ProboScope: Data Plane Probe Packet Generation EPFL Technical Report EPFL-REPORT-201824

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

ABSTRACT

Knowing if and when a particular forwarding rule has been installed in a switch is essential for understanding network behavior. This knowledge is missing in current Software Defined Networks, where the controller needs to ensure correct functioning of the network during forwarding state updates. Studies show that an absolute certainty about a rule's presence can be achieved only by exercising it in the data plane. In this work we describe ProboScope, a transparent layer at the controller that relies on carefully constructed data plane packets to provide reliable acknowledgments of rule installations. Driven by the theoretical underpinnings of the way probing rules and packets are constructed, we devise techniques that can generate probe packets in a variety of scenarios, in parallel with simultaneous rule installations, all while paying attention to the already installed rules. Our evaluation shows ProboScope's effectiveness and low overhead. Moreover, it demonstrates two important findings: (i) ProboScope can turn a buggy switch into a corrected one, and perhaps surprisingly (ii) ProboScope can increase the performance of a switch that delays barrier responses in an effort to ensure correct behavior.

1. INTRODUCTION

Software Defined Networking (SDN) owes its ever-increasing acceptance to the simple, flexible concept: allowing control over the installation of forwarding rules in the network elements' flow tables. Software running in a logically centralized controller uses this functionality to make it easy to extend and manage the network. Arguably, reliable SDN forwarding rule installation is the key building block of today's networks.

As the SDNs mature, researchers are paying more attention to the critical transitions from one network-wide forwarding state to another via a so-called *network update*. All these approaches take as a given that the forwarding rules are reliably installed. The OpenFlow specification does not provide positive acknowledgments on a per-rule basis (it was even proposed and rejected[1]). As such, the developers must resort to using the Barrier command to accomplish this task.[2] As we have pointed out [18], this is not the intended application of the Barrier command. To make things worse, several studies [15, 20] show that switches do not fully respect barriers — a few hundred milliseconds can elapse between the barrier response being sent to the controller, and time the rule is physically installed in the data plane (also visible in Figure 1). As a result, serious violations of network policies are possible and they can jeopardize SDN acceptance. Moreover, the problem seems to be persistent — although it was first reported 4 years ago [20], it is still present in modern-day switches [7,12]. Therefore, we expect it to exist in the future as well. Finally, even if the next-generation switches provide reliable update acknowledgments, the existing deployments will likely not replace all devices at once — they are likely to use a mix of new and old (*e.g.*, faulty) switches.

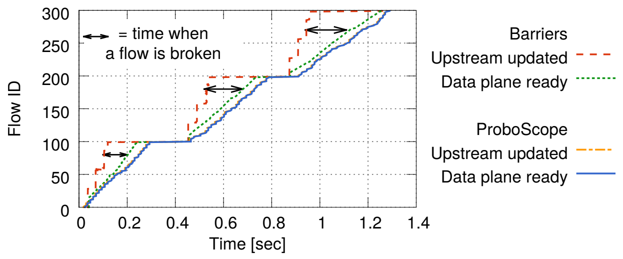


Figure 1: A theoretically consistent network update exhibits prolonged packet drops on a real OpenFlow switch. ProboScope eliminates this problem.

We argue that the critical part in ensuring the success of ongoing and future SDN deployments is in providing a simple primitive: reliable confirmations of forwarding rule installation in a switch data plane. Given all the possible issues that can arise anywhere on the path from the software running in the controller, via the switch control plane, down to the switch data plane, the only certain way of ensuring that a forwarding rule had indeed been installed is to exercise that rule in the data plane of a switch. Our work on RUM [15] outlines the vision for building such a mechanism, and demonstrates

[1]`https://mailman.stanford.edu/pipermail/openflow-discuss/2014-May/005352.html`

[2] In OpenFlow 1.4, it is also possible to use Bundles, but they share many limitations with Barriers.

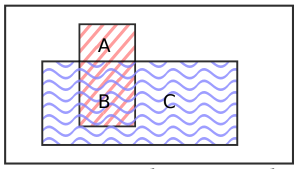


Figure 2: For two overlapping rules with the same forwarding actions, the high-priority (red stripes) rule can be distinguished from a low-priority (blue waves) one only if the probe belongs to A but not B. This distinction is not needed if the forwarding actions are different.

how to provide correct Barrier behavior from otherwise buggy OpenFlow network switches. In RUM, a probe packet needs to be injected toward the switch with the intention of matching on the installed (*i.e.*, "probed") rule, and captured elsewhere in the network.

An integral part of such data plane testing is the task of generating probe packets. The system described in this paper, ProboScope, goes beyond the existing work in that it dissects and solves the problem of probe packet generation. We make several steps, starting with providing the theoretical underpinnings of the way probing rules and packets are constructed. Next, we devise techniques that can generate probe packets in a variety of scenarios and in parallel with simultaneous rule installations, while paying attention to the already installed rules. To generate the probe packet itself, ProboScope leverages the reconstructed switch flow table and the probed rule to formulate the constraints that a probe needs to satisfy, and encodes them as an SMT/SAT problem. Finally, we minimize the network-wide overhead (*i.e.*, extra header fields required for probing and additional, probe "catching" rules) by formulating and solving a vertex graph coloring problem.

Generating probe packets is challenging for a number of reasons. First, it needs to be quick and efficient because the controller may wait for the rule modification to complete, for example during a multi-stage network update. Moreover, the problem is computationally intractable (NP-hard) — we prove that it maps to the satisfiability problem (SAT) in our technical report [14]. The reason for this level of hardness is because the probe packet needs to match the installed rule and avoid certain other rules present in a switch. Figure 2 depicts an example of overlapping rules that needs to be carefully handled when rule actions are identical. 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, rule deletions, multicasting, equal-cost multi-path routing (ECMP) etc. that all have to be carefully dealt with. Third, minimizing the number of rules that are required for network-wide probing among multiple switches is another challenge by itself. Solving this problem is important as we need to minimize the overhead of the extra rules, as well as minimize any requirements placed on the content in the packet headers.

To the best of our knowledge, ProboScope is the only mechanism for ascertaining rule installation in the data plane. ATPG [27] pre-computes the test packets with one of the goals being exercising all *reachable* rules, where our goal is to do rule installation probing on the fly. Moreover, ATPG alone is not sufficient to generate probes that *distinguish* the probed rule from underlying lower-priority rules. For example, in the case of two partially overlapping rules with the same action, one of the probes generated by ATPG might belong to the intersection of the rules (as shown in Figure *2*) — such a probe does not distinguish whether one or both rules are installed. In contrast, ProboScope addresses this problem. Although VeriFlow [10] performs online checking of policy compliance strictly in the control plane, one could use its core engine to build functionality similar to ProboScope. Besides requiring many solutions described in this paper, a VeriFlow-based approach (that works well for traditional forwarding rules) can be significantly slower when rule sets match on various combinations of fields.

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

1. We formulate the theoretical underpinnings of the way probing rules and packets are constructed. We handle unicast, multicast, ECMP, drop rules, rule deletions and modifications. When necessary, we provide proofs that our theoretical foundation is correct.
2. We go beyond the state-of-the-art by providing more detail on how to translate the abstracted view into real packets. In addition, we optimize the way of converting the constraints into a form presented to the off-the-shelf SMT/SAT solvers.
3. We minimize the number of rules needed by ProboScope by formulating and solving a graph vertex coloring problem. Our study shows that only several extra rules per switch suffice in most topologies.
4. Our evaluation demonstrates the ProboScope's efficiency and its low overhead. On FatTree topologies with a few tens of switches, ProboScope introduces only a 7 ms delay per path during an update. In more detail, ProboScope takes between 1.44 and 4.13 milliseconds on average to generate a probe packet on two datasets. As a point of reference, ProboScope is 1.76x to 44x times faster on average than VeriFlow's core engine on datasets with overlapping rules. Moreover, we show the benefits of using ProboScope with fast but incorrect switches (*e.g.*, no packet drops when using ProboScope vs. thousands of packets dropped by a commercial switch, Figure 1), as well as those that are slow but correctly-behaving.

Probes that confirm rule installation enable building systems that provide higher level functionality, such as:

Correct and timely Barrier responses. The barrier

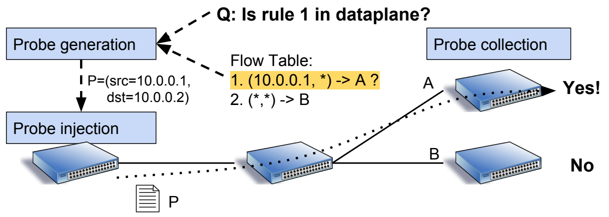


Figure 3: Overview of data-plane rule checking

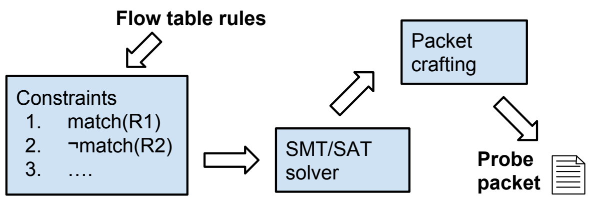


Figure 4: Steps involved in probe generation. Multiple non-overlapping rules can generate probes independently and in parallel.

command can now be correctly implemented. Our short paper [15] outlines how to do that for switches that respond too soon, or even reorder OpenFlow commands beyond the barrier boundary. Moreover, ProboScope makes it possible to avoid abusing the barrier command, and obtain acknowledgments of rule installations.

Regression and interoperability testing. ProboScope is also useful for systematically testing rule installation. We have shown in our earlier work [13] that switches might respond in unexpected ways to commands that contain parameters that are underspecified (*e.g.*, message with a VLAN ID larger than 4095). By using ProboScope, a systematic testing tool can try to ensure that the switch installed a tested rule and that it behaves as expected. Additionally, while running in a real network, our system can also detect the aforementioned problems with deployed switches.

2. PROBOSCOPE OVERVIEW

In this section we provide an overview of ProboScope's operation. ProboScope provides the basic functionality of knowing when a forwarding rule has been installed, modified, or deleted in the data plane of a switch. The treatment of different cases is not the same, and we discuss it later on in the paper.

ProboScope is positioned as a layer (proxy) between the OpenFlow controller and the network elements (switches). It receives OpenFlow commands from the controller and passes them along to the switches, or buffers them for a while. Also, it receives responses to the commands and other messages originating from the switches, and forwards them to the controller. We allow ProboScope to modify the messages it intercepts, as well as to create new messages. This introspective nature of the system allows it to reconstruct a flow table at each switch, similarly to other systems [10, 21].

Figure 3 shows the core mechanism that the system uses to check for successful rule modification. ProboScope 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 directs the switch to send the probe packet back to the proxy. The receipt of the correctly modified probe packet, coming from the appropriate switch, confirms a rule installation and triggers an acknowledgment to the controller.

Before we let ProboScope to check for rule installations, we 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 value of a header field that is otherwise unused by rules in the network. The probe packet then has this particular field set to a value chosen by ProboScope; this value cannot be used by the production traffic. In a network that requires probing for 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 in Section 3.7.

Figure 4 outlines how the probe packets are created. To reason about the probe packet, ProboScope uses an abstract packet view [10, 27], structured as a collection of header fields. ProboScope 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 into a form that is understood by an off-the-shelf satisfiability (SMT/SAT) solver. Finally, ProboScope applies a packet generation library to the output of the solver to create the real probe packet.

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. GENERATING PROBE PACKETS

3.1 Probing a single forwarding rule

The presence of a given rule on a switch can be tested if and only if there exists a packet header that gets transformed by a switch differently depending on whether the probed rule is installed. Based on this observation we formulate four constraints (summarized in Table 1) that the probe packet has to satisfy.

i)	$Matches(probe, R_{probed})$
ii)	$Matches(probe, R_{catch})$
iii)	$\forall R \in HigherPrio(Rules, R_{probed}) : \neg Matches(probe, R)$
iv)	$\forall R \in LowerPrio(Rules, R_{probed}) : IsHighestMatch(probe, R, Rules) \Rightarrow DiffOutcome(probe, R_{probed}, R)$ where $IsHighestMatch(pkt, R, Rules) := Matches(pkt, R) \wedge \big(\forall S \in HigherPrio(Rules, R) : \neg Matches(pkt, S)\big)$

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

Constraint 1: *To probe for a presence of rule R_{probed}, probe packet P needs to match the given rule.*

Only packets that match a rule can be affected by this rule. Therefore, the match of rule R_{probed} has to cover the header of packet P which defines the first constraint.

Constraint 2: *The probe packet P has to match a probe catching rule at the downstream switch.*[3]

ProboScope 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 while trying to avoid affecting the production traffic, we 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 among all rules.

Constraint 3: *The probe P cannot match any rule with a priority higher than the probed rule priority R_{probed}.* The rules installed at a switch can have different priorities and if a packet header matches multiple rules, it is processed according to the one with the highest priority. A probe packet has to avoid all rules with a higher priority than the priority of the probed rule [4]. Otherwise, regardless of the probed rule being installed, the probe will always follow the higher priority rule. Additionally, when probing for rules on multiple switches, the switch we are generating a probe for might also contain a probe-catch rule that needs to be avoided by the probe — in this case we simply treat such a rule as an ordinary high-priority rule.

Constraint 4: *Assuming Constraint 3 holds, in the absence of rule R_{probed} at the switch, the probe packet P must be processed by a lower priority rule with a different outcome.*

Finally, even the rules with priority lower than the probed rule R_{probed} can affect the probe generation. For example, if the probe matches a low priority rule that forwards packets to the same port as R_{probed}, there is no way to determine if R_{probed} is installed. Thus the probe has to avoid any such rule. However, there is an intricate difference between a packet "matching" rule R and "being processed" by rule R. Notably, if we just prevent P from matching all lower-priority rules with the same outcome, we may fail to generate a probe even if such a packet exists as it is illustrated by the next example:

- $R_{lowest} := match(srcIP{=}*, dstIP{=}*) \rightarrow fwd(1)$, *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 rule 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 rule R_{lowest} for such a probe).

The provided example demonstrates that care should be taken to properly formulate the constraint. Formally, we start by defining a predicate indicating whether packet P will be processed according to the rule R as $IsHighestMatch(P, R, OtherRules) := Matches(P, R) \wedge \big(\forall S \in OtherRules :$

$$(S.priority > R.priority) \Rightarrow \neg Matches(P, S)\big)$$

Using this, the probe P is constrained as follows: $\forall L \in LowerPrioRules(R_{probed}) : \big(IsHighestMatch(P, L) \Rightarrow DiffOutcome(P, R_{probed}, L)\big)$. Here the predicate $DiffOutcome(P, Rule_1, Rule_2)$ determines whether $Rule_1$ is distinguishable from $Rule_2$ by sending probe P and observing its forwarding result. One may think about $DiffOutcome$ simply as a test $Rule_1.outport \neq Rule_2.outport$, but we later expand this definition to accommodate rewrite and multicast rules.

Finally, a curious reader may wonder why we do not use the "$IsHighestMatch(probe, R_{probed}, HigherPrioRules(R_{probed}))$" formulation instead of Constraints 1 and 3. The answer is that the two formulations are equivalent and we chose the version with an easier explanation.

3.2 Probing 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

[3] We discuss the cases when the probed rule drops packets or forwards to multiple ports in Section 3.3 and our technical report [14].

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

feasibility of probe generation for some 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 $R_{high2} := match(srcIP{=}10.0.0.1) \rightarrow rewrite(ToS \leftarrow voice), fwd(1)$ 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_{high2} is installed.

In general we can distinguish probes either based on ports they appear on, or by observing the 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, unlike for output ports, checking if the rewrites are different requires special attention. A strawman solution that checks if both rewrite actions modify the same header fields to the same values does not work. Consider a combination of a forwarding rule with another that is also modifying the header:

- $R_1 := match(srcIP{=}*, dstIP{=}*) \rightarrow fwd(1)$, *i.e.*, default forwarding rule; and
- $R_2 := match(srcIP{=}10.0.0.2, dstIP{=}*) \rightarrow rewrite(ToS \leftarrow 0), fwd(1)$, *i.e.*, host 10.0.0.2 is not allowed to use type of service

While the rewrites are structurally different (*e.g.*, $rewrite(None) \neq rewrite(ToS \leftarrow 0)$), they produce the same outcome if the probe packet uses $ToS = 0$. Therefore, to compare the outcome of rewrite actions, we need to take into account not only the rewrites themselves but also the header of probe packet P and how it is transformed by the rules in question. Formally, we say that the rewrites of two rules are equal for a given packet (*i.e.*, $rewrite(P, R_1) = rewrite(P, R_2)$) if they rewrite each bit of the packet the same way, *i.e.*, $\forall i \in 1 \ldots headerlen : \big(BitRewrite(P[i], R_1) = BitRewrite(P[i], R_2)\big)$ and $BitRewrite(P[i], R)$ is either $P[i]$, 0 or 1, depending if the rewrite from rule R leaves the bit unchanged or sets it to a fixed value. After this change, our $DiffRewrite$ definition is sound as it captures exactly the rewrite behavior of the switch.

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 breaks the data plane forwarding.

3.3 Probing with 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, acknowledging drop rule installations brings in a new challenge. It requires detecting when the probes *stop* arriving back at the controller. We call such a situation *negative probing*.

Negative probing carries a risk of prematurely acknowledging drop rule installations. If the rule is not installed but probe packets get lost or delayed for other reasons (e.g. overloaded link, packets damaged during transmission or any other type of failure), ProboScope is unable to determine the difference and confirms the installation. While we believe the presented solution is sufficient in most use cases, we present a fully reliable method in Section 5.

3.4 Handling of multicast / ECMP rules

After discussing the rules that modify header fields and send packets to a single port or drop them, the only remaining type are rules that may 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 $DiffPorts$. The only remaining rule types are those that may forward packets to several ports (*e.g.*, multicast / broadcast and ECMP). We easily incorporate them into our formal framework 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 simplicity, we assume that the 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. 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 $True$) [5] 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 inter-

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

sect (a probe appearing at a port in the intersection does 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, $DiffPorts(R_1, R_2) := \big((F_1 \setminus F_2) \neq \emptyset\big)$.

There is an additional way to distinguish a multicast rule that is not unicast (*i.e.*, $|F_1| \neq 1$) from an ECMP rule. 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 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.

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 Flow modifications and deletions

Probing for rule deletions and modifications is done using the same methods as probing for additions. A rule modification has to keep the match and the 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 logical view) of the flow table and adjust it by removing all lower-priority rules, as well as decreasing 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.

Further, rule deletion can be treated as an opposite of a rule installation. We look for a probe that satisfies the same conditions. However, we know that the rule deletion was successful when the probe starts hitting actions of underlying lower-priority rule.

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 and reliable probe generation for such cases is a part of our future work (we describe complications of concurrent probing in Section 5). However, 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.

3.6 Unacknowledgeable 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 probed if it is completely hidden by higher-priority rules. For example, one cannot probe for the presence of a backup rule if the primary rule is actively forwarding packets. Similarly, a rule is impossible to probe 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 probe for rules that send packets to the network edge as they simply exit the network. While it is impossible to probe for such rules, this might not be an issue depending on a particular scenario: (*i*) datacenter deployments typically use hardware switches only in the network core and use software switches (which can correctly report rule installations) at the edge (*e.g.*, at the VM hypervisor), or (*ii*) edge switches with a spare port can be configured to mirror all egress traffic to this spare port which would allow us to catch copies of all probes exiting the network.

In all cases of unacknowledgeable rules, ProboScope falls back to one of the less reliable control-plane techniques (such as timeouts [15]) to acknowledge the rule.

3.7 Multi-switch probing

The presented solution requires all downstream switches of the probed switch to recognize probe packets and send them to a controller. Therefore, they must have a catching rule installed with a priority higher than other rules, to ensure that the probes are intercepted and not treated as normal traffic.

In practice, a network consists of many interconnected switches and potentially each of them has to be probed. Thus, a probed switch is a downstream switch for other switches and has a catching rule as well, and switches need to have different catching rules. Otherwise, the high priority catching rule at the probed switch would intercept all probes instead of letting them match the probed 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 probing 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 probing one packet header field H and a set $Reserved$ of values of this field, $Reserved = \{S_i : i \textit{ is a switch}\}$. 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 several catching rules: a rule matching on $match(H = S_j)$ for each $S_j \in Reserved \backslash \{S_i\}$. According to Constraints 2 and 3 in Table1, the value of field H in a probing packet has to equal S_i — it cannot match any high priority catching rule at the probed switch, but must be intercepted by a catching rule at the downstream switch. Unfortunately, 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 ProboScope 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 $R_{catch} := match(H_1 = *, H_2 = S_i) \to fwd(controller)$, and
2. (slightly lower priority) rules

$$R_{filter(j)} := match(H_1 = S_j, H_2 = *) \to drop$$

   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[6] , 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 [17]. While finding an exact solution to this problem is NP-hard, doing so is (as our evaluation in Section 7.3.2 suggests) feasible for real-world topologies. Our study of publicly available network topologies [11, 22] 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.

4. 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.

4.1 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 much more than a simple per-field comparison. Instead, a hardware

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

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 solved by off-the-shelf solutions. Instead, similarly to other work in this field (*e.g.*, VeriFlow, HSA, ATPG), 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. However, we cannot avoid generating the real probe packet and therefore as a 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) uses a similar translation, its authors do not go into the details of how to deal with this task.

4.2 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 the 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. Fortunately, we can show that 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 $exclfld$ which changes the validity of $Matches(probe, R)$ during the elimination 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.

4.3 Solving constraints

The last step for constructing a real probe packet is solving the constraints that it needs to satisfy. As it turns out (see Appendix section of technical report [14]), 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 Constraints 3 and 4), our constraint formulation is very convenient for SAT/SMT conversion. In particular, we convert Constraint 3 to a simple conjunction of several $\neg Matches$ terms and Constraint 4 as 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, Constraints 1-3 together are $Matches(P, R_{catch}) \wedge Matches(P, R_{low}) \wedge \neg Matches(P, R_{high})$ which field-wise corresponds to $(p_{5,6} = 0b11) \wedge (p_{1,2} = 0b01) \wedge \neg\,(p_{1,2} = 0b01 \wedge p_{3,4} = 0b10)$. (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. IMPROVEMENTS TO THE BASIC SCHEME

Consider only overlapping rules Probe packet generation involves generating a long list of constraints which need to be satisfied in order to find the probe packet. 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, Constraints 3 and 4 are trivially satisfied for any probe that satisfies Constraint 1. As a corollary it follows that 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.

Probing of multiple rules at the same time The explanation so far considered only probing for a single rule at a time. However, this is highly inefficient as the switch gets underutilized while ProboScope is waiting for probe confirmation. Therefore, it is desired to install and probe for multiple rules at the same time. This approach imposes two challenges.

First, ProboScope needs to distinguish probes confirming different rules. To solve this problem, we include metadata such as the rule under test, and the 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, and limit the number of possibilities to only two: either probed rule is installed and is matching, or a lower-priority rule is matching.

Second, ProboScope needs to generate probes that work correctly for all already confirmed rules and at the same time for all subsets of rules which were sent to the switch but are not yet confirmed. 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 rules are non-overlapping, our approach works because based on the previous lemma, we know that non-overlapping rules have no impact on each other's probe packets. 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 $R_3 := match(srcIP{=}10.0.0.0/24, dstIP{=}10.0.0.3) \rightarrow drop$

After ProboScope 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 the probe packet P_2, ProboScope also needs to re-generate P_1 as it is no longer a valid probe for R_1

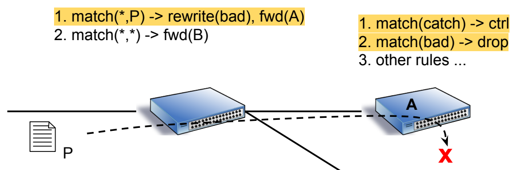


Figure 5: Illustration of the drop-postponing method to reliably probe for drop rules.

(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, ProboScope cannot send rule R_3 to the switch until R_1 is confirmed, otherwise (re-)generating P_1 becomes impossible. Finally, until rule R_2 is confirmed, probe for R_3 needs to consider both possibilities of R_2 being installed or not. 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. Naturally, rules that do not overlap with any yet unconfirmed rule can skip the queue and we send them to the switch unless there is a barrier command forcing the order. On top of that, ProboScope can start solving for probes even while holding the overlapping rules in the queue — ProboScope just assumes that we will install and confirm the rules in order in which they were queued and thus can compute the content of the switch flow table in advance. We leave concurrent probing for several unconfirmed overlapping rules as future work.

Drop-postponing The final improvement is a way to reliably acknowledge drop rules (rather than relying on negative probing) presented in Figure 5. 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 ProboScope but with a modified header, which allows it to realize when the drop rule is installed. Finally, after successfully acknowledging the "drop" rule, ProboScope 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 real (non-probing) traffic.

While this method allows for most precise acknowledgments of drop rule installation, it has 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 ProboScope will double the number of rule modifications).

6. IMPLEMENTATION

We design ProboScope as a set of Python proxies. Such proxy-based design enables chaining many proxies to simplify the system and provide various functionalities (*e.g.*, modify switch behavior and provide acknowledgments in this case). Moreover, it makes system inherently scalable — each ProboScope proxy is responsible for intercepting only a single switch-controller connection and can be run on a separate machine if needed.

ProboScope consists mainly of two proxies — Multiplexer and Prober. Multiplexer connects to Probers of all neighboring switches and is responsible for forwarding their PacketOut/In messages to/from the switch. Prober is the main proxy and is responsible for tracking the switch flow table, generating the necessary probes for the new rules and sending acknowledgments to the controller. In order to offload latency from the critical path, Prober forwards the FlowMod messages as soon as it receives them, and delegates the probe computation to one of its workers.

ProboScope can use conventional SMT solvers for the probe generation. In particular, we implement conversion for Z3 [4] and STP [5] 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 7.2). 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 [5]). While such optimizations pay off well for large problems, they might be an overkill and create as major bottleneck for the probe generation task. Thus, we wrote our own, optimized, conversion to plain SAT (we use PicoSAT [1] as a solver). The conversion is written in Cython (to be on par with plain C code speed) and we use the DIMACS format [3] 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 while evaluating ProboScope (we borrowed it in the past), 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 [12]. We plan to open source our code, and leave implementing multicast and ECMP support for future.

7. EVALUATION

Since we used formal proofs and step by step constraint construction, we leave the validity checking of the generated probes to our unit-tests. Instead, we showcase one use-case for probe generation, and for the remainder of this section we focus on performance. In particular, we perform a simple end-to-end experiment to show the benefits of having reliable rule acknowledgments in one possible use case where ProboScope prevents dropping 8297 packets during a network update. Further, we concentrate on performance/overhead measurements both at the controller and the switch.The results show that: 1) the current prototype running on a single core needs less than 5 ms on average to generate a probe, and 2) modern switches can handle an additional load imposed by injecting and handling probes at a rate of up to 1000 probes per second.

7.1 End to end benefits

We first apply ProboScope in a scenario involving an end-to-end network update. The network consists of three switches S1, S2 and S3 connected in a triangle. There are two end hosts: H1 connected to S1, and H2 connected to S2. Switch S3 is the probed switch. 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 three different switches in the role of a probed switch (S3): HP ProCurve 5406zl, Dell S4810, and an OpenVSwitch with a proxy that modifies its behavior to mimic the Pica8 switch described in [12]. We always use OpenVSwitch as S1 and S2. When using ProboScope, based on the results of measurements in Section 7.3 we use a conservative rate of sending 500 probes/s.

We observe two distinct situations. The first observed behavior is that HP 5406zl and Pica8 report rule installation too early. As a result, a rule at upstream switch S1 gets updated too soon and traffic gets forwarded to a temporary blackhole (because the corresponding rule at S3 is not yet ready). Figures 1 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, ProboScope 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 time without ProboScope.

Data set	ProboScope			VeriFlow core	
	avg [ms]	max [ms]	probes found	avg [ms]	max [ms]
Campus	4.13	5.46	10642 / 10958	10.47	85.10
Stanford	1.44	24.52	2442 / 2755	67.19	86.16

Table 2: Time it takes to generate a probe in ProboScope and equiv. classes in Veriflow for a single rule.

Figure 6a shows a different switch problem. While Dell S4810 sends barrier replies reliably, it also delays them and batches in large groups. Using ProboScope allows us to confirm rule installations much quicker (as soon as they are active in the data plane). This experiment shows the somewhat unexpected benefit of using ProboScope: it turns a correctly functioning, but slow (in terms of barrier responses) switch into a better performing one, while retaining correct behavior.

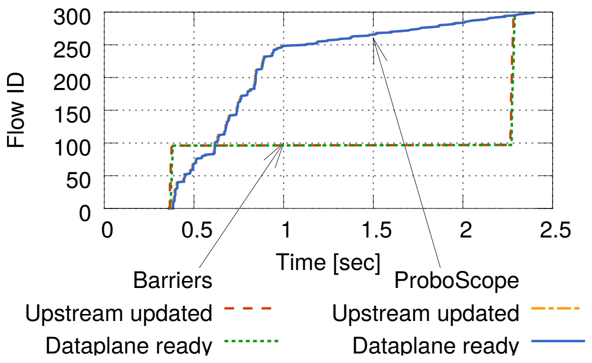
7.2 Controller-side performance

In the following sections we evaluate performance of our system. First, we answer the question if ProboScope can generate probes fast enough to be usable in practice.

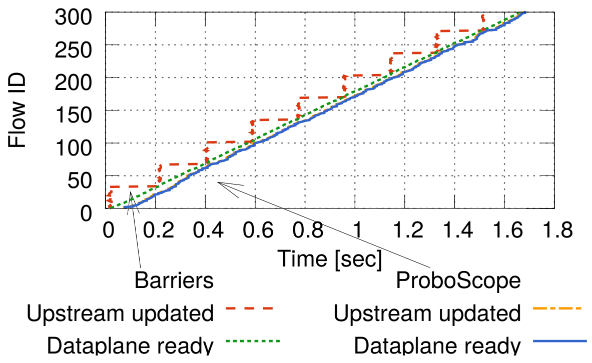
Having access to a dataset containing rules from an actual Openflow deployment is hard. We observe that the ACL rules are the most similar to Openflow rules, since they match on various combinations of header fields. Hence we report the times ProboScope takes to generate probes for rules from two publicly available data sets with ACLs: Stanford backbone router "yoza" configuration [9] (called Stanford, with 2755 rules), and ACL configurations from a large-scale campus network [23] (Campus, 10958 rules).

For each dataset we construct a full flow table and then ask ProboScope to generate a probe for each rule. In Table 2 we report average and maximum per-rule probe generation time. For comparison, we report the average and maximum per-rule equivalence class generation time in VeriFlow (for each rule, we remove it, compute classes with this rule and add it back). Note, that in order to compete with ProboScope, after finding the equivalence classes, VeriFlow would need to do additional work to find the probes.

On average, ProboScope 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



(a) Dell S4810



(b) PICA8 emulation

Figure 6: Time when flows move to an alternate path in an end-to-end experiment. For Dell, ProboScope performs better than Barriers for many flows because the switch batches Barrier responses, which forces the controller to wait longer before updating switch S1. For the two other switches (see also Figure 1), ProboScope prevents packet drops.

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. In contrast, VeriFlow performs well on simple, nonoverlapping forwarding rules, but struggles for complex rules matching on various header fields. 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 ProboScope is able to find (for reasons why ProboScope may fail to find a probe see Section 3.6). In the measured scenarios, our system was able to generate probes for the majority of rules. An example of an unacknowledgeable rule is a low-priority deny ACL rule (dropping packets), that is indistinguishable from the default drop all nonmatching traffic rule in the switch. For these rules, ProboScope can fall back to using timeouts [15].

7.3 Switch-side performance

The technique introduced in this paper requires installing catching rules and active probing of the switch states. Therefore, we need to make sure that the act of sending probes does not overload the switches, and that the catching rules do not occupy too much of the limited TCAM space in the switches.

7.3.1 PacketIn and PacketOut processing overhead

While it is theoretically possible to inject probes via data plane tunnels (*e.g.*, VXLANs) to and from a desired switch, the approach we selected is to rely 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 functionalities. To quantify the overhead, we first measure the maximum 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 mix PacketOut messages and flow modifications using the $k:2$ ratio (to keep the total number of rules stable, the 2 modifications are: delete

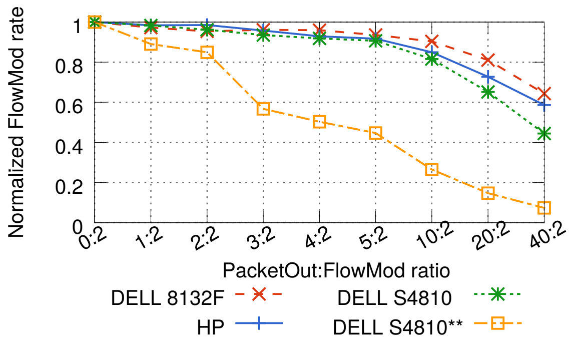


Figure 7: Impact of PacketOut messages on rule modification rate normalized to the rate with no PacketOuts. Following each FlowMod with up to 5 PacketOut messages has small impact on switch performance.

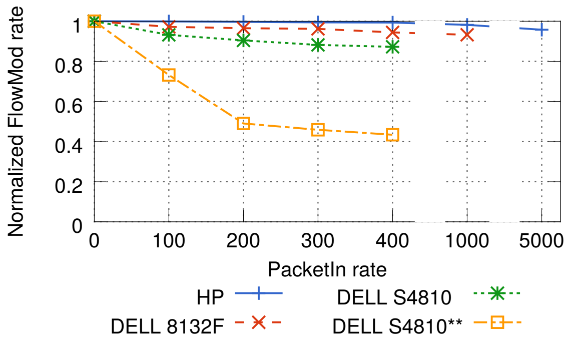


Figure 8: Impact of PacketIns on rule modification rate normalized to the rate with no PacketIns. Except for Dell S4810 with all rules having equal priority, PacketIns have negligible impact on switches.

an existing rule and add a new one). We vary k and observe how it affects the flow modification rate.

The results presented at Figure 7 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 7) is more easily affected by PacketOuts because its baseline rule modification rate is higher in such a configuration.

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 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, ** in Figure 8).

7.3.2 *Number of catching rules required*

Recall that our approach requires distinct switch IDs, and these effectively introduce rule overhead. To quantify this overhead, we compute the number of distinct switch IDs required for probing in the network topologies from Internet Topology Zoo [11] and Rocketfuel [22] datasets. 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 in the basic version where each switch has a distinct ID, as well as with coloring optimizationfor 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.

7.4 Larger networks

Finally, we show that ProboScope 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 is connected to a hypervisor switch, that implements reliable rule update acknowledgments (also implemented as a proxy on top of OvS). 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. ProboScope 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 two experiments to show how ProboScope copes under high load and what is its impact on update latency. In both cases 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 10 shows that ProboScope 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.

In the second scenario we determine the delay induced by a proxy-based system. We start updating the next path only when the current path was completely updated. This way, any additional delay in update confirmation is clearly visible at the end of the experiment. In Figure 11, we see that ProboScope consistently falls behind the ideal switches. However, after 2000 path

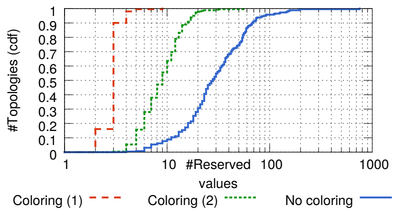


Figure 9: Number of reserved values in the probing field (also equal to a number of catching rules) for topologies from Topology Zoo [11]. Coloring 1 and 2 correspond to colorings for different type of catching rules.

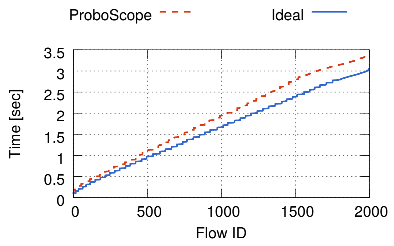


Figure 10: Batched update in a large network. ProboScope provides rule modification throughput comparable to ideal switches.

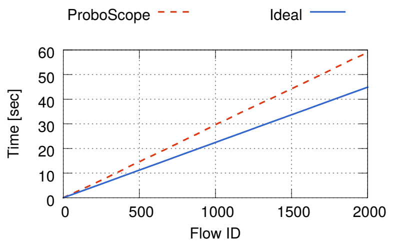


Figure 11: Sequential update in a large network. ProboScope increases the path update latency by 7 ms compared to ideal switches.

updates, the difference is 14 seconds, which means less than 7 ms per path update.

8. RELATED WORK

This paper dissects and solves the key problem of probe packet generation first described in our short paper on RUM [15]. Other systems, such as ATPG [27] have also used probe packets as their focal point but there are some fundamental differences. As explained in Section 1, ATPG tries to test all rules assuming they are installed, while ProboScope can reliably detect if a particular rule is installed. VeriFlow [10] is similar to ProboScope in its online nature, but its goal is different. Namely, it relies on its control-plane view to perform policy compliance checking, while ProboScope offers the missing piece of checking whether the data plane reflects the control plane. In addition, unlike VeriFlow which explicitly slices the packet header space into a set of equivalence classes (sets of packets with the same behavior), ProboScope maps the probe packet generation to the satisfiability problem (SAT) which finds equivalence classes implicitly. As we show in the evaluation, off-the-shelf SAT solvers are well-suited for solving probe generation problem especially for complex rule sets. Moreover, VeriFlow finds equivalence classes, but finding a probe requires additional work.

Other works [6, 16] have used SAT solvers and theorem provers to check network policy compliance. SecGuru [6] works on a control-plane network view and is unable to generate probes to determine if a particular rule is installed in the data plane. Anteater [16] performs static analysis of forwarding tables collected from the data plane, but does not check for rule installation per se. In contrast with these works, we concentrate on checking if/when rules get installed on a single switch. SDN traceroute [2] concentrates on mechanisms to inject and catch packets in an SDN network. This system aims to observe switch behavior for a particular, given, packet. Our goal is to generate a correct packet. Many systems place a proxy between the controller and the switches [8, 21, 25] to achieve various goals. We take their presence as an additional confirmation that such proxies are a viable design.

9. CONCLUSIONS

In this paper we address one of the key issue in ensuring reliability in Software Defined Networking: knowing when a forwarding rule has been successfully installed in the data plane. In particular, we show how data plane probe packets should be constructed in a quick and efficient manner. Our system, ProboScope, is the first such system, and it exhibits significantly better performance than the core of a control-plane only verification tool that could be adapted for checking rule installation.

10. 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] D. Challenge. Satisfiability: Suggested Format. *DIMACS Challenge. DIMACS*, 1993.
[4] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2008.
[5] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV*, 2007.
[6] K. Jayaraman, N. Bjrner, G. Outhred, and C. Kaufman. Automated Analysis and Debugging of Network Connectivity Policies. Technical Report MSR-TR-2014-102, Microsoft Research, 2014.
[7] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic Scheduling of Network Updates. In *SIGCOMM*, 2014.
[8] 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.

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

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

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

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

[13] M. Kuźniar, P. Perešíni, M. Canini, D. Venzano, and D. Kostić. A SOFT Way for OpenFlow Switch Interoperability Testing. In *CoNEXT*, 2012.

[14] M. Kuźniar, P. Perešíni, and D. Kostić. ProboScope: Data Plane Probe Packet Generation. Technical Report EPFL-REPORT-201824, EPFL, 2014. `https://infoscience.epfl.ch/record/201824`.

[15] M. Kuźniar, P. Perešíni, and D. Kostić. Providing Reliable FIB Update Acknowledgments in SDN. Technical Report EPFL-REPORT-201823, EPFL, 2014. To appear at CoNEXT 2014, `https://infoscience.epfl.ch/record/201823`.

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

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

[18] P. Perešíni, M. Kuźniar, M. Canini, and D. Kostić. OpenFlow Needs You! A Call for a Discussion about a Cleaner OpenFlow API. In *EWSDN*. IEEE, 2013.

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

[20] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An Open Framework for OpenFlow Switch Evaluation. In *PAM*, 2012.

[21] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In *OSDI*, 2010.

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

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

[24] 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.

[25] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford. HotSwap: Correct and efficient controller upgrades for Software-Defined Networks. In *HotSDN*, 2013.

[26] 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.

[27] 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 4.3 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 [24]. 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 [26]:

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

  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.

$R[i]$	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.

- 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 $True$.

$R_1[i]$	$R_2[i]$	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.

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.