Dynamic, Fine-Grained Data Plane Monitoring with Monocle

Peter Perešíni, Maciej Kuźniar, Dejan Kostić

Abstract—**Ensuring network reliability is important for satisfying service-level objectives. However, diagnosing network anomalies in a timely fashion is difficult due to the complex nature of network configurations. We present Monocle — a system that uncovers forwarding problems due to hardware or software failures in switches, by verifying that the data plane corresponds to the view that an SDN controller installs via the control plane. Monocle works by systematically probing the switch data plane; the probes are constructed by formulating the switch forwarding table logic as a Boolean satisfiability (SAT) problem. Our SAT formulation quickly generates probe packets targeting a particular rule considering both existing and new rules. Monocle can monitor not only static flow tables (as is currently typically the case), but also dynamic networks with frequent flow table changes. Our evaluation shows that Monocle is capable of fine-grained monitoring for the majority of rules, and it can identify a rule suddenly missing from the data plane or misbehaving in a matter of seconds. In fact, during our evaluation Monocle uncovered problems with two hardware switches that we were using in our evaluation. Finally, during network updates Monocle helps controllers cope with switches that exhibit transient inconsistencies between their control and data plane states.**

Index Terms—**Software-Defined Networking, Monitoring, Reliability**

I. Introduction

Ensuring network reliability is paramount, and this continues to be the case as the Software-Defined Networks (SDNs) are starting to be deployed [10], [26]. Most of SDN benefits (*e.g.*, flexibility, programmability) stem from its logically centralized view that is presented to network operators. This view materializes in the network by configuring network elements with forwarding rules that dictate how packets will be processed. Ensuring SDN reliability maps to ascertaining the correspondence between the high-level network policy devised by the network operators and the actual data plane configuration in switch hardware.

Multiple layers exist in the policy-to-hardware mapping [9], and SDN layering makes correspondence checking easier because of the well-defined interfaces between layers. Tools exist that can check correspondence across one or more layers ([12]–[14], [27]). Part of this difficult problem is ensuring correspondence between the desired network state that the controller wants to install, and the actual hardware (data plane). We refer to this problem as *data plane correspondence*.

Guaranteeing data plane correspondence is difficult or downright impossible by construction or pre-deployment testing, because of the possibility of various software and hardware failures ranging from transient inconsistencies (*e.g.*, switch reporting a rule was updated sooner than it happens in data plane [18]), through systematic problems (switches incorrectly implementing the specification, *e.g.*, ignoring the priority field in OpenFlow [18]), to hardware failures (*e.g.*, soft errors such as bit flips, line cards not responding, *etc.*), and switch software bugs [27].

We argue for checking data plane correspondence by actively monitoring it. However, the choice of monitoring tools is limited – operators can use end-to-end tools (*e.g.*, ping, traceroute, ATPG [27], *etc.*), or periodically collect switch forwarding statistics. We argue that these methods are insufficient – ping/traceroute and other similar tools do not determine what packet header values can test for data plane correspondence. They are also often not capable of sending arbitrary packets that are required in the SDN context. ATPG provides end-to-end data plane monitoring and can quickly localize problems, however it is designed to batch-generate probes for all network rules at the same time and as a consequence it requires substantial time (*e.g.*, minutes to hours [27], depending on coverage) to pre-compute its probes after each network change. This delay is too long for modern SDNs where the ever-increasing amount and rate of change demand a quick, dynamic monitoring tool that is the focus of this paper. In particular, a major reason behind SDN getting traction is that it makes it easy to quickly provision/reconfigure network resources (*e.g.*, virtual machines being started in a cloud data center). New network demands created by Amazon EC2 spot instances, more control being given to the applications [7], and more frequent routing recomputation (*e.g.*, every second [3]) make it even harder to ensure data plane correspondence.

Our system Monocle allows network operators to simplify their network troubleshooting by providing automatic *data plane correspondence* monitoring. Monocle transparently operates as a proxy between an SDN controller and network switches, verifying that the network view configured by the controller (for example using OpenFlow) corresponds to the actual hardware behavior. To ensure that a rule is correctly functioning, Monocle injects a monitoring packet (also referred to as a *probe*) into a switch, and examines the switch behavior. Monocle monitors multiple network switches in parallel and continuously, *i.e.*, both during *reconfiguration* (while the data plane is undergoing change during rule installation), and in *steady-state*. During reconfiguration, Monocle closely monitors the updated rule(s) and informs the controller when the rule updates sent to the switch finish being installed in hardware. This information could be used by a controller

P. Perešíni is unaffiliated, work was done at EPFL
M. Kuźniar is at Google, work was done at EPFL
D. Kostić is with KTH Royal Institute of Technology

to enforce consistent updates [23]. In steady-state, Monocle periodically checks all installed rules and reports rules misbehaving in the data plane. This localization of misbehaving rules can then be used to build a higher level troubleshooting tool. For example, link failures manifest themselves as multiple simultaneously failed rules.

Generating data plane monitoring packets is challenging for a number of reasons. First, it needs to be quick and efficient – the monitoring tool needs to quickly react to network reconfigurations, especially if the controller acts on its output. Moreover, the problem is computationally intractable (NP-hard [17]) because the monitoring packets need to match the installed rule while avoiding certain other rules present in a switch. This case routinely occurs with Access Control rules, for which the common action is to drop packets. Second, a big challenge is dealing with the multitude of rules: drop rules, multicasting, equal-cost multi-path routing (ECMP) etc. that all have to be carefully dealt with.

The **key contributions** of this work are as follows:

1) We present the design and implementation of Monocle, a data plane correspondence monitoring tool that can operate on fine-grained timescales needed in SDN. In particular, Monocle goes beyond the state-of-the-art in its ability to quickly recompute the required monitoring information during a rule update.
2) We formulate a set of formal constraints the monitoring packets must satisfy. We handle unicast, multicast, ECMP, drop rules, rule deletions and modifications. When necessary, we provide proofs that our theoretical foundation is correct. This formal treatment of the rule generation problem is the key advancement over our earlier work on RUM [16]. We also optimize the conversion of the constraints a probe needs to satisfy into a form presented to an off-the-shelf SAT solver.
3) We go beyond the state-of-the-art by providing more detail on how the probe solution (computed in abstract header space) is translated into a real packet.
4) We minimize Monocle's overhead (extra flow table space) by formulating and solving a graph vertex coloring problem.
5) Our evaluation demonstrates that Monocle: (i) detects failed rules and links in a matter of seconds while monitoring a 1000-rule flowtable in a hardware switch, (ii) ensures truly consistent network updates by providing accurate feedback on rule installation with only several ms of delay, (iii) takes between 1.48 and 4.03 ms on average to generate a probe packet on two datasets, (iv) typically has small overhead in terms of additional packets being sent and received, and (v) works with larger networks as shown by delaying an installation of 2000 flows by only 350ms.
6) We describe our experience of using Monocle showing that it can reveal switch problems that were previously unknown to us. This is one of the major advancements of this article over our earlier conference paper [22].

II. Monocle design

Monocle is positioned as a layer (proxy) between the OpenFlow controller and the network routers/switches. Such design allows it to intercept all rule modifications issued to switches and maintain the (expected) contents of flow tables in each switch. After determining the expected state of a switch, Monocle can compute packet headers that exercise the rules on that switch. Figure 1 shows the core mechanism that the system uses to monitor a rule. Monocle uses data plane probing as the ultimate test for a rule's presence in the switch forwarding table. Probing involves instructing an "upstream" switch to inject a packet toward the switch that is being probed. The "downstream" switch has a special catching rule installed which forwards the probe packet back to Monocle. Upon the receipt of the correctly modified probe packet coming from the appropriate switch, Monocle can confirm that the tested rule behaves correctly in the data plane and can move to monitoring other rules. To ensure that probing does not affect the controller-generated network state, Monocle filters out all probes before they reach the controller.

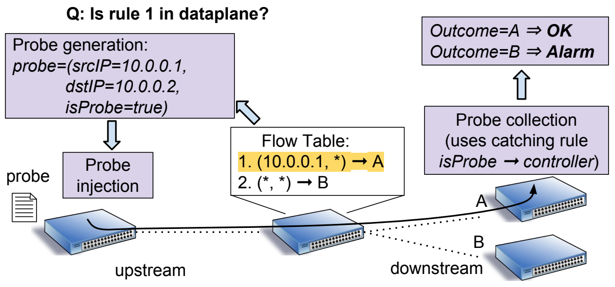

Q: Is rule 1 in dataplane?
Probe generation:
probe=(srcIP=10.0.0.1, dstIP=10.0.0.2, isProbe=true)
Probe injection
probe
Flow Table:
1. (10.0.0.1, *) → A
2. (*, *) → B
Outcome=A ⇒ OK
Outcome=B ⇒ Alarm
Probe collection (uses catching rule isProbe → controller)
A
B
upstream
downstream

Fig. 1: Overview of data-plane rule checking

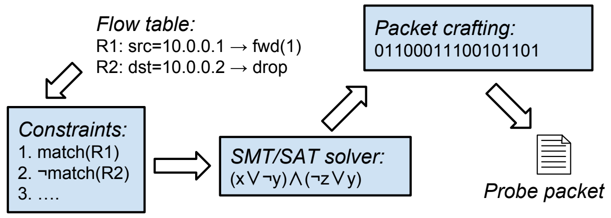


Fig. 2: Steps involved in probe generation. Probes for different rules can be generated in parallel.

Before Monocle starts monitoring the network, it computes and installs the catching rules. To reliably separate production and probing traffic, the catching rule needs to match on a particular value of a header field that is otherwise unused by rules in the network; moreover, this value cannot be used by the production traffic. In a network that requires monitoring rules at multiple switches several such catching rules are needed. It is therefore important to minimize the number of extra catching rules that have to be installed. We formulate this problem as a graph vertex coloring problem and solve it.

As Monocle relies on catching rules for its proper functioning, we need to verify that these rules work correctly. Fortunately, it is easy to check if the catching rules are installed by injecting packets that match them directly. Additionally a broken catching rule appears as a correlated failure of all rules checked using this rule. Similarly, Monocle relies on correct PacketOut message handling. We check that this assumption holds before deployment, but all PacketOut failures during network operation are indistinguishable from rule failures.

Figure 2 outlines how the probe packets are created. Mon-

Hit	$Matches(probe, R_{probed}) \quad \wedge \quad \forall R \in Rules: R.priority > R_{probed}.priority \Rightarrow \neg Matches(probe, R)$
Distinguish	Let $LowPrioRules := \{R \in Rules: \ R.priority < R_{probed}.priority\}$ and $IsHighestMatch(pkt, R, Rules) := Matches(pkt, R) \quad \wedge \quad (\forall R' \in Rules: R'.priority > R.priority \Rightarrow \neg Matches(pkt, R'))$ Then $\forall R \in LowPrioRules: IsHighestMatch(probe, R, LowPrioRules) \Rightarrow DiffOutcome(probe, R_{probed}, R)$
Collect	$Matches(probe, R_{catch})$

TABLE I: Summary of constraints that probe packets needs to satisfy when probing for rule R_{probed}.

ocle leverages its knowledge of the flow table at the switch to create a set of constraints that a probe packet should satisfy. Next, our system converts the constraints to a form understood by an off-the-shelf satisfiability (SMT/SAT) solver. Keeping constraint complexity low is important for the solving step. For this reason, Monocle formulates constraints over an abstract packet view [14], [27], structured as a collection of header fields. As the final step, Monocle converts the SAT solution, represented in an abstract view, into a real probe packet. It uses existing packet generation libraries to perform this task.

While we use OpenFlow 1.0 as a reference when describing and evaluating the system, its usefulness is not limited to this protocol. Presented techniques are more general and apply to other types of matches and actions (*e.g.*, multiple tables, action groups, ECMP).

III. Steady-state Monitoring

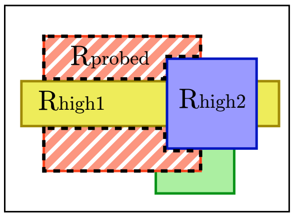
During steady-state monitoring, Monocle tests whether the control plane view of the switch forwarding state (constructed by observing proxied controller commands) corresponds to the data plane forwarding behavior. To ascertain the correspondence, Monocle actively cycles through all installed rules and for each rule it (i) generates a data plane packet confirming the presence of the rule in data plane, (ii) injects this packet into the network, and (iii) moves on to testing the next rule as soon as the packet travels through the switch and it is successfully received by Monocle. In this section, we explain the creation of monitoring packets by gradually looking at increasingly complex forwarding rules.

A. Basic unicast rules

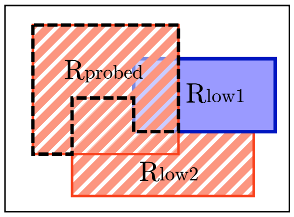
The presence of a given rule on a switch can be reliably determined if and only if there exists a packet that gets processed by a switch differently depending on whether the monitored rule is installed and working correctly. Therefore, the probe packet for monitoring the rule has to: (i) *hit* the given rule, (ii) *distinguish* the absence of the rule, and (iii) be *collected* by Monocle at the downstream switch. We formulate these conditions as formal constraints summarized in Table I.

Hitting a rule: Only packets that *match* a given rule can be affected by this rule. Therefore, the header of any potential probe packet P must be matching the R_{probed} rule. Additionally, R_{probed} is seldom the only rule on the switch and different rules can overlap (*i.e.*, a packet can match multiple rules; switch resolves such a situation by taking rule priorities into account[1]). As such, for a probe P to be *processed* according to R_{probed}, P cannot match any rule with a priority higher than the priority of R_{probed} (Figure 3a).

[1] According to the OpenFlow specification, the behavior for equal-priority overlapping rules is undefined. Thus, we do not consider such a situation.



(a) Hit: A probe for the striped rule needs to avoid any higher-priority rule(s).



(b) Disitnguish: A probe needs to distinguish the rule from lower-priority rule(s) with the same outcome.

Fig. 3: Illustration of probe generation intricacies. A valid probe must belong to a region(s) within the dashed outline(s)

Distinguishing the absence of a monitored rule: Even the rules with priority lower than the probed rule R_{probed} affect the probe generation (Figure 3b). For example, if the probe matches a low priority rule R_{low2} that forwards packets to the same port as R_{probed}, there is no way to determine if R_{probed} is installed or not. Thus, the probe has to avoid such rules. There is an intricate difference between a packet matching a rule R and being processed by R. Notably, if we just prevent P from matching all lower-priority rules with the same outcome, we may fail to generate a probe despite the fact that a valid probe exists. Consider a following set of rules ordered from lowest to highest priority (and unrelated to Figure 3b):

- $R_{lowest} := match(srcIP{=}*, dstIP{=}*) \rightarrow fwd(1)$, *i.e.*, default forwarding rule
- $R_{lower} := match(srcIP{=}10.0.0.1, dstIP{=}*) \rightarrow fwd(2)$, *i.e.*, traffic engineering diverts some flows
- $R_{probed} := match(srcIP{=}10.0.0.1, dstIP{=}10.0.0.2) \rightarrow fwd(1)$, *i.e.*, override specific flow, *e.g.*, for low latency

If the constraint prevented P from matching R_{lowest} (the same output port as R_{probed}), we would be unable to find any probe that matches R_{probed}. However, there exists a valid probe $P := (srcIP{=}10.0.0.1, dstIP{=}10.0.0.2)$ as the behavior of the switch with and without R_{probed} is different (R_{lower} overrides R_{lowest} for such a probe).

The provided example demonstrates that special care should be taken to properly formulate the *Distinguish* constraint listed in Table I: When R_{probed} is potentially missing from the data plane, probe P cannot distinguish R_{probed} from an arbitrary lower priority rule R_{LP} having the same outcome as R_{probed} if probe P matches both R_{probed} and R_{LP} and at the same time P does not match any rule with a priority higher than R_{LP} (except R_{probed} itself). To formalize this statement, we define predicate $IsHighestMatch(P, R, OtherRules)$ that indicates whether packet P is processed according to rule R even if it matches some other rules on the switch. Using $IsHighestMatch$ we can assert that the probed rule R_{probed} must be distinguishable (*e.g.*, have a *different* *outcome*) from the rule which would process probe P if R_{probed} was not installed. For simplicity one may think about $DiffOutcome(P, Rule_1, Rule_2)$ as a test $Rule_1.outport \neq Rule_2.outport$, but we later expand this definition to accommodate rewrite and multicast.

Collecting probes: Monocle decides if a rule is present in the data plane based on what happens to the probe packet (referred to as probe *outcome*). To gather this information but not affect the production traffic, we need to reserve a set of values of some header field exclusively for probes and ensure that production traffic will not use the reserved values. We then pre-install a special "probe-catch" rule on each neighboring switch; this catching rule redirects probe packets to the controller and needs to have the highest priority. Naturally, as a last constraint, the probe P has to match the probe-catch rule R_{catch} of the expected next-hop switch.

B. Unicast rules with rewrites

Rules in the network may rewrite portions of the header before forwarding the packet. Accounting for header rewrites affects the feasibility of probe generation for certain rules. Consider a simple example containing two rules:

- $R_{low} := match(srcIP{=}*) \rightarrow fwd(1)$ and
- $R_{high} := match(srcIP{=}10.0.0.1) \rightarrow fwd(1)$.

It is impossible to create a probe for the high-priority rule R_{high} because it forwards packets to the same port as the underlying low-priority rule. However, if instead of R_{high} there was a different rule $R'_{high} := match(srcIP{=}10.0.0.1) \rightarrow rewrite(ToS \leftarrow voice), fwd(1)$ that marks certain traffic with a special type of service, we could distinguish it from R_{low} based on the rewriting action. The outcome of the switch processing a probe $P := (srcIP{=}10.0.0.1, ToS \neq voice)$ unambiguously determines if R'_{high} is installed.

In general we can distinguish probes either based on ports they appear on, or by observing modifications done by the rewrites. Therefore, we define $DiffOutcome(P, R_1, R_2) := DiffPorts(R_1, R_2) \vee DiffRewrite(P, R_1, R_2)$. However, checking if two rewrites are different requires more care than checking for different output ports. A strawman solution that checks if rewrite actions defined in two rules modify the same header fields to the same values does not work. Consider again rules R_{low} and R'_{high}. While the rewrites are structurally different (*e.g.*, $rewrite(None) \neq rewrite(ToS \leftarrow voice)$), they produce the same outcome if the probe packet happens to have $ToS = voice$. Therefore, to compare the outcome of rewrite actions, we need to take into account not only the rewrites themselves but also the header of the probe packet P and how it is transformed by the rules in question. Formally, we say that the rewrites of two rules are different for a given packet if and only if they rewrite differently at least one bit of the packet, *i.e.*, $DiffRewrite(P, R_1, R_2) := \exists i \in 1 \ldots headerlen :$

$$\big(BitRewrite(P[i], R_1) \neq BitRewrite(P[i], R_2)\big)$$

where $BitRewrite(P[i], R)$ is either 0, 1, or $P[i]$ depending if rule R rewrites the bit to a fixed value or leaves it unchanged.

Finally, the rules in the network must not rewrite the header field reserved for probing. This assumption is required for two reasons: (i) if the probed rule rewrites the probe tag value, the downstream switch will be unable to distinguish and catch the probes; and additionally (ii) the headers of ordinary (non-probing) packets could be rewritten as well and afterward treated as probes; this would break the data plane forwarding.

C. Drop rules

Drop rules can be easily distinguished from unicast rules based on output ports— the downstream switch either receives the probe or not. However, verifying that probes are dropped (a situation we call *negative probing*) brings in a risk of false positives: If the rule is not installed but monitoring packets get lost or delayed for other reasons (e.g. overloaded link, packets damaged during transmission, etc.), Monocle is unable to determine the difference and assumes the rule itself drops the packets and thus is correctly installed in the data plane.

While false positives should be tolerable in most cases (*e.g.*, the production traffic is likely to share the same destiny as the probes and therefore the end-to-end invariant – traffic should be dropped – is maintained), we present a fully reliable method useful mainly for network updates monitoring in Section IV-C.

Finally, since drop rules do not output any packets, header rewrites performed by them are meaningless and do not distinguish rules. As such we define $DiffRewrite(P, R_{drop}, R') := False$ to fit our theory.

D. Multicast / ECMP rules

After discussing the rules that modify header fields and send packets to a single port or drop them, the only remaining rules are those that forward packets to several ports (*e.g.*, multicast/broadcast and ECMP). Such rules can be easily incorporated into our formal framework just by modifying the definition of $DiffOutcome$.

Both ECMP and multicast rules define a *forwarding set* of ports and send a packet to all ports in this set (multicast/broadcast) or a different port from this set at different times (ECMP). Moreover, note that drop and unicast rules are just special cases of multicast with zero and one element in their forwarding sets, respectively. Therefore, we only need to define $DiffOutcome$ for the following three combinations of rule types: (i) multicast + multicast, (ii) ECMP + ECMP, and (iii) multicast + ECMP. In all of these cases, we can distinguish rules again based on either their ports (forwarding sets) or based on their header rewrites, *e.g.*, $DiffOutcome(P, R_1, R_2) := DiffPorts(R_1, R_2) \vee DiffRewrite(P, R_1, R_2)$. We start by describing the case of distinguishing by different ports.

If both rules are multicast, a packet will appear on all ports from one of the forwarding sets. Therefore, if any port distinguishes these forwarding sets, we can use it to confirm a rule. As such, $DiffPorts(R_1, R_2) := (F_1 \neq F_2)$ where F_1 and F_2 denote forwarding sets of R_1 and R_2 respectively.

If both rules are ECMP, since each rule can send a packet to any port in its forwarding set, we can reliably distinguish them only if the forwarding sets do not intersect[2] (a probe

[2] In an alternative, probabilistic approach Monocle could generate enough probes that statistically at least one differentiates distinct forwarding sets. We decided not to use this method because it heavily relies on the switch hashing function and may fail if the switch hashes on a small subset of header fields.

appearing at a port in the intersection will not distinguish the rules as both rules can send a packet there). Thus, in this case $DiffPorts(R_1, R_2) := \big((F_1 \cap F_2) = \emptyset\big)$.

If only one of the rules (assume R_1) is multicast, we are sure that a packet will either appear on all ports in F_1, or on only one (unknown) port in F_2. We can simply capture the probe on any port that does not belong to F_2. Therefore, $DiffPorts(R_1, R_2) := \big((F_1 \setminus F_2) \neq \emptyset\big)$.

Finally, there is an additional way to distinguish an ECMP rule from a multicast rule that is not unicast (*i.e.*,$|F_1| \neq 1$). We can differentiate them by counting received probes (an ECMP rule always sends a single probe). This way of counting the expected number of probes on the output is applicable in general and can extend the definitions of $DiffOutcome$, but since it is practically useful only in the presented scenario, we treat it as an exception rather than a regular constraint.

Now we analyze a situation when a rule may apply (possibly different) rewrite actions to packets sent to different ports. We again need to consider the three types of combinations of rules R_1, R_2 with forwarding sets F_1, F_2 and adjust the definition of $DiffRewrite$ for each of them. When considering $DiffRewrite$, we take into account only actions that precede sending a packet to a port that belongs to $F_1 \cap F_2$ since if a packet appears at any other port, the location is sufficient to distinguish the rules. Additionally, we will need a new predicate: $DiffRewriteOnPort(P, R_1, R_2, port)$ which is true if the rule R_1 rewrites packet P differently than rule R_2 on port $port$. With the aforementioned observations we consider possible cases.

If both rules are multicast, there is going to be a probe packet at each output port in one of the forwarding sets. Thus, it is sufficient if there is a single port in the $F_1 \cap F_2$ on which the outputted packet is different depending which rule processed it. Therefore, $DiffRewrite(P, R_1, R_2) := \exists port \in F_1 \cap F_2 : DiffRewriteOnPort(P, R_1, R_2, port)$ where F_1 and F_2 are forwarding sets of R_1 and R_2.

If both rules are ECMP, we need to be able to distinguish them regardless of which output port one of them chooses. This means that in this case $DiffRewrite(P, R_1, R_2) := \forall port \in F_1 \cap F_2 : DiffRewriteOnPort(P, R_1, R_2, port)$.

Finally, if only one of the rules (assume R_1) is multicast, we still do not know which port will be selected by R_2. Thus, for the same reason as in the previous case, $DiffRewrite(P, R_1, R_2) := \forall port \in F_1 \cap F_2 : DiffRewriteOnPort(probe, R_1, R_2, port)$.

E. Chained tables

Monocle as described so far assumes that each packet enters a switch on one of its ports, gets matched against a forwarding table once, and leaves the switch on one of the ports. In practice, switches may contain a pipeline of tables that each packet traverses (*e.g.*, chaining tables abstraction in OpenFlow 1.1). In such a case, Monocle would require the first table to have additional "fast forward" rules that redirect the probes to the desired tables. Similarly, each table needs to have a catching rule that intercepts the probe. Such design still requires only one probe per rule and in a sense treats chained tables in a single switch as a chain of switches, albeit with a more complicated probe injection mechanism.

F. Unmonitorable rules

For some combinations of rules it is impossible to find a probe packet that satisfies all the aforementioned constraints, as can be seen in the following examples.

First, a rule cannot be monitored if it is completely hidden by higher-priority rules. For example, one cannot verify the presence of a backup rule if the primary is actively forwarding packets. Similarly, a rule is impossible to monitor if it overrides lower priority rules but does not change the forwarding behavior, *e.g.*, a high-priority exact match rule cannot be distinguished from default forwarding if the output port is the same. If distinguishing such rules is necessary and the network operator is allowed to modify an unused header field of the production traffic[3], Monocle could modify rules by adding header rewrites to force different outcomes of the rules in question. We leave this as a potential future work.

Finally, it is impossible to monitor rules that send packets to the network edge as the probes would simply exit the network. While it is impossible to monitor such egress rules, many deployments (*e.g.*, typically in a datacenter) use hardware switches only in the network core and software switches at the edge (*e.g.*, at the VM hypervisor). This lessens the importance of egress-monitoring — the software switches tend to update their data plane instantly and hardware failures are likely to manifest in the unavailability of the whole machine promptly diagnosed by server monitoring solutions.

In addition to the abovementioned theoretical limitations, there are some practical reasons when Monocle may fail to monitor the network correctly. For example, if the switch does not correctly match packets, detecting problems is nondeterministic. Similarly, if the up-/down-stream switches do not handle PacketOut/In messages according to the specification Monocle may report false failures.

IV. Update monitoring

While monitoring networks in a steady state is important, the configuration is most fragile during policy updates. Monocle treats such periods with special caution and switches to a dynamic mode. In this mode, our system focuses only on rules that change, which lets it generate probes quickly enough to confirm data plane updates almost in real time. Such knowledge is important for controllers enforcing consistent network updates [23], as they cannot update the "upstream" switch until the "downstream" switch finishes updating its data plane. In this section we describe aspects of dynamic monitoring that differ from its static counterpart.

A. Rule additions, modifications, deletions

Generating probes for monitoring rule updates is similar to monitoring a static flow table. In particular, a probe for

[3]While allowing to modify header fields of a production traffic is not possible for general Internet connection providers, it might be feasible in a private datacenter setting.

rule addition is constructed the same way as a steady-state probe assuming that rule was already installed. The only difference is that for some switches, Monocle should tolerate transient inconsistencies (*e.g.*, monitored rule missing from the data plane) and should not raise an alarm instantly. Instead, Monocle signals to the controller that the rule is safely in the data plane once the transient inconsistency disappears.

Similarly, a rule deletion is treated as the opposite of installation. We look for a probe that satisfies the same conditions. However, rule deletion is successful only when the probe starts hitting actions of an underlying lower-priority rule. Next, rule modifications keep the match and priority unchanged. This means that the probe will always hit the original or the new version of the rule, regardless of other lower priority rules in the flow table. As such, we simply make a copy of the (expected) content of the flow table, adjust it by removing all lower-priority rules, and decrease the priority of the original rule. Afterward, we can use the standard probe generation technique on this altered version of the flow table to probe for the new rule version.

Finally, a single OpenFlow command can modify or delete multiple rules. Probing in such a case is similar to probing for concurrent modification of multiple overlapping rules at the same time. We describe the complications of concurrent probing in the next section, and leave reliable probe generation in the general case for future work. However, by knowing the content of switch flow table, it is possible (at a performance cost) to translate a single command that changes many rules to a set of commands changing these rules one by one, and confirm them separately.

B. Monitoring multiple rules and updates simultaneously

In steady-state, generating a probe for a given rule does not affect other probes. Therefore, Monocle generates and then uses the probes for multiple rules in parallel. However, after catching the probe Monocle still needs to match it to the monitored rule. To solve this problem, we include metadata such as rule under test and expected result to the probe packet payload that cannot be touched by the switches. This allows us to pinpoint which rule was supposed to be probed by the received probe packet. We use this technique in both steady-state and dynamic monitoring modes.

When monitoring simultaneous updates, Monocle must generate probes that work correctly for all already confirmed rules and at the same time for all subsets[4] of unconfirmed rules sent to the switch. This is required because the probe must work correctly even when the switch updates its data plane while other probes are still traveling through the network. As long as the unconfirmed updates are non-overlapping, the updates do not interfere with each other (see Section V-D) and we can generate probes and monitor the updates separately. Unfortunately, in a general case the problem is more challenging. Our current implementation handles unconfirmed overlapping rules by queuing rules that overlap with any yet unconfirmed rule until it is confirmed. We leave probe generation under several unconfirmed overlapping rules as a potential future work.

[4] According to the OpenFlow specification, a switch can reorder flow installation commands if they are not separated by a barrier message. Moreover, some switches do this even for barrier-separated commands [18].

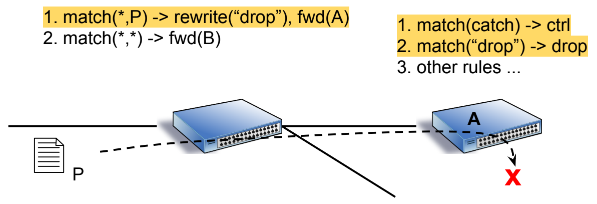


Fig. 4: Drop-postponing method reliably probes for drop rules.

To illustrate why probe generation for multiple overlapping updates is challenging in a general case, consider the controller issuing three rules (in this order):

- low priority $R_1 := match(srcIP = 10.0.0.1, dstIP = *) \rightarrow fwd(1)$
- high priority $R_2 := match(srcIP = *, dstIP = 10.0.0.2) \rightarrow fwd(2)$
- middle priority $R_3 := match(srcIP = 10.0.0.0/24, dstIP = 10.0.0.0/24) \rightarrow drop$

After Monocle receives the rule R_1, it has to send it to the switch, generate a valid probe (*e.g.*, $P_1 := (10.0.0.1, 10.0.0.2)$) and start injecting it. Assume the controller would then install rule R_2. On top of generating probe P_2, Monocle also needs to re-generate P_1 as it is no longer a valid probe for R_1 (if the switch installs R_2 before R_1, P_1 will always be forwarded by R_2, and therefore become unable to confirm R_1). Additionally, Monocle has to invalidate all in-flight probes P_1. And even if Monocle now receives R_3, probing for R_3 is impossible until R_1 is confirmed (assuming the default switch behavior is to drop). Similarly, until rule R_2 is confirmed, probe for R_3 needs to take into account two scenarios – either R_2 has been installed or not. The number of such combinations could rise exponentially, *e.g.*, 5 rules may require considering up to 2^5 outcomes.

C. Drop-postponing

The final improvement is a way to reliably monitor installation of drop rules (rather than relying on negative probing). The method is presented in Figure 4 and relies on modifying rules without affecting the end-to-end forwarding invariants. Specifically, instead of installing a drop rule on a switch, we can install a modified version of the rule which matches the same packets but instead of dropping, it rewrites the packet to a special header and forwards it to one of the switch's neighbors. The aforementioned neighbouring switch must have an appropriate pre-installed rule which matches this special header and drops all matching traffic. Moreover, this drop rule must have a priority lower than the priority of probe-catching rule but sufficiently high that it dominates other rules. This way, all non-probe traffic is dropped one hop later while probe packets are still forwarded to Monocle but with a modified header, which allows it to realize when the drop rule is installed. Finally, after successfully acknowledging the "drop" rule, Monocle can update the rule to be a real drop rule as probing is no longer necessary; this change does not modify the end-to-end network behavior for production traffic.

While this method allows for precise monitoring of drop rule installation, it has several drawbacks. First, it temporarily increases the utilization of a link to the next switch by forwarding all to-be-dropped traffic there for some time. Second, it adds an additional rule modification to really drop packets after acknowledging the temporary "drop" rule.

V. Solving constraints and packet crafting

As discussed in Section III, probe generation involves creating a probe packet that satisfies a given set of constraints. Here we describe how to perform this task by leveraging the existing work on SMT/SAT solvers.

A. Abstracting packets

While constraints from Table I are relatively simple, their complexity is hidden behind predicates such as $Matches(P, R)$ or $DiffRewrite(P, R_1, R_2)$. In particular, when dealing with real hardware, the implementation of packet matching is performing more than a simple per-field comparison. Instead, a switch needs to parse respective header fields and validate them before proceeding further. For example, a switch may drop packets with a zero TTL or an invalid checksum even before they reach the flow table matching step. As such, it is important to generate only valid probe packets.

While the "wire-format" packet correctness can be achieved by enforcing packet validity constraints, doing so is undesirable as such constraints are too complex (*e.g.*, checksums, variable field start positions depending on other fields such as VLAN encapsulation, *etc.*) to be efficiently solved by off-the-shelf solutions. Similarly to other work in this field (*e.g.*, [13], [14], [27]), we use an abstract view of the packet, *i.e.*, instead of representing a packet as a stream of bits with complex dependencies, we treat the packet as a series of (abstract) fields corresponding to well-defined protocol fields (similarly to the definition of OpenFlow rules).

By introducing abstract fields, we can solve the probe generation problem without dealing with the packet wire-format details. As the final step we need to "translate" the abstracted view into a real packet. As we show in the rest of this section, this process contains some technical challenges. While previous work (*e.g.*, ATPG [27]) uses a similar translation, its authors do not go into the details of how to deal with this task.

B. Creating raw packets

The process of creating a raw probe packet given an abstracted header can be handled by the existing packet crafting libraries. The library can handle all relevant assembly steps (computing protocol headers, lengths, checksums, *etc.*). The only remaining task is providing consistent data to the library. In particular, there are two requirements on the abstract data that we provide to the library: (i) limited domains of some fields and (ii) conditionally present fields.

1) Limited domain of possible field values: Some (abstract) packet header fields cannot have arbitrary values because the packet would be deemed invalid by the switch (*e.g.*, DL_TYPE or NW_TOS fields in OpenFlow). Therefore, we need to make sure that our abstract probe contains only valid values. A basic solution is to add an additional "must be one of the following values" constraint on the abstract field. This solution is preferred for small domains (*e.g.*, input port). For domains that are big, we have an alternative solution: Assume that (abstract) header field H can only be fully wildcarded or an exact match (*e.g.*, fully specified). Moreover, assume that the domain of field H contains at least one spare value, *i.e.*, a valid value which is currently not used by any rule in the flow table. Then, we can run the probe generation step without any additional constraints and look at the resulting probe $probe$. If $probe[H]$ contains a valid value for the domain, we leave it as is. However, if $probe[H]$ contains an invalid value, we replace it by the spare value.

Lemma: Replacement of an invalid value of field H by a spare value does not affect the validity of $probe$.

Informal proof: Assume $probe[H]$ contains an invalid value. As all rules in the flow table can contain only valid values from the domain, it is clear that for each rule R in the flow table either $R.match[H] \neq probe[H]$ or $R.match[H] = *$. Setting $probe[H] := spare$ does not change inequalities to equalities and vice versa as we assume $spare$ is a value not used by any rule. Thus, the substitution does not affect the $Matches(probe, R)$ test and therefore preserves validity of the solution with respect to the given constraints. □

2) Some (abstract) packet header fields are included only conditionally: For example, one cannot include TCP source/destination port unless `IP.proto=0x06`. We use a term *conditionally-included* to denote a header field that is present in the header only when another field is present and has a particular value (*e.g.*, TCP source port is present only if the transport protocol is TCP). Similarly, a field that cannot be in the header because of the value of another field (*e.g.*, UDP source port if transport protocol is TCP) is called *conditionally-excluded.* While it is easy to remove all conditionally-excluded fields from the probe solution (*e.g.*, by ignoring their values), we need to make sure that the solution remains valid. A particular concern is whether for any rule R the value of $Matches(probe, R)$ stays the same. We show that the statement holds if rules are well-formed (*i.e.*, they respect conditionally-included fields as required by the OpenFlow specification $\geq$ 1.0.1).

Lemma: Eliminating all conditionally-excluded fields from any valid solution does not change the validity of $Matches(probe, R)$ for any well-formed rule R.

Informal proof: We will eliminate all conditionally-excluded fields one by one. For a contradiction, assume that there exists a conditionally-excluded field H and rule R such that during the elimination of H the validity of $Matches(probe, R)$ changes. Clearly, H cannot be wildcarded in $R.match$ otherwise the validity of $Matches(probe, R)$ would not change. As a consequence, $R.match$ includes field H and as rule R is well-formed (an assumption), $R.match$ has to also include an exact match for parent field H' of H, *i.e.*, the field which determines conditional inclusion of H. We can now finalize the contradiction: If $probe[H'] \neq R.match[H']$, value of $Matches(probe, R)$ is $False$ regardless of the value of $probe[H]$ which contradicts the assumption that leaving out H changes the value of $Matchs(probe, R)$. Further, if $probe[H'] = R.match[H']$, field H is conditionally-included which also contradicts the assumptions. Finally, parent field H' itself might be conditionally-excluded in $probe$; in such case we perform the same reasoning leading to contradiction on its parent recursively. □

C. Solving constraints

Next, we show how to solve the constraints (listed in Table I) that the probe packet needs to satisfy. As it turns out (see Appendix section of technical report [17]), the problem of probe generation is NP-hard. Therefore, our goal is to reuse the existing work on solving NP-hard problems, in particular work on SAT/SMT solvers. While this requires some work (*e.g.*, eliminating for-all quantifiers in *Hit* and *Distinguish* constraints), our constraint formulation is very convenient for SAT/SMT conversion. In particular, we convert the *Hit* constraint to a simple conjunction of several $\neg Matches$ terms and the *Distinguish* constraint to a chain of if-then-else expressions: $If(m_1, d_1, If(m_2, d_2, If(m_3, d_3, ...)))$ where m_i and d_i are in the form of $Matches(P, R)$ and $DiffOutcome(probe, R_{probed}, R)$ for some rule R; this effectively mimics priority-matching of a switch's TCAM. The only remaining part is a way to model $Matches$ and $DiffOutcome$ predicates. $DiffOutcome$ consists of $DiffRewrite$ and $DiffPorts$. Basic set operations allow us to evaluate $DiffPorts$ to either $True$ or $False$ before encoding to SAT. Both $DiffRewrite$ and $Matches$ are similar in nature. Therefore, due to space limitations, we use a simple example to present the encoding only for $Matches$ in context of the first three constraints. For example, assume that all header fields are 2-bit wide (including IP source and destination). The goal is then to generate a probe packet for a low-priority rule $R_{low} := match(srcIP{=}1, dstIP{=}*) \rightarrow fwd(1)$ while using probe-catching rule $R_{catch} := match(VLAN{=}3)$ and assuming a high-priority rule $R_{high} := match(srcIP{=}1, dstIP{=}2) \rightarrow fwd(2)$. We represent probe packet as a sequence of 6 bits $p_1 p_2 \ldots p_6$ where bits 1-2 correspond to IP source, bits 3-4 to IP destination and bits 5-6 to VLAN. Then, *Catch* and *Hit* constraints together are $Matches(P, R_{catch}) \wedge Matches(P, R_{low}) \wedge \neg Matches(P, R_{high})$ and *Distinguish* constraint is simply $True$ as R_{low} is distinguishable from the default drop-all rule. This field-wise corresponds to $(p_{5\text{-}6} = 0b11) \wedge (p_{1\text{-}2} = 0b01) \wedge \neg(p_{1\text{-}2} = 0b01 \wedge p_{3\text{-}4} = 0b10)$, where prefix $0b$ means the binary representation. This can be further expanded to $(p_5 \wedge p_6) \wedge (\neg p_1 \wedge p_2) \wedge (p_1 \vee \neg p_2 \vee \neg p_3 \vee p_4)$, which is a SAT instance.

D. Consider only overlapping rules

Probe packet generation involves generating a long list of constraints which need to be satisfied. To increase solving speed, we strive to simplify the constraints based on the following observation:

Lemma: Let R be a rule that does not overlap with R_{probed}. Then the presence/absence of R in a switch flow table does not affect results of probe generation.

Proof: By definition, rules R_{probed} and R overlap if and only if there exists a packet x that matches both. The negation (*i.e.*, non-overlapping condition) is therefore $\forall x : \neg Matches(x, R_{probed}) \vee \neg Matches(x, R)$. As the expression holds for all packets, it must hold for probe P as well, *i.e.*, $\neg Matches(P, R_{probed}) \vee \neg Matches(P, R)$ holds. Combined with the assumption $Matches(P, R_{probed})$, it implies $\neg Matches(P, R)$. □

Therefore, parts of *Hit* and *Distinguish* constraints related to rule R are trivially satisfied for any probe that matches R_{probed}. As a corollary, all rules that do not overlap with R_{probed} can be filtered out before building constraints. This is a powerful optimization, as typically rules only overlap with a handful of other rules. This observation also determines how complex probe generation problem is in practice. If most rules in the switch do not overlap (*e.g.*, destination IP address forwarding), most constraints are pruned early on, and the solution is quick.

VI. Network-wide monitoring

Monocle design allows it to monitor and generate probes for each switch in the network separately. However, care must be taken to avoid interference among catching rules of different Monocle instances. In particular, each monitored switch could be a downstream switch for multiple other switches, each of them requiring a catching rule on its own. At the same time, these catching rules should not match the probes used to monitor that switch, otherwise the catching rules at the monitored switch would intercept all probes instead of letting them match the monitored rule.

To overcome this problem, a single reserved value of the probe-catching header field is no longer sufficient. Instead, we propose two solutions that offer a trade-off between the number of header fields that need to be reserved for monitoring and the additional load imposed on the control channel.

The first solution is similar to [2]. The solution reserves a single header field H for monitoring and uses a set $Reserved$ of reserved values of this field, $Reserved = \{S_i : i \text{ } is \text{ } a \text{ } switch\}$. The assumptions are similar to a single-switch probing: (i) production traffic never uses these reserved values of H, and (ii) no rule can rewrite field H. Then, each switch i installs $|Reserved| - 1$ catching rules; a rule matching on $match(H = S_j)$ for each $S_j \in Reserved \backslash \{S_i\}$. According to *Hit* and *Collect* constraints in Table I, the value of H in a probing packet has to be equal S_i — it cannot match any catching rule at the probed switch, but must be intercepted by a catching rule at the downstream switch.

Unfortunately, during monitoring of flow table updates, this method causes all probes (except for the ones dropped at the probed switch) to return to the controller even if they were forwarded by rules other than the probed one. This potentially increases control-channel load as well as forces Monocle to analyze more returned probes.

To address this problem, we propose a second solution at the cost of reserving two header fields H_1 and H_2 for probing. Switch i preinstalls two types of rules used during probing:

1) a (high priority) probe-catch rule $R_{catch} := match(H_1 = *, H_2 = S_i) \rightarrow fwd(controller)$, and

2) (slightly lower priority) rules $R_{filter(j)} := match(H_1 = S_j, H_2 = *) \rightarrow drop$ for all $S_j \in Reserved \backslash \{S_i\}$.

The generated probe needs to have $H_1 = S_{probed}, H_2 = S_{next}$ where S_{probed} and S_{next} are identifiers of the probed and desired downstream switch, respectively. Such a probe is not affected by any catching rule on the probed switch but gets sent to the controller only if it reaches the correct downstream switch. The probe gets dropped by other neighbors of the probed switch so the Monocle receives it back only after the update happened in the data plane.[5]

Both presented solutions have a potential downside: they require as many reserved values of field(s) H and as many catching rules in each switch as there are switches in the network. However, what matters for the first method is that any two connected switches have different identifiers. Finding an assignment of labels to nodes of a graph in a way that no two connected nodes have the same label and the total number of distinct labels is minimum is a well-known vertex coloring problem [20]. While finding an exact solution is NP-hard, doing so (as our evaluation in Section VIII-C2 suggests) is feasible for real-world topologies. Our study of publicly available network topologies [15], [24] shows that at most 9 distinct values are required in network topologies consisting of up to 11800 switches. Moreover, the time required is not crucial as it is a rare effort. Network topology changes, such as addition of new switches or links, trigger catching rule recomputation. Link or switch removals (such as network failures) do not require recomputation; the setup may simply no longer be optimal but it is still working.

The number of identifiers used by the second method can also be reduced in a similar fashion. In this case, however, it is not enough to ensure that two directly connected switches have distinct numbers assigned. Additionally, any pair of switches that have a common neighbor must also have different identifiers. Otherwise the method loses the guarantee that the controller does not receive a probe until the probed rule is modified. As such, the method works best on topologies which do not contain "central" switches with high number of peers. From the algorithmic perspective, we can use the vertex coloring problem solver and just modify the underlying graph; we take the original graph and for each switch, we add fake edges between all pairs of its peers, essentially adding a clique to the graph.

VII. Implementation

We design Monocle as a combination of C++ and Python proxies. Such proxy-based design enables chaining many proxies to simplify the system and provide various functionalities (*e.g.*, improving switch behavior by providing update acknowledgments). Moreover, it makes system inherently scalable — each Monocle proxy is responsible for intercepting only a single switch-controller connection and can be run on a separate machine if needed.

Monocle mainly consists of two proxies — Multiplexer and Monitor. Multiplexer connects to Monitors of all monitored switches and is responsible for forwarding their PacketOut/In messages to/from the switch. Monitor is the main proxy and is responsible for tracking the switch flow table, generating the necessary probes, and sending update acknowledgments to the controller. To reduce latency on the critical path, Monitor forwards the FlowMod messages from the controller as soon as it receives them, and delegates the probe computation to one of its worker processes.

Monocle can use conventional SMT solvers for the probe generation. In particular, we implement conversion for Z3 [6] and STP [8] solvers. However, our measurements indicate that these solvers are not fast enough for our purposes (they are 3-5 times slower than our custom-built solver in the experiments presented in Section VIII-B). While we do not know the exact cause, it is likely that (i) Python version of bindings is slow, and (ii) SMT solvers often try too hard to reduce the problem size before passing it to SAT (*e.g.*, by using optimizations such as bit-blasting [8]). While such optimizations pay off well for large and complex SAT problems, they might be an overkill and a bottleneck for the probe generation task. Thus, we wrote our own, optimized, conversion to plain SAT (we use PicoSAT [1] as a SAT solver). The conversion and PicoSAT binding is written in Cython[6] to be on par with plain C code speed and we use the DIMACS format [5] to represent the CNF formulas as one-dimensional vectors of integers. We use such a single-dimensional representation instead of a more intuitive two-dimensional one (vector of vectors of integers, inner vectors representing disjunctions) because such representation resulted in poor performance – in particular, it necessitated *malloc()-ing* of too many small objects, which was the major bottleneck for the conversion.

Finally, since we do not have access to a real PICA8 switch for our evaluation[7], we create and use an additional proxy placed in front of an OpenVSwitch in one of the experiments. This proxy intercepts, delays and modifies the control plane communication in order to mimic the behavior (rule reordering and premature barrier responses) and update speeds of the PICA8 switch as described in [18].

VIII. Evaluation

In our evaluation, we answer the following questions: (i) How quickly can Monocle detect failed rules and links? (*in a matter of seconds*), (ii) Is steady-state monitoring useful? (*by using Monocle we detected issues with two hardware switches*), (iii) How quick and effective is Monocle in helping controllers deal with transient discrepancies between control and data planes? (*it enables correct execution of consistent network updates [23] by providing accurate feedback on rule installation with only several milliseconds of delay*), (iv) How long does Monocle take to generate probing packets? (*a few milliseconds*), (v) How big is the overhead in terms of additional rules and additional packets being sent/received? (*typically small*), (vi) Does Monocle work with larger networks? (*it does and delays an installation of 2000 paths for only 350 milliseconds*).

[5] Unless the modification affects only rewrite actions, not the output port.

[6] Do not confuse with CPython, the standard Python interpreter.

[7] We already returned a borrowed model used in [18].

Our evaluation focuses on unicast and drop rules in the steady state, as well as all update cases desctibed in Section IV except for drop postponing.

A. Monocle use cases

We start by showcasing Monocle's capabilities in both steady-state and dynamic monitoring modes.

1) Detecting rule and link failures in steady-state: To demonstrate Monocle's failure detection abilities, we conduct an experiment where we monitor the data plane of an HP ProCurve 5406zl switch. We connect this switch with 4 links to 4 different OpenVSwitch instances mimicking a star topology with the hardware switch in the middle. We run OpenVSwitches and Monocle on a single 48-core machine based on the AMD Opteron 8431 Processor. To detect failures, we configure Monocle to monitor the switch with a conservative rate of 500 probes/s (Section VIII-C), re-try sending a probe if there is no response for more than 50ms, and raise an alarm if a given probe is not received after 3 retries. In our first experiment, we install 1000 layer-3 forwarding rules on the HP switch, and let Monocle monitor the switch. Afterwards, we "fail" a random rule (by removing it from the data plane without telling Monocle) and we measure the time it takes for Monocle to detect the failure. We repeat the experiment 1000 times and plot the CDF of the resulting distribution. The results (solid line in Figure 5) suggest that, depending on where the failed rule happens to be with respect to the monitoring cycle (Monocle repeatedly goes through all the monitored rules), Monocle can detect the failure in 150 to 3000 ms.

Next, we study how fast a system built on top of Monocle could detect correlated failures, *e.g.*, failures that affect multiple rules simultaneously. In this experiment, we configure Monocle to raise an alarm only after detecting a given threshold (number) of individual rule failures. During the experiment, we fail multiple rules simultaneously, or, in one case, fail a whole link to which 102 of the installed rules forward to. We again repeat the experiment 1000 times and plot the CDF. As the violet (dash-dot-dot) line in Figure 5 shows that identifying big correlated failures (*e.g.*, a link failure) can be done quickly (on average in 200 ms, out of which 150 ms is the detection timeout). For smaller number of failures and higher thresholds, Monocle requires more time as it is unlikely that many (or, in the extreme case, all) of the failed rules would be covered early on in the monitored cycle.

2) Monocle detects previously unknown switch problems: While running regular experiments to evaluate our system, Monocle surprised us by detecting problems with the two switches we used.

Rare non-deterministic rule failures on an undisclosed switch: When we run the basic steady-state experiments on a switch from a vendor which wanted to be anonymized, Monocle detected a failure with a rate of one in around 500-1500 experimental runs. By looking at the issue, we verified that the reported error is a real problem and that the problem affects rules which were not "failed" in the current run of the experiment but some runs ago. Importantly, the switch believed that the problematic rule was installed

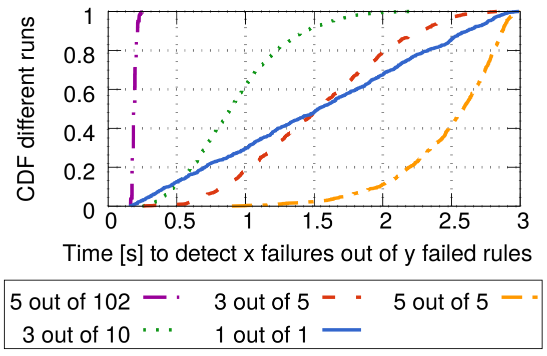


Fig. 5: Time to detect a configured threshold of failures after a rule/link failure with a probing rate of 500 probes/sec and 1000 rules in the switch flow table.

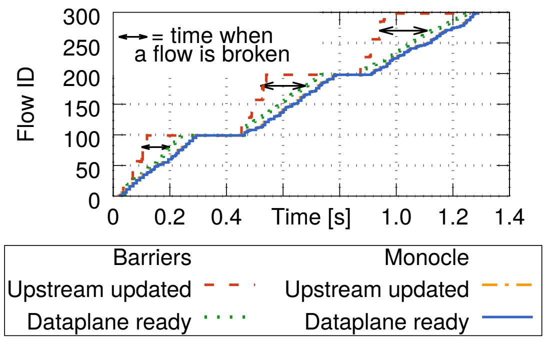
(according the to `ovs-ofctl dump-flows` command run on the switch). However the packets destined to this rule were hitting underlying drop-all rule (verified through the rule packet counters and an absence of packets on the output port).

Due to the rare occurrence we triggered this problem only a dozen times and the problem disappeared once we modified a few unrelated rules on the switch. Our guess is that the problem happens because of a TCAM memory corruption and the issue disappears as soon as the switch is forced to update affected TCAM entry after the switch flow table is modified. This has been confirmed with the vendor which stated that the problem was due to a direct IO memory access to the TCAM and that they fixed it in a newer version of a chip.

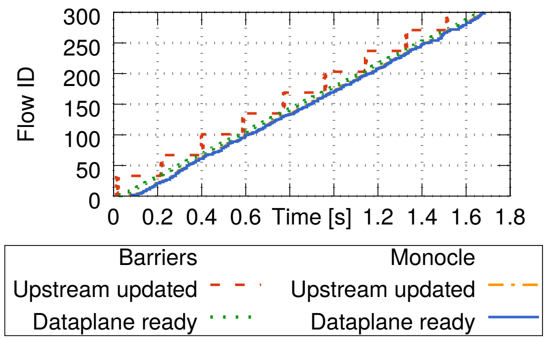
A previously unknown firmware bug in HP 5406zl switch: In one of our experiments which we run but did not include in this article we were "partially" failing a rule and checking whether Monocle can notice such failure. We injected such a "partial" failure by replacing a rule with a rule matching only half of the header space. For example, failing rule $R := match(srcIP = 192.168.0.0/24) \to fwd(1)$ would be performed as (i) insert rule $R' := match(srcIP = 192.168.0.0/25) \to fwd(1)$; and (ii) remove rule R. Upon detecting a failure we would go back to the original state by (i) removing rule R'; and (ii) installing rule R.

Surprisingly, when running the experiment, Monocle quickly and repeatedly detected a failure even after we restored the rules to the original state. Similarly to the previous case, the switch would drop packets destined to rule R even while it reported in CLI that the rule is installed. Our suspicion is that the switch firmware contains a bug related to moving overlapping rules between software and hardware flow tables (when configured to use only hardware flow tables, the switch rejects installing rule R'. Moreover, with the software flow table enabled, the switch reports that rule R is in hardware before we install rule R', and gets moved to software later).

3) Helping controller deal with transient inconsistencies: Some OpenFlow switches prematurely acknowledge rule installation [16], [18]. As Monocle closely monitors flow table updates, it can help the controller to determine the actual time when the rules are active in the data plane. This in turn allows the controller to perform network updates without any transient



(a) HP 5406zl



(b) PICA8 emulation

Fig. 6: Time when flows move to an alternate path in an end-to-end experiment. For both switches, Monocle prevents packet drops by ensuring that the controller continues the consistent update only once the rules are provably in data plane.

inconsistencies. We demonstrate this by using Monocle in a scenario involving an end-to-end network update.

We configure a testbed consisting of three switches S1, S2 and S3 connected in a triangle, and two end hosts – H1 connected to S1, and H2 connected to S2. Switch S3 is the monitored switch exhibiting transient inconsistencies between control and data planes. Initially, we install 300 paths that are forwarding packets belonging to 300 IP flows from H1 to H2 through switches S1 and S2. We send traffic that belongs to these flows at a rate of 300 packets/s per flow. Then, we start a consistent network update [23] of these 300 paths, with the goal of rerouting traffic to follow the path S1-S3-S2. For each flow, we install a forwarding rule at S3 and when it is confirmed, we modify the corresponding rule at S1. We repeat the experiments using two different switches in the role of a probed switch (S3): HP ProCurve 5406zl, and an OpenVSwitch with a proxy that modifies its behavior to mimic the Pica8 switch described in [18]. We always use OpenVSwitch as S1 and S2.

Because both HP 5406zl and Pica8 report rule installations before they actually happen in the data plane, a rule at the upstream switch S1 gets updated in the vanilla experiment too soon and traffic gets forwarded to a temporary blackhole. Figures 6a and 6b show when the packets for a particular flow stop following the old path, and when they start following the new path. The gap between the two lines shows the periods when packets end in a blackhole. In the experiment, a theoretically consistent network update led to 8297 and 4857 dropped packets at HP and Pica8 respectively. In contrast, Monocle ensures reliable rule installation acknowledgments so both lines are almost overlapping and there are no packet drops. The total update time is comparable to the elapsed time without Monocle.

B. Monocle performance

Here, we evaluate Monocle's performance. First, we answer the question whether Monocle can generate probes fast enough to be usable in practice.

Having access to a dataset containing rules from an actual Openflow deployment is hard. As we show in Section V-D, performance strongly depends on composition of the rule set and therefore it is crucial to find a good representative data set. We decided not to use synthetic benchmarks because we wanted to use a data sets that are as close as possible to those in actual deployments. We observe that rules in Access Control Lists (ACL) are those most similar to Openflow rules, since they match on various combinations of header fields. Hence we report the times Monocle takes to generate probes for the rules from two publicly available data sets with ACLs: Stanford backbone router "yoza" configuration [13] containing 2755 rules (referred to as "Stanford"), and ACL configurations from a large-scale campus network [25] with a total of 10958 ACL rules (referred to as "Campus"; we aggregate ACL rules from 300 routers into a single virtual flow table).

Data set	avg [ms]	max [ms]	probes found
Campus	4.03	5.29	10642 / 10958
Stanford	1.48	3.85	2442 / 2755

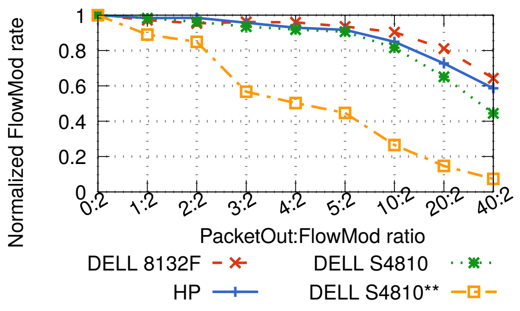
TABLE II: Time Monocle takes to generate a probe

For each dataset we construct a full flow table and then ask Monocle to generate a probe for each rule. In Table II we report average and maximum per-rule probe generation time. On average, Monocle needs between 1.44 and 4.13 milliseconds to generate a probe on a single core of a 2.93-GHz Intel Xeon X5647. This time depends mostly on the number of rules, and not on the rule composition and header fields used for matching. This is the case because the SAT solver is very efficient and the most time-consuming part is to check for the rule overlaps and to send all constraints to the solver. Further, our solution can be easily parallelized both across the switches (separate proxy and probe generator for each switch) and across the rules on a particular switch (probe generation is independent in steady-state).

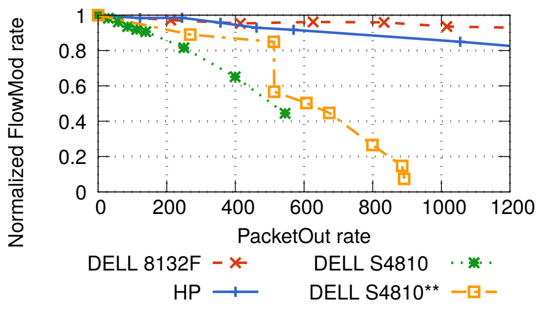
Finally, we also show the number of generated probes compared to the number of rules Monocle is able to find (for reasons why Monocle may fail to find a probe see Section III-F). On both datasets our system was able to generate probes for the majority of rules.

C. Overhead

Next, we show that the overhead of sending and receiving probes is modest when using hundreds or even thousands



(a) Monocle can send 1 to 5 probes per each FlowMod without heavily impacting switch performance.



(b) Switches can handle hundreds to thousands of PacketOuts per second.

Fig. 7: Impact of PacketOut messages on rule modification rate normalized to a baseline rate for each switch. Note that Figure 7b is based on data from Figure 7a instead of a separate measurement of switch performance.

probes per second, depending on the switch. We also show that the catching rules occupy a small amount of TCAM space.

1) PacketIn and PacketOut processing overhead: While it is possible to inject/collect probes via data plane tunnels (*e.g.*, VXLANs) to and from a desired switch, the approach we implemented relies on the control channel. Therefore, it is essential to make sure that the switch's control plane can handle the additional load imposed by the probes without negatively affecting other functionality. We start by estimating what are the maximum PacketOut/PacketIn rates switches can handle when otherwise idle.

To measure the maximum switch PacketOut rate, we issue 20000 PacketOut messages and record the time when the last injected packet arrived at the destination. To measure the maximum PacketIn rate, we install a rule forwarding all traffic to the controller, send traffic to the switch, and observe the message rate at the controller. We repeat both experiment 5 times and report averages; in all cases the standard deviation is lower than 3%.

The observed throughputs are 7006 PacketOut/s and 5531 PacketIn/s on an older HP ProCurve 5406zl switch, 850 PacketOut/s and 401 PacketIn/s on a modern, production grade, Dell S4810 switch, and 9128 PacketOut/s and 1105 PacketIn/s on Dell 8132F with experimental OpenFlow support. If the packet arrival rate is higher than maximum PacketIn rate available at a given switch, the switches start dropping PacketIns. These values assume no other load on the switch.

In the next experiment, we measure overhead of PacketOut messages on the performance of flow table updates. We emulate in-progress network updates by mixing PacketOut and FlowMod messages using the $k : 2$ ratio (to keep the total number of rules stable, the 2 FlowMod messages are: delete an existing rule and add a new one). We vary k and observe how it affects the flow modification rate.

The results presented in Figure 7a show that the performance of all switches is only marginally affected by the additional PacketOut messages as long as these messages are not too frequent. Apart from one exceptional case, the measured switches maintain 85% of their original performance even if each flow modification command is accompanied by up to five PacketOut messages. Dell S4810 with all rules having the same priority (marked with ** in Figure 7a) is more easily affected by PacketOuts because its baseline rule modification rate is higher in such a configuration.

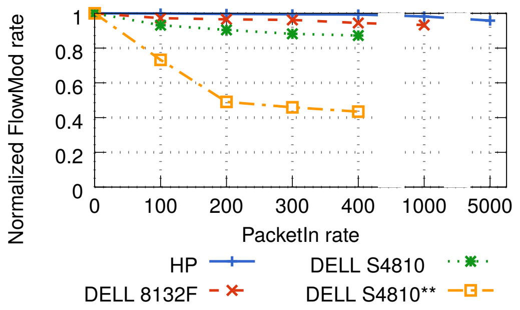


Fig. 8: Impact of PacketIns on rule modification rate normalized to the rate with no PacketIns. Except for Dell S4810 with all rules having equal priority, PacketIns have negligible impact on switches.

We also reuse the same measurement to estimate performance curves for a varying PacketOut rate. This can be done by connecting scatter points $(x, y) = (FmodRate * ratio, FmodRate)$ of different ratios for each switch as plotted in Figure 7b. Switches can handle, depending on the model, hundreds to thousands PacketOuts per second without excessively impacting the rule installation speed.

Finally, we perform an update while injecting data plane packets at a fixed rate of r packets/s causing r PacketIn messages/s and observe how they affect the rule update rate. Figure 8 shows that all switches are almost unaffected by the additional load caused by PacketIn messages. Again, Dell S4810 performance drops by up to 60% when the baseline modification rate is high (all rules have the same priority, marked with **).

2) Number of catching rules required: Recall that our approach for multi-switch monitoring requires multiple probe-catching rules, and these effectively introduce rule overhead. To quantify this overhead, we compute the number of catching rules required for monitoring the network topologies from Internet Topology Zoo [15] and Rocketfuel [24] datasets. To

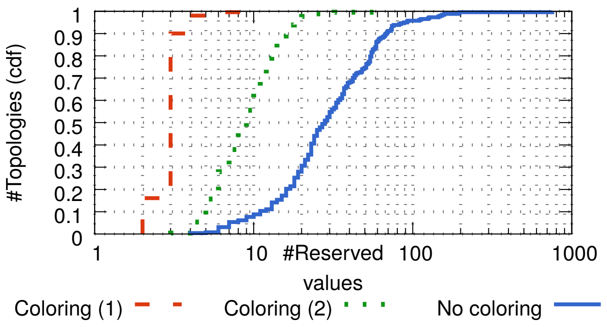


Fig. 9: Number of reserved values in the probing field (also equal to a number of catching rules) for topologies from Topology Zoo [15]. Coloring 1 and 2 correspond to vertex coloring optimization for catching rules with 1 and 2 reserved fields, respectively.

assign probe-catching rules to different switches, we use an optimal vertex coloring solution computed using an integer linear program formulation; solving takes only a couple of minutes to compute the results for all 261+10 topologies.

We start by counting the number of topologies from Topology Zoo that require at least a given number of reserved values of the probe-catching field(s)[8] in the basic version where each switch has a distinct ID, as well as using vertex coloring optimizations for both of the previously explained strategies. Figure 9 presents a couple of interesting observations. First, both vertex coloring optimizations significantly decrease the number of the required values. Moreover, the strategy using just a single reserved field works with a very low number of IDs in practice — up to 9 values are sufficient for networks as big as 754 switches. The final, somewhat unexpected, conclusion is another tradeoff introduced by the technique with two reserved fields. Since the number of IDs it requires is at least as large as the largest node-degree in the network, the number is sometimes high (the maximum is 59).

Rocketfuel topologies confirm these observations — for networks of up to 11800 switches, the technique with a single reserved field requires at most 8 values while the second technique needs to use up to 258 values (note that we use greedy coloring heuristic for the second technique as our ILP formulation runs out-of-memory on our machine). Taking these observations into account, the most practical solution is the one that requires a single reserved field for probing.

D. Larger networks

Finally, we show that Monocle can work in larger networks without prohibitive overheads. We do not have access to a large network, therefore, we set up an experiment that consists of a FatTree network built of 20 OpenVSwitches. As before, we add a proxy emulating Pica8 behavior to each of these switches. Further, each ToR switch has a single emulated host underneath, running a hypervisor switch that implements reliable rule update acknowledgments (also implemented as a proxy on top of OpenVSwitch). For comparison, we construct the same FatTree, but consisting of 28 (ideal) switches with reliable acknowledgments. We ignore the data-plane traffic to avoid overloading the 48-core machine we use for the experiment. Monocle is realized as a chain of three proxies per switch. As already mentioned, the proxies are highly independent and the problem can be easily parallelized. Probe generation for each switch is done in two threads.

[8] Which is the same as the number of probe-catching rules that must be installed on a switch, see Section VI

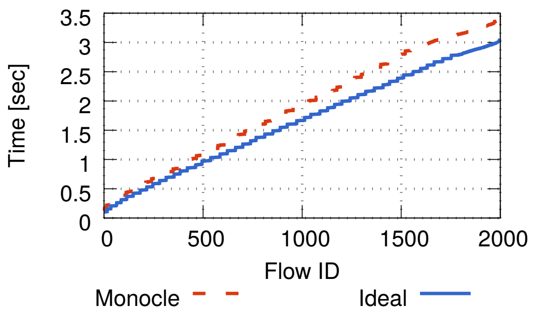


Fig. 10: Batched update in a large network. Monocle provides rule modification throughput comparable to ideal switches.

We carry out an experiment to show how Monocle copes with high load and what is its impact on update latency. In the experiment the controller performs an update installing 2000 random paths in the network, starting 40 new path updates (5-7 rule updates each) every 10 ms.[9] Installation of each path is done in two phases: (i) install all rules except for the ingress switch rule, and (ii) install the remaining rule.

Figure 10 shows that Monocle performs comparably to the network built with ideal switches. Even though the probes have to compete for the control plane bandwidth with rule modifications, the entire update takes only 350 ms longer.

IX. Related work

Ensuring reliable network operation is important for network operators. As such, there exist a large amount of previous work concentrating on different aspects of the problem. In particular, systems like Anteater [19], HSA/NetPlumber [12], [13], SecGuru [11], VeriFlow [14], *etc.*, focus on ensuring that the control plane view of the network corresponds to the actual policy as configured by the network operator. However, problems such as hardware failures, soft errors and switch implementation bugs can still manifest as an obscure and undetected data plane behavior. By systematically dissecting and solving the problem of probe packet generation, Monocle, which is an extension of our earlier short paper on RUM [16], closes the gap and complements these other works. Monocle monitors the packet forwarding done at the hardware level and ensures that it corresponds to the control plane view. While RUM presents a high level idea of using probes for rule monitoring, it does not delve into details of how to generate these probes. This topic is the main focus of Monocle.

[9] Sending all rules at once would cause head-of-line blocking effects on the update. [21]

Tools most similar to Monocle are ATPG [27] and RuleScope [4]; both of them use data plane probes to cross-check switch behavior. However, there are some fundamental differences. To the best of our knowledge, ATPG (i) generates probes taking into the account only *Hit* and *Collect* constraints. It never checks whether the probes actually can *Distinguish* the rule from a lower priority one. As a result, ATPG does not guarantee to detect problems with overlapping rules depicted in Figure 3b. (ii) More importantly, ATPG takes a substantial time to generate the monitoring probes it needs. While this approach works well for static networks, it has serious limitations in highly dynamic SDN networks. In contrast, Monocle copes easily with this case, down to the level that it can observe the switch reconfiguring its data plane during a network update.

RuleScope's advantage over Monocle is its ability to systematically detect and troubleshoot rule priority inversions; Monocle can be adapted to look for such failures as well (by restricting the probe search only to overlaps with other rules) but we leave such implementation and evaluation as a future work. On the other hand, similarly to ATPG, RuleScope does not consider *Distinguish*-ing rules from lower-priority ones based on rule actions and its design is more concerned with stable state monitoring/troubleshooting rather than quick dynamic inspection of the switch.

Also working with a data plane, SDN traceroute [2] focuses on mechanisms that follow packets in an SDN network. Traceroute aims to observe the behavior for a particular packet. Our goal is to observe switch behavior for a particular rule.

Our system is by no means the first to use a SAT solver; other works [11], [19] demonstrate that checking network policy compliance is feasible by converting the problem into a Boolean satisfiability question. Monocle tries to reduce the size and scope of the problem to achieve much finer timescale.

Finally, many systems place a proxy between the controller and the switches [12], [14] to achieve various goals. We take their presence as an additional confirmation that such proxies are a viable design.

X. Conclusions

In this article we address one of the key issues in ensuring reliability in SDN: checking the correspondence between the network state that the SDN controller wants to install, and the actual behavior of the data plane in the network switches. We present a dynamic approach that exercises rules in switches to ascertain that they are functioning correctly. In particular, we show how data plane probe packets should be constructed in a quick and efficient manner. Our system, Monocle, can work on a millisecond timescale to generate probe packets to check when rules are installed in the data plane. In steady-state, it can detect misbehaving rules in switches in a matter of seconds. Finally, Monocle proved its usefulness by discovering switch problems previously unknown to us.

Acknowledgments

We thank Kirill Bogdanov and the CoNEXT 2015 reviewers for excellent feedback. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110. This work is also in part financially supported by the Swedish Foundation for Strategic Research.

References

[1] PicoSAT. http://fmv.jku.at/picosat.

[2] K. Agarwal, E. Rozner, C. Dixon, and J. Carter. SDN traceroute: Tracing SDN Forwarding without Changing Network Behavior. 2014.

[3] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, 2011.

[4] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen. Is Every Flow on The Right Track?: Inspect SDN Forwarding with RuleScope. In *IEEE INFOCOM*, 2016.

[5] D. Challenge. Satisfiability: Suggested Format. *DIMACS Challenge. DIMACS*, 1993.

[6] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2008.

[7] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*, 2013.

[8] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV*, 2007.

[9] B. Heller, C. Scott, N. McKeown, S. Scott, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, K. Zarifis, and P. Kazemian. Leveraging SDN Layering to Systematically Troubleshoot Networks. In *HotSDN*, 2014.

[10] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[11] K. Jayaraman, N. Bjrner, G. Outhred, and C. Kaufman. Automated Analysis and Debugging of Network Connectivity Policies. Technical Report MSR-TR-2014-102, MSR, 2014.

[12] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking using Header Space Analysis. In *NSDI*, 2013.

[13] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, 2012.

[14] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.

[15] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *Journal on Selected Areas in Communications*, 29(9), 2011.

[16] M. Kuźniar, P. Perešíni, and D. Kostić. Providing Reliable FIB Update Acknowledgments in SDN. In *CoNEXT*, 2014.

[17] M. Kuźniar, P. Perešíni, and D. Kostić. Monocle: Dynamic, Fine-Grained Data Plane Monitoring. Technical Report 208867, EPFL, 2015. https://infoscience.epfl.ch/record/208867.

[18] M. Kuźniar, P. Perešíni, and D. Kostić. What You Need to Know About SDN Flow Tables. In *PAM*, 2015.

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

[20] E. Malaguti and P. Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 17(1):1–34, 2010.

[21] P. Perešíni, M. Kuźniar, M. Canini, and D. Kostić. ESPRES: Transparent SDN Update Scheduling. In *HotSDN*, 2014.

[22] P. Perešíni, M. Kuźniar, and D. Kostić. Monocle: Dynamic, Fine-Grained Data Plane Monitoring. In *CoNEXT*, 2015.

[23] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[24] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *SIGCOMM*, 2002.

[25] Y.-W. E. Sung, S. G. Rao, G. G. Xie, and D. A. Maltz. Towards Systematic Design of Enterprise Networks. In *CoNEXT*, 2008.

[26] TechTarget. Carriers bet big on open SDN. http://searchsdn.techtarget.com/news/4500248423/Carriers-bet-big-on-open-SDN, last visited on Oct 5, 2015.

[27] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *CoNEXT*, 2012.