Monocle: Dynamic, Fine-Grained Data Plane Monitoring

(EPFL-REPORT-208867)

Peter Perešíni†, Maciej Kuźniar†, Dejan Kostić‡
† EPFL ‡KTH Royal Institute of Technology
†<name.surname>@epfl.ch ‡dmk@kth.se

ABSTRACT

Ensuring network reliability is important for satisfying service-level objectives. However, diagnosing the cause of 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 handles a variety of scenarios involving existing and new rules, and quickly generates probe packets targeting a particular rule. 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. Also, during network updates Monocle can help controllers cope with switches that exhibit transient inconsistencies between their control plane and data plane states.

1. INTRODUCTION

Ensuring network reliability is paramount. Software-Defined Networks (SDNs) are being widely deployed, and OpenFlow is a popular protocol for configuring network elements with forwarding rules that dictate how packets will be processed. Most of SDN benefits (*e.g.*, flexibility, programmability) stem from its logically centralized view that is presented to network operators. 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 [8], 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 ([10–12, 24]), 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 [16]), through systematic problems (switches incorrectly implementing the specification, *e.g.*, ignoring priority field in OpenFlow [16]), to hardware failures (*e.g.*, soft errors such as bit flips, line cards not responding, *etc.*) and switch software bugs [24]).

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 [24], *etc.*) or periodically collect switch forwarding statistics. We argue that these methods are insufficient – ping/traceroute cannot reveal the problem if it does not affect ICMP traffic. While ATPG provides end-to-end data plane monitoring and can quickly localize problems, it requires substantial time (*e.g.*, minutes to hours [24], depending on coverage) to pre-compute its data plane probes. This delay is too long for modern SDNs where the ever-increasing amount and rate of change demand a quick, dynamic monitoring tool. 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 [6], 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 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 provides a service to the controller which informs when the rule updates sent to the switch finished being installed in hardware. This information could be used by a network controller to enforce consistent updates [19]. In steady-state, Monocle periodically checks all installed rules and reports rules which are 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 be capable of quickly reacting to network reconfigurations, especially if the controller acts on its output. Moreover, the problem is computationally intractable (NP-hard). The reason for this level of hardness is 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 paper are as follows:

1. We present the design and implementation of Monocle, the first 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 monitoring information after 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. In addition, we optimize the way of converting the constraints into a form presented to the off-the-shelf SMT/SAT solvers.
3. We go beyond the state-of-the-art by providing more detail on how the SAT 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.

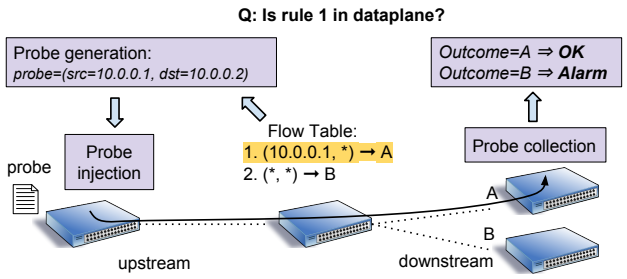


Figure 1: Overview of data-plane rule checking

2. Monocle DESIGN

Monocle is positioned as a layer (proxy) between the OpenFlow controller and the network elements (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 the proxy. 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.

Before we let Monocle monitor the rules, it needs to configure the network by assigning and installing 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; 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.

Figure 2 outlines how the probe packets are created. Monocle leverages its knowledge of the flow table at the switch to create a set of constraints that a probe

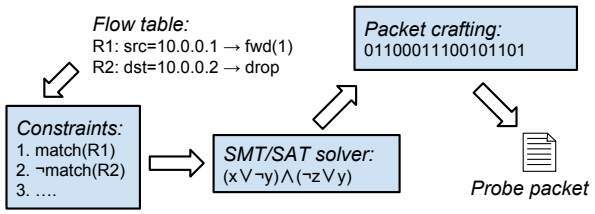


Figure 2: Steps involved in probe generation. Probes for different rules can be generated in parallel.

packet should satisfy. Next, our system converts the constraints into a form that is 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 [12, 24], structured as a collection of header fields. As the final step, Monocle needs to convert the SAT solution, represented in an abstract view, into a real probe packet. Monocle leverages an 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).

3. 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.

3.1 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 transformed 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, and summarize them in Table 1.

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 situation by taking rule priorities into account[1]). As such, for a probe P to be really *processed* according to R_{probed}, P cannot match any rule with a priority higher than the priority of R_{probed}.

Distinguishing the absence of a monitored rule: Even the rules with priority lower than the probed rule R_{probed} affect the probe generation. For example, if the probe matches a low priority rule R_{low} 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 any such rule. Again, 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 (from lowest to highest priority):

- $R_{lowest} := match(srcIP{=}*, dstIP{=}*) \rightarrow fwd(1)$, *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 1: In the case when R_{probed} is missing from the data plane, a lower priority rule R_{LP} with the same outcome cannot be distinguished by probe P if and only if P matches R_{LP} and there is no other rule that matches P and has a priority higher than R_{LP}. To formalize the previous sentence, we define predicate $IsHighestMatch(P, R, OtherRules)$ indicating whether packet P will be processed according to the rule R even if it matches some other rules on the switch. Using $IsHighestMatch$ we can now assert that the probed rules must be distinguishable (*e.g.*, have a different outcome as R_{probed}) from the rule which would process P if R_{probed} is not installed. For simplicity one may think about $DiffOutcome(P, Rule_1, Rule_2)$ simply as a test $Rule_1.outport \neq Rule_2.outport$, but we later expand this definition to accommodate rewrite and

[1] According to the OpenFlow specification, the behavior when overlapping rules have the same priority is undefined. Therefore, we do not consider such a situation.

Hit	$Matches(probe, R_{probed}) \quad \wedge \quad \forall R \in HigherPriority(Rules, R_{probed}) : \neg Matches(probe, R)$
Distinguish	$\forall R \in LowerPriority(Rules, R_{probed}) :$ $IsHighestMatch(probe, R, Rules) \Rightarrow DiffOutcome(probe, R_{probed}, R)$ where $IsHighestMatch(pkt, R, Rules) := Matches(pkt, R) \quad \wedge$ $\big(\forall S \in HigherPriority(Rules, R) : \neg Matches(pkt, S)\big)$
Collect	$Matches(probe, R_{catch})$

Table 1: Summary of constraints that probe packets needs to satisfy when probing for rule R_{probed}.

multicast rules.

Collecting probes: Monocle decides if a rule is present in the data plane based on what happens (referred to as probe *outcome*) to the probe packet. 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 a production traffic will not use these reserved values. We then preinstall 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 among all rules. Naturally, as a last constraint, the probe P has to match the probe-catch rule R_{catch}.

3.2 Unicast rules with rewrites

On top of forwarding, certain rules in the network may rewrite portions of the header before outputting 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 R_{high} is replaced by 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 can 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.

3.3 Drop rules

Since drop rules do not output any packets, we can easily distinguish them 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 monitoring of network updates in Section 4.3.

3.4 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). Both cases can be easily incorporated into our formal framework just by modifying the definition of $DiffPorts$.

These 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). For now, assume that rewriting actions are the same for all ports in the forwarding set.

Moreover, note that drop and unicast rules are just special cases of multicast with zero and one element in their forwarding sets, respectively. This way we only need to discuss three combinations of rules — 2×multicast, 2× ECMP, and multicast + ECMP. In all of these cases, we can distinguish rules based on either rewrites (*i.e.*, $DiffRewrite$ is $True$) [2] or based on their forwarding sets (*i.e.*, $DiffPorts$ is $True$).

If both rules are multicast, a packet will appear on all ports from one of the forwarding sets. Therefore, if there exists any port that 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 distinguish them only if the forwarding sets do not intersect (a probe 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 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: $RewriteOnPort(pkt, R, port)$ defined as the outcome of processing a packet pkt by rule R observed on port $port$. With the aforementioned observations we consider possible cases.

[2] Since drop rules do not output packets, their rewrites are meaningless. We define $DiffRewrite(P, R_{drop}, R') := False$ to fit our theory.

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 probe is different depending which rule processed it. Therefore, $DiffRewrite := \big(\exists probe : \exists x \in F_1 \cap F_2 : RewriteOnPort(probe, R_1, x) \neq RewriteOnPort(probe, R_2, x)\big)$

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 $DiffRewrite := \big(\exists probe : \forall x \in F_1 \cap F_2 : RewriteOnPort(probe, R_1, x) \neq RewriteOnPort(probe, R_2, x)\big)$.

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, $DiffRewrite := \big(\exists probe : \forall x \in F_1 \cap F_2 : RewriteOnPort(probe, R_1, x) \neq RewriteOnPort(probe, R_2, x)\big)$.

3.5 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 rule is actively forwarding packets. Similarly, a rule is impossible to monitor if it overrides lower priority rules but it 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. 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 use 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; this would be promptly diagnosed by existing server monitoring solutions.

4. MONITORING RECONFIGURATIONS

While monitoring steady-state network configuration is important, it is during network updates when the Monocle's ability to quickly generate probes is put under the test. In the dynamic monitoring mode, Monocle focuses monitoring only to the rules being changed. This allows it to confirm almost in a real time when the switch updated the data plane Such knowledge is important for controllers trying to enforce consistent network updates [11], as the controller cannot update "up-

stream" switch sooner than the "downstream" switch finished updating its data plane. In this section we describe aspects of dynamic monitoring that differ from its static counterpart.

4.1 Rule additions, modifications, deletions

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

4.2 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.

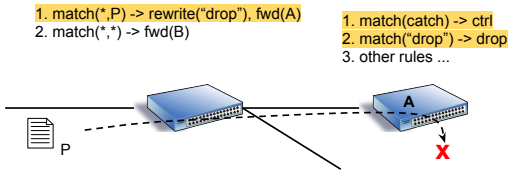


Figure 3: Illustration of drop-postponing method to reliably probe for drop rules.

Unfortunately, monitoring simultaneous updates requires generating probes that work correctly for all already confirmed rules and at the same time for all subsets[3] of unconfirmed rules sent to the switch. This is required because the probe must work correctly even in case 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 5.4) and we can generate probes and monitor the updates separately. Unfortunately, in a general case the problem is more challenging. As an example, 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 sees the rule R_1, it sends it to the switch, generates a valid probe (*e.g.*, $P_1 := (10.0.0.1, 10.0.0.2)$) and starts injecting it. Next, the controller wants to 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). In particular, this requires invalidating all in-flight probes P_1. Next, probing for R_3 is impossible until R_1 is confirmed (assuming the default switch behavior is to drop). Finally, until rule R_2 is confirmed, probe for R_3 needs to consider whether R_2 has been installed. The number of such combinations rises exponentially, *e.g.*, 5 rules require considering 2^5 outcomes.

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.

4.3 Drop-postponing

The final improvement is a way to reliably monitor

[3] According to the OpenFlow specification, a switch can re-order multiple flow installation commands if they are not separated by a barrier message. Moreover, some switches do this even in case the commands are barrier-separated [16].

drop rules (rather than relying on negative probing) presented in Figure 3. 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. Switches need to have a preinstalled drop rule which matches this special header and drops all matching traffic. Moreover, this drop rule has 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 most precise monitoring of drop rule installation, it has the following drawbacks: First, it (temporarily) increases the utilization of a link to the neighboring switch because it forwards 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. Depending on the frequency of drop-rules issued by the controller, this might result in up to 50% control-plane performance degradation (if the controller is installing only drop rules, the Monocle will double the number of rule modifications).

5. SOLVING CONSTRAINTS AND PACKET CRAFTING

As discussed in Section 3, 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.

5.1 Abstracting packets

While constraints from Table 1 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.*, [11, 12, 24]), we use an abstract view of the packet — instead of representing a packet as a stream of bits with complex dependencies, we abstract out the dependencies and treat the packet as a series of (abstract) fields where each field corresponds to a well-defined protocol field (similarly to the definition of OpenFlow rules).

By introducing abstracted 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 [24]) uses a similar translation, its authors do not go into the details of how to deal with this task.

5.2 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.

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 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 field fld can be only fully wildcarded or fully specified. Moreover, assume that the domain of fld 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 result *probe*. If $probe[fld]$ contains a valid value for the domain, we leave it as is. However, if the $probe[fld]$ contains an invalid value, we replace it by the spare value.

Lemma: Previous substitution does not affect the validity of probe.

Proof: Assume $probe[fld]$ contains an invalid (*e.g.*, out-of-domain) 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 $probe[fld] \neq R.match[fld]$ or $R.match[fld] = *$. Setting $probe[fld] := spare$ 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 it preserves validity of the solution with respect to the given constraints.

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 can be present in the header only when another field is present and has a particular value (*e.g.*, TCP source port 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.

Proof: We will eliminate all conditionally-excluded fields one by one. For a contradiction, assume that there exists a conditionally-excluded field $exclfld$ that during the elimination changes the validity of $Matches(probe, R)$ for some rule R. Clearly, $exclfld$ cannot be wildcarded in R otherwise the validity of $Matches(probe, R)$ would not change. Because rule R is well-formed and there is an exact match for $exclfld$, R has to also include an exact match for $parfld$ — a parent field of $exclfld$ (*i.e.*, the field which determines conditional inclusion of $exclfld$). However, if $probe[parfld] \neq R.match[parfld]$, value of $Matches(probe, R)$ is $False$ regardless of the value of $probe[exclfld]$ which contradicts the assumptions. Further, if $probe[parfld] = R.match[parfld]$, field $exclfld$ is conditionally-included which also contradicts the assumptions. Finally, $parfld$ itself might be conditionally-excluded in *probe*; in such case we perform the same reasoning leading to contradiction on its parent recursively.

5.3 Solving constraints

Next, we show how to solve the constraints (listed in Table 1) that the probe packet needs to satisfy. As it turns out (see Appendix section of technical report [15]), 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 $p_1p_2 \dots p_6$ where bits 1-2 correspond to IP source, bits 3-4 to IP destination and bits 5-6 to VLAN. Then, *Hit* and *Distinguish* constraints together are $Matches(P, R_{catch}) \wedge Matches(P, R_{low}) \wedge \neg Matches(P, R_{high})$ which 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 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.

5.4 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.

6. NETWORK-WIDE MONITORING

Monocle design allows it to generate probes for, and monitor 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 on the probes used to monitor the switch, otherwise the catching rules at the monitored switch would intercept all probes instead of letting them match the monitored rule.

We propose two solutions that overcome this difficulty and offer a tradeoff between the number of header fields that need to be reserved for monitoring and the additional load imposed on the control channel. Initially, both strategies require assigning each switch i a network wide unique identifier S_i. We later explain a possible optimization to both methods.

The first strategy reserves for monitoring one packet header field H and a set *Reserved* of values of this field, $Reserved = \{S_i : i \text{ is a switch}\}$. The assumption is that real traffic never uses these values in the reserved field and that no rule can rewrite this field.

Then, each switch i installs catching rules matching on $match(H = S_j)$ for each $S_j \in Reserved \backslash \{S_i\}$. According to *Hit* and *Collect* constraints in Table 1, the value of field 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, in the *reconfiguration* mode, this method causes all probes (except for ones dropped at the probed switch) to return to the controller even if they were forwarded by rules other than the probed one. This increases control-channel load as well as forces Monocle to analyze more returned probes.

The second solution addresses the problem of overloading the control channel with probes 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 controller sees it only once[4], which confirms the rule modification.

Thus far, both presented solutions have a potential downside: the number of reserved values of field(s) H is equal to the number of switches in the network. Further, each switch has to have as many catching rules installed as well. However, what really matters for the first method is that no two neighboring switches have the same identifier. Finding an assignment of labels to nodes in a graph such that no two connected nodes have the same label value and the total number of values is minimum is a well-known vertex coloring problem [18]. While finding an exact solution to this problem is NP-hard, doing so is (as our evaluation in Section 8.3.2 suggests) feasible for real-world topologies. Our study of publicly available network topologies [13, 20] shows that at most 9 distinct values are required in networks 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. 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. Algorithm-wise, we can reuse vertex-coloring solver — we take original graph and for each switch, we add fake edges between all pairs of its peers, essentially adding a clique to the graph. Afterward we solve the vertex coloring problem on this modified graph.

7. 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 — Multi-

[4] Unless there are many probes in flight or the modification affect only rewrite actions, not the output port.

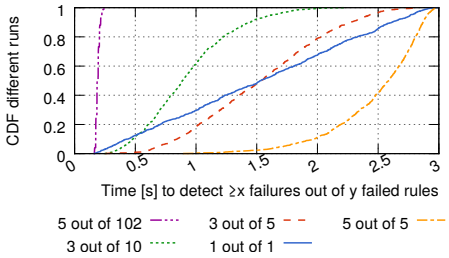
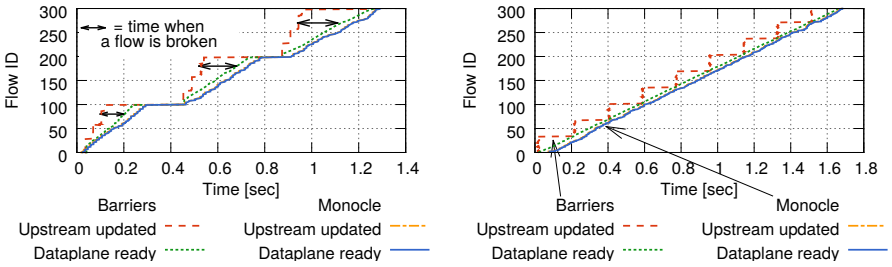


Figure 4: Time to detect a configured threshold of failures after a rule/link failure with a probing rate of 500 probes/sec and 1000 rules in the switch flow table.



(a) HP 5406zl (b) PICA8 emulation

Figure 5: 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.

plexer 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 as soon as it receives them, and delegates the probe computation to one of its workers.

Monocle can use conventional SMT solvers for the probe generation. In particular, we implement conversion for Z3 [5] and STP [7] 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 experiments presented in Section 8.2). 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 to SAT (*e.g.*, by using optimizations such as bit-blasting [7]). 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 is written in Cython (to be on par with plain C code speed) and we use the DIMACS format [4] 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, we create and use an additional proxy placed in front of an OpenVSwitch in one of the experiments. This proxy intercepts and modifies control plane communication between a controller and a correctly working, fast switch to mimic the behavior (rule reordering and premature barrier responses) and update speeds of the PICA8 switch as described in [16].

8. 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*) How quick and effective is Monocle in helping controllers deal with transient inconsistencies? (*it ensures truly consistent network updates by providing accurate feedback on rule installation with only several milliseconds of delay*), (*iii*) How long does Monocle take to generate probing packets? (*a few milliseconds*), (*iv*) How big is the overhead in terms of additional rules and additional packets being sent/received (*typically small*), (*v*) Does Monocle work with larger networks (*it does and delays an installation of 2000 paths for only 350 milliseconds*).

8.1 Monocle use cases

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

8.1.1 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 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 8.3), re-send each probe up to 3 times, and raise an alarm if no probe is received after 150 ms. In our first experiment, we install 1000 layer-3 forwarding rules on the HP switch, and let Monocle monitor the switch. Afterwards, we fail (remove from the data plane) a random rule and we measure the time it takes for Monocle to detect the failure. We repeat the ex-

periment 1000 times and plot the CDF of the resulting distribution. The results (blue line in Figure 4) 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 between 150 ms and 3 seconds.

Next, we study how fast Monocle detects 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 rest of links in Figure 4 show, Monocle quickly identifies the link failure (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 in the monitored cycle.

8.1.2 *Helping controller deal with transient inconsistencies*

Some OpenFlow switches prematurely acknowledge rules installation [14, 16]. 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 inconsistencies. We demonstrate this by using Monocle in a scenario involving an end-to-end network update.

We setup 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 [19] 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 [16]. 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 5a and 5b 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.

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

Table 2: Time Monocle takes to generate a probe

8.2 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. We observe that rules in Access Control Lists (ACL) are the 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 rules from two publicly available data sets with ACLs: Stanford backbone router "yoza" configuration [11] (called Stanford, with 2755 rules), and ACL configurations from a large-scale campus network [21] (Campus, 10958 rules).

For each dataset we construct a full flow table and then ask Monocle to generate a probe for each rule. In Table 2 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 an 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 sent to a particular switch (each probe generation in SAT is independent).

Finally, we also show how many probes compared to the number of rules Monocle is able to find (for reasons why Monocle may fail to find a probe see Section 3.5). In the measured scenarios, our system was able to generate probes for the majority of rules.

8.3 Overhead

Next, we show that the act of sending probes does not overload the switches, and that the catching rules occupy small amount of TCAM space in the switches.

8.3.1 *PacketIn and PacketOut processing overhead*

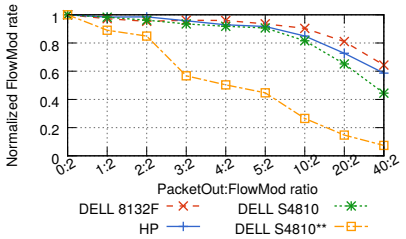


Figure 6: Impact of PacketOut messages on rule modification rate normalized to the rate with no PacketOuts. Following each FlowMod with up to 5 PacketOut messages has small impact on switch performance.

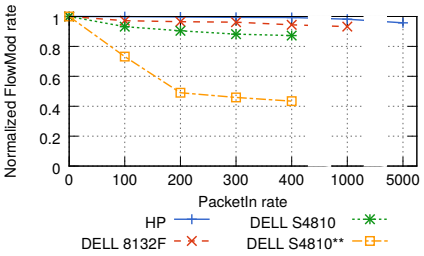


Figure 7: Impact of PacketIns on rule modification rate normalized to the rate with no PacketIns. Except for Dell S4810 with all rules having equal priority, PacketIns have negligible impact on switches.

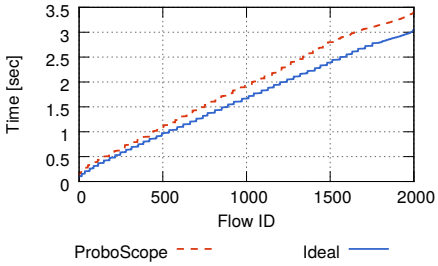


Figure 8: Batched update in a large network. Monocle provides rule modification throughput comparable to ideal switches.

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. To quantify the overhead, we first measure the maximum switch PacketOut rate by issuing 20000 PacketOut messages, and recording when they arrive 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. The rates are 7006 PacketOut/s and 5531 PacketIn/s, averaged over 5 runs on an older, HP ProCurve 5406zl switch. The observed throughputs are 850 and 401 respectively on a modern, production grade, Dell S4810 switch, and 9128 and 1105 on Dell 8132F with experimental OpenFlow support (in all cases the standard deviation is lower than 3%). If the packet arrival rate is higher than maximum PacketIn rate available at a given switch, both switches start dropping PacketIns. These values assume no other load on the switch.

In the second experiment, we emulate in-progress network updates by mixing PacketOut messages and flow modifications using the $k : 2$ ratio (to keep the total number of rules stable, the 2 modifications 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 6 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. All switches maintain 85% of their original performance if each flow modification is accompanied by up to five PacketOut messages. Dell S4810 with all rules having the same priority (marked with ** in Figure 6) is more easily affected by PacketOuts because its baseline rule modification rate is higher in such a configuration.

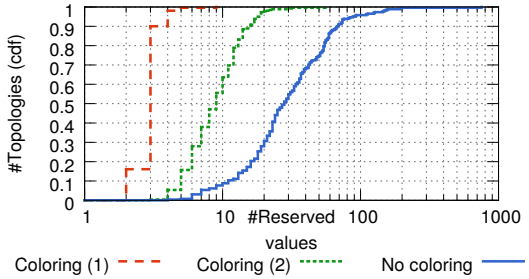


Figure 9: Number of reserved values in the probing field (also equal to a number of catching rules) for topologies from Topology Zoo [13]. Coloring 1 and 2 correspond to colorings for different type of catching rules.

Similarly, 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 7 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, ** in Figure 7).

8.3.2 *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 [13] and Rocketfuel [20] datasets. To 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.

The results presented in Figure 9 are for Topology Zoo, and show how many topologies require a particular number of IDs (number of reserved values of the probe-catching field) in the basic version where each switch

has a distinct ID, as well as with coloring optimization for the both previously explained strategies.

There are a couple of interesting observations. First, both vertex coloring optimizations significantly decrease the number of the required values. Moreover, the technique that requires just one 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.

8.4 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.

We perform an experiment to show how Monocle copes with high load and what is its impact on update latency. In the experiment the controller issues an update installing 2000 random paths in the network. Each update has two phases: (*i*) install all rules except for the ingress switch rule, and (*ii*) update the remaining rule. In the first scenario, we modify all paths in large batches, starting 40 new path updates (5-7 rule updates each) every 10 ms. Figure 8 shows that Monocle performs comparably to the network built of the 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.

9. RELATED WORK

Ensuring reliable network operation is extremely 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 [17], HSA/NetPlumber [10, 11], SecGuru [9], VeriFlow [12], *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 [14], 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.

A tool most similar to Monocle is ATPG [24] that also uses data plane probes to cross-check switch behavior. However, there are some fundamental differences: (*i*) to the best of our knowledge, ATPG 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. (*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.

Also working with a data plane, SDN traceroute [2] concentrates on mechanisms that follow packets in an SDN network. Traceroute aims to observe switch 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 [9, 17] 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 in order to achieve much finer timescale.

Finally, many systems place a proxy between the controller and the switches [10,12] to achieve various goals. We take their presence as an additional confirmation that such proxies are a viable design.

10. CONCLUSIONS

In this paper 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, non-invasive approach that exercises rules in switches to as-

certain 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.

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

[15] 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`.

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

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

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

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

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

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

[22] G. Tseitin. On the Complexity of Derivation in Propositional Calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.

[23] M. N. Velev. Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 310–315, Piscataway, NJ, USA, 2004. IEEE Press.

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

APPENDIX

A. PROBE GENERATION IS NP-HARD

Lemma: Probe-generation is an NP-hard problem.

We prove this by providing a polynomial reduction from SAT problem, *i.e.*, by producing a probe-generation problem for a given SAT problem. In particular, let I be an instance of SAT problem, *i.e.*, I is a formula in conjunctive normal form. Let $x_1, x_2, ..., x_n$ be variables of I. Our reduction uses exactly n header fields (or, equivalently, n bits of a header field which can use an arbitrary wildcard). The reduction is best illustrated on an example $I = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \neg x_3$. We create three high-priority rules, one rule for each disjunction in I. In particular, i-th disjunction logically corresponds to R_i by requiring that the disjunction is true if and only if the probe packet is *not* matching rule R_i, *i.e.*, header fields of rule must match bit 0 for each positive variable, bit 1 for each negative variable and contain wildcard for each variable not present in the disjunction. In our case, $R_1 := (0, 0, *)$, $R_2 := (*, 1, 0)$ and $R_3 := (*, *, 1)$. Then, we ask for a probe packet matching low-priority all-wildcard rule $R_{low} := (*, *, *)$ excluding all higher-priority rules.

Lemma: A probe packet is a valid solution to the aforementioned probe-generation problem if and only if values of probe fields interpreted as values of variables are a valid solution to the original SAT instance I.

We will leave the details of the proof as an exercise for the reader – the only step required is to recognize that the conversion from probe-generation to SAT described in Section 5.3 yields exactly the original SAT problem.

B. ENCODING CONSTRAINTS AS CNF EXPRESSIONS

In this section we briefly describe how to encode constraints into conjunctive normal form (CNF) which is used as an input to all off-the-shelf SAT solvers.

Definition: A formula is in CNF form if it is a conjunction of terms where each term is a disjunction of literals (variables and their negations). An example CNF is $\varphi := x_1 \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2)$.

Let $\varphi_1, \ldots, \varphi_n$ be formulas in CNF form. Then, we can perform following operations and obtain CNF formula as a result

- Conjunction $\varphi := \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n$: The formula is already in CNF form (for math purists: we need to eliminate implicit parentheses around each subformula)
- Disjunction $\varphi := \varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n$: One can repeatedly apply distribution theorem $(\psi_1 \wedge \psi_2) \vee \psi_3 \Leftrightarrow (\psi_1 \vee \psi_3) \wedge (\psi_2 \vee \psi_3)$ to expand the formula into CNF. However, in general, such expansion may lead to an exponential formula size making it impractical. A better approach is to create an *equisatisfiable* formula, *i.e.*, a formula which is satisfied under given valuation of variables if and only if the original formula is satisfied. The idea is to create a new formula by introducing new fresh variables and is usually referred to as Tseitin transform [22]. As an example, consider $\varphi := \varphi_1 \vee \varphi_2$ and a fresh new variable v. We can write $\varphi' := (v \vee \varphi_1) \wedge (\neg v \vee \varphi_2)$ and observe that it is satisfied if and only if at least one of φ_1 and φ_2 is satisfied. It should be mentioned that while it looks that we only swept the problem of disjunctions one level deeper, disjunctions $v \vee \varphi_i$ with v being a literal can be expanded to CNF without an exponential blowup. For longer disjunctions $\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n$, we use an extended form $\varphi' := (v_1 \vee \varphi_1) \wedge (v_2 \vee \varphi_2) \wedge \cdots \wedge (v_n \vee \varphi_n) \wedge (\neg v_1 \wedge \neg v_2 \wedge \cdots \wedge \neg v_n)$
- Implication: $\varphi := \varphi_1 \rightarrow \varphi_2$ is equivalent to $\neg\varphi_1 \vee \varphi_2$
- Substitution with variable $\varphi := x \leftrightarrow \varphi_1$ is simply $(x \rightarrow \varphi_1) \wedge (\varphi_1 \rightarrow x)$ or using previous point: $(\neg x \vee \varphi_1) \wedge (x \vee \neg\varphi_1)$
- Negation $\neg\varphi$: It turns out that we need to support only several special cases of negation:
  - negation of a literal: $\neg(v) = \neg v$, $\neg(\neg v) = v$
  - negation of a CNF consisting only of single disjunction: $\varphi := \neg(l_1 \vee l_2 \vee \cdots \vee l_n)$ is equivalent to $\neg l_1 \wedge \neg l_2 \wedge \cdots \wedge \neg l_n$ where $l_1, \ldots, l_n$ are literals
  - negation of a CNF where each disjunction is trivial: $\varphi := \neg(l_1 \wedge l_2 \wedge \cdots \wedge l_n)$ is equivalent to $(\neg l_1 \vee \neg l_2 \vee ... \vee \neg l_n)$
- If-then-else chain substitution

  $\varphi := \Big(s = if(i_1, t_1, if(i_2, t_2, if(\ldots, if(i_n, t_n, else))\ldots))\Big)$

  First, we substitute all subexpressions as new fresh variables. Then, we use the following construction from [23]:

$$
\begin{aligned}
\varphi = &\Big(\neg i_1 \vee \neg t_1 \vee s\Big) \bigwedge \\
&\Big(\neg i_1 \vee t_1 \vee \neg s\Big) \bigwedge \\
&\Big(i_1 \vee \neg i_2 \vee \neg t_2 \vee s\Big) \bigwedge \\
&\Big(i_1 \vee \neg i_2 \vee t_2 \vee \neg s\Big) \bigwedge \\
&\cdots \bigwedge \\
&\Big(i_1 \vee i_2 \vee \cdots \vee i_{n-1} \vee \neg i_n \vee \neg t_n \vee s\Big) \bigwedge \\
&\Big(i_1 \vee i_2 \vee \cdots \vee i_{n-1} \vee \neg i_n \vee t_n \vee \neg s\Big) \bigwedge \\
&\Big(i_1 \vee i_2 \vee \cdots \vee i_{n-1} \vee i_n \vee \neg else \vee s\Big) \bigwedge \\
&\Big(i_1 \vee i_2 \vee \cdots \vee i_{n-1} \vee i_n \vee else \vee \neg s\Big)
\end{aligned}
$$

$R[i]$	i-th bit of P matches R iff
0	$\neg P[i]$
1	$P[i]$
*	$True$

Table 3: Converting $Matches(P, R)$ to a CNF formula. Resulting formula is a conjunction of per-bit terms and is satisfied if and only if P matches R.

$R_1[i]$	$R_2[i]$	Bit rewrites are different iff
0	0	$False$
0	1	$True$
1	0	$True$
1	1	$False$
*	0	$P[i]$ (*e.g.*, bit needs to be set to 1)
*	1	$\neg P[i]$ (*e.g.*, bit needs to be set to 0)
0	*	$P[i]$
1	*	$\neg P[i]$
*	*	$False$

Table 4: Converting $DiffRewrite(P, R_1, R_2)$ to a CNF formula. Resulting formula is a disjunction of per-bit terms and is satisfied if and only if R_1 rewrites at least one bit of P differently than R_2.

Note that the construction is quadratic in size and therefore very long if-then-else chains should be split by repeatedly substituting some postfix of the chain by a fresh variable.

- Predicate $Matches(P, R)$ is simply a conjunction per-bit terms defined in Table 3. When encoding into SAT, we perform trivial simplification by excluding all $True$ terms from the conjunction.
- Predicate $DiffOutcome$ is a disjunction of $DiffRewrite$ and $DiffPorts$. Note that truth value of $DiffPorts$ can be determined in a preprocessing step and as such we can simplify $DiffOutcome$ to either $True$ or $DiffRewrite$.
- Predicate $DiffRewrite(P, R_1, R_2)$ (which represents expression $rewrite(P, R_1) \neq rewrite(P, R_2)$) is a disjunction (over all bits of P) of expressions from Table 4 (where $P[i]$ represent the variable holding the value of i-th header bit (see $Matches()$ definition) and $R[i]$ is 0, 1 or * depending on whether rule R rewrites bit to 0, 1 or it does not update the bit). Finally, we can perform trivial simplifications on the returned disjunction — remove all $False$ sub-expressions as well as return simply $True$ if one of the sub-expressions is $True$.

Acknowledgments

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.