrix. Before deploying the new devices in the real network, we run the following three operations. First, the operators load these new devices in NetMatrix, obtaining an emulated network consisting of these newly introduced devices. Then, we run the VSB discovery algorithm, designed in §4.3.3, to find potential VSBs between our network model and the emulated network. In other words, based upon the high fidelity of NetMatrix's emulation capability, we look the emulated network as real network that has deployed these new devices. Finally, the operators produce patches according to the VSB detection results, thus ensuring the network model keep its correctness. The operators can update the real network with the new devices, and our network model has been able to represent this new network.

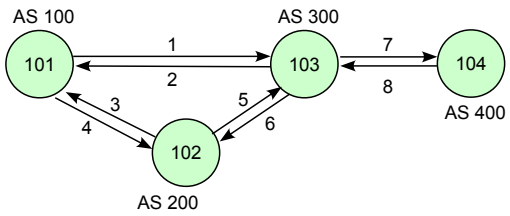


Figure 4.4: An example for BGP node interaction. Node 101-104 are BGP instance nodes. In real-world, each BGP node is a router. Suppose the operator wants to verify the BGP routing reachability from 101 to 104.

4.4 Verification Scaling to Global Networks

With a faithful network model in hand, the operators should have a high confidence to reason about their intended network properties. Because the reachability is the most common property we verified in our experience, the description before §4.4.3 mainly focuses on verifying the reachability property, and we discuss how do we verify other properties (*e.g.*, blackhole-freeness) in §4.4.3.

Goal. Our verification should be able to reason about the k-failure scenario in a scalable way. The k-failure scenario reasoning means the intended property holds under the case at most k links fail. This is important because link failures occur very common in the networks [9]. The state-of-the-art efforts like Batfish and ERA can only reason about the regular scenario, *i.e.*, the (k=0)-failure scenario [9]. While Minesweeper works for the k-failure scenario, where $k \geq 0$, it is not scalable to real networks, since it encodes the entire network control-plane behavior into a huge SMT formula.

Key insight. The insight of our algorithm design is to dynamically encode the route propagation constraints along with the network model generation. More specifically, by taking the intended property as input (*e.g.*, the A-to-B reachability), HOYAN's verification module generates the subnetwork model from A to B (via the similar way as §4.3.1), while dynamically encoding and aggressively simplifying each route's propagation constraints until the generation completes.

4.4.1 Basic Algorithm Design

We now detail our verification algorithm; we use Figure 4.4 as an example to illustrate our design.

Initialization. As shown in Figure 4.2, our verification approach also relies on NIB for the data, including network configuration, topology and IP prefixes, need by model generation. Each BGP instance node i locally maintains a table T_i, and each item in T_i is a tuple $\langle$`ROUTE`, `RECV`, `SEND`$\rangle$, where `ROUTE` represents a route received by i (*e.g.*, a BGP update $R_{k \to i}$), `RECV` and `SEND` denote this route's receiving and sending conditions, respectively. In the reachability reasoning, both receiving and sending conditions are formulas encoded by links' unique IDs. For example, as shown in Table 4.1 and Figure 4.4, two items in Node 103's local table, T_{103}, are: $\langle R_{101 \to 103}, 1, 1 \rangle$ and $\langle R_{102 \to 103}, 4 \wedge 5, \neg 1 \wedge 4 \wedge 5 \rangle$, where $4 \wedge 5$ and $\neg 1 \wedge 4 \wedge 5$ represent the receiving and sending conditions of $R_{102 \to 103}$, respectively; 1, 4 and 5 denote different links' IDs.

Table 4.1: Node 103's local table. The AS paths of $R_{101\to103}$ and $R_{102\to103}$ are 100 and 200 100, respectively.

ROUTE	Receiving Condition (RECV)	Sending Condition (SEND)
$R_{101\to103}$	1	1
$R_{102\to103}$	$4 \wedge 5$	$\neg 1 \wedge 4 \wedge 5$

Furthermore, we maintain a global queue Q responsible for tracking all the routes. Each element in Q is a tuple recording a route and its receiving condition.

The first step. Given a verification query for the reachability ($A \to B$), and k for the k-failure scenario (suppose $k = 1$ for our description), our verification starts by pushing an origin route $\langle R_{0\to A}$, True$\rangle$ in Q, where True is $R_{0\to A}$'s receiving condition, since it is the origin route.

Dynamic encoding. We pop the first element, say $\langle R_{i\to j}, r\rangle$, from Q. We thus insert a new item $\langle R_{i\to j}, r, r\rangle$ in T_j—we temporarily set SEND = RECV. Then, we reason about the best route in T_j, and re-encode the sending conditions of items according to the selection policy. For example, when $\langle R_{101\to103}, 1\rangle$ is popped from Q, we insert $\langle R_{101\to103}, 1, 1\rangle$ in T_{103}. Because $R_{101\to103}$ is the only route in T_{103} by far, the sending and receiving conditions are the same, *i.e.*, 1. After a while, $\langle R_{102\to103}, 4 \wedge 5\rangle$ is popped from Q, we insert $\langle R_{102\to103}, 4 \wedge 5, 4 \wedge 5\rangle$ in T_{103}; however, $R_{101\to103}$ is the best route, since it has the shorter AS paths (*i.e.*, 100). We therefore re-encode the sending condition of $R_{102\to103}$, getting $\neg 1 \wedge 4 \wedge 5$, which means link 4 and 5 should be alive but link 1 should not be alive, if we select this route. Because of $k = 1$, *i.e.*, at most one link can fail, $\neg 1 \wedge 4 \wedge 5$ allows. Table 4.1 shows Node 103's local table.

After updating T_j, j advertises routes to its peers (say h); thus, we generate and push $\langle R_{j\to h}, s\rangle$ in Q, where s (*i.e.*, $R_{j\to h}$'s receiving condition) is generated by using a conjunction to connect the sending condition of $R_{i\to j}$ and the link $j \to h$'s ID. For example, 102 advertises a BGP update to 103. Because the sending condition of $R_{101\to102}$ is 4 and the link $102 \to 103$'s ID is 5, we push $\langle R_{102\to103}, 4 \wedge 5\rangle$, in Q. We repeat the above dynamic encoding operations until Q is empty.

Reachability checking. To check the reachability $A \to B$, we use disjunction to connect all the items' receiving conditions in T_B. For example, in Figure 4.4, we finally get $(1 \wedge 7) \vee (\neg 1 \wedge 4 \wedge 5 \wedge 7)$, which means A and B are reachable and the reachability between A and B can tolerate at most one link fault.

4.4.2 Scaling Our Verification to the WAN

In the WAN which contains tens of thousands of routers, we notice that the receiving and sending conditions of different routes become longer increasingly with the verification process, because more and more links are encoded into both receiving and sending conditions. This leads to extremely prolonged reasoning time, making the verification poorly scale. More formally speaking, suppose we verify the reachability between A and B, and there are m BGP nodes (routers) $r_1, ..., r_m$ connected to B with different links $l_1, ..., l_m$, respectively. If each of these m nodes has n routes (or

item) in its local table T, the reachability between A and B can be represented as the following:

$$\bigvee_{i=1}^{m} (\bigvee_{j=1}^{n} (P_{i(j)} \wedge l_i)), \tag{4.1}$$

where $P_{i(j)}$ means the sending condition of the jth route of BGP node i. According to the dynamic encoding in §4.4.1, we have:

$$P_{i(j)} = \neg Rv_{i(1)} \wedge \neg Rv_{i(2)} \wedge ... \wedge \neg Rv_{i(j-1)} \wedge Rv_{i(j)}, \tag{4.2}$$

where $Rv_{i(j)}$ is the receiving condition of the jth route of node i. It is easy to learn that generating and solving such a formula are both very time-consuming.

Given the fact that generating $P_{i(j)}$ is the bottleneck in dynamic encoding, we need to answer a question: *can we simplify $P_{i(j)}$ along with the dynamic encoding process?*

Based on our experience, we find three rules are very effective to our simplification purpose. The former two fit in any case, while the third one is designed specific to the failure-case reasoning.

Rule 1: *Suppose a conjunctive form, $X = x_1 \wedge ... \wedge x_m$, connects a disjunctive form, $Y = y_1 \vee ... \vee y_n$, with a conjunction: $X \wedge Y = (x_1 \wedge ... \wedge x_m) \wedge (y_1 \vee ... \vee y_n)$. If X and Y have the same variable, then we have $X \wedge Y = X$. For example, $(B \wedge C) \wedge (A \vee D \vee B) = B \wedge C$.*

Rule 2: *If one conjunctive form C in a given disjunctive normal form (DNF) includes another conjunctive form E in the same DNF, we can simplify this DNF by removing C. For example, $B \vee (B \wedge C) = B$.*

Rule 3: *If max number of link failure is given, say k, we count the number of negation variables in each conjunctive form, and remove the conjunctive form that has more than k negative variables.*

Note that we make no claim that the above rules are novel by itself, as they are the basic rules for boolean logic; we merely utilize them, since they are specifically effective to simplify our encoding process based on our experience and design principle.

How to simplify $P_{i(j)}$?

Because most $Rv_{i(j)}$ is in DNF, $\neg Rv_{i(j)}$ can also be converted to a DNF. In order to apply our rules, we have:

$$\begin{aligned} P_{i(j)} &= \neg Rv_{i(1)} \wedge \neg Rv_{i(2)} \wedge ... \wedge \neg Rv_{i(j-1)} \wedge Rv_{i(j)} \\ &= (((Rv_{i(j)} \wedge \neg Rv_{i(1)}) \wedge \neg Rv_{i(2)}) \wedge ... \wedge \neg Rv_{i(j-1)}) \end{aligned}$$

Namely, we first simplify $Rv_{i(j)} \wedge \neg Rv_{i(1)} = DNF_1$, then simplify $DNF_1 \wedge \neg Rv_{i(2)} = DNF_2$, and finally simplify $DNF_{(j-2)} \wedge \neg Rv_{i(j-1)}$. Our experience observes that moving $Rv_{i(j)}$ to the front of $P_{i(j)}$ to simplify $\neg Rv_{i(1)} \wedge ... \wedge \neg Rv_{i(j-1)}$ iteratively enables us to "cut off" $P_{i(j)}$ most effectively. We denote $Rv_{i(j)} = (\bigwedge x_k) \vee ... \vee (\bigwedge y_l)$, since $Rv_{i(j)}$ is a DNF. We have $Rv_{i(j)} \wedge \neg Rv_{i(1)} = (\bigwedge x_k \wedge \neg Rv_{i(1)}) \vee ... \vee (\bigwedge y_l \wedge \neg Rv_{i(1)})$. Because $\neg Rv_{i(1)}$ is a DNF, we can apply Rule 1 to simplify $\bigwedge x_k \wedge \neg Rv_{i(1)}$, ..., $\bigwedge y_l \wedge \neg Rv_{i(1)}$, respectively. This is because $\bigwedge x_k \wedge \neg Rv_{i(1)}$ corresponds the form of $X \wedge Y$ in Rule 1. During the entire process, since we always generate a new DNF, we can apply Rule 2 and 3 to simplify $P_{i(j)}$.

4.4.3 Reasoning about Other Properties

HOYAN can be easily extended to verify more properties. Building on our A-to-B reachability, we support discuss how to reason about the following properties.

Blackhole. A blackhole is a situation where a router unintentionally drops packets. Thus, we can check the blackhole (from A to B) by reasoning about the complement of A-to-B reachability, *i.e.*, $blackhole_{A \to B} = \overline{reachability_{A \to B}}$.

Waypointing. The operator may want to check whether A reaches B through an intended sequence of routers (*e.g.*, to enforce advanced service chaining policies). HOYAN achieves this capability by explicitly checking whether B is reachable from A through the intended routers.

Chapter 5

Timeline

The following table summarizes the timeline for the research deliverables of the proposed work.

	Oct. 2018	Nov. 2018	Dec. 2018	Jan. 2019	Feb. 2019	Mar. 2019	Apr. 2019	May 2019
Hoyan System Build								
Complete the functionality of the network model trainer								
Complete the process logic for BGP								
Integrate static route, ACLs to Hoyan to support data plane verification								
System Evaluation								
Tune the network model in Hoyan to make it faithful								
Deploy Hoyan in production and collect experience								
Evaluate state-of-the-art solutions(Minesweeper, Batfish) in production								
Evaluate Hoyan to compare performance with the state of art solutions								
SIGCOMM' 19 submission								
Wrap up and write thesis								
Thesis Defense								

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] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 476–489. ACM, 2018.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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`.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

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

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

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

[59] 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.

[60] 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.

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

[62] 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.

[63] 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.

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

[65] 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.

[66] 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.

[67] Tim Nelson, Da Yu, Yiming Li, Rodrigo Fonseca, and Shriram Krishnamurthi. Simon: Scriptable interactive monitoring for sdns. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 19:1–19:7, New York, NY, USA, 2015. ACM.

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

[69] 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.

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

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

[72] 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.

[73] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. In *43rd ACM Symposium on Principles of Programming Languages (POPL'16)*, January 2016.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

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

[81] 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.

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

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

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439. ACM, 2016.

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

[90] 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.

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

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] 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.

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

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

[99] 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.

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

[101] 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.

[102] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic Test Packet Generation. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[103] 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.

[104] 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.

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

[106] 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.