2017 IEEE 37th International Conference on Distributed Computing Systems

Pronto: Efficient Test Packet Generation for Dynamic Network Data Planes

Yu Zhao[‡], Huazhe Wang[*], Xin Li[*], Tingting Yu[‡] and Chen Qian[*]
[‡] University of Kentucky, U.S.
[*]University of California at Santa Cruz, U.S.
Email:{yzh355,tyu}@g.uky.edu,{huazhe.wang,xli178,cqian12}@ucsc.edu

***Abstract*—Computer networks are becoming increasingly complex today and thus prone to various network faults. Traditional testing tools (e.g., ping, traceroute) that often involve substantial manual effort to uncover faults are inefficient. This paper focuses on fault detection of the network data plane using test packets. Existing solutions of test packet generation either take very long time (e.g., more than one hour) to complete or generate too many test packets that may hurt regular traffic. In this paper, we present Pronto, an automated test packet generation tool that generates test packets to exercise data plane rules in the entire network in a short time (e.g., several seconds) and can quickly react to rule changes due to network dynamics. In addition, Pronto minimizes the number of test packets by allowing a packet to test multiple rules at different switches. The performance evaluation using two real network data plane rule sets shows that Pronto is faster than a recently developed tool by more than two orders of magnitude. Pronto can update the probes for rule changes using less than 1ms while existing methods have no such update function.**

I. Introduction

Network faults are ubiquitous and inevitable [2], [3]. However, testing and debugging a large network is a complex task. A standard enterprise network with tens of switches may contain approximately one million rules in its data plane. In large networks, forwarding rules at routers/switches are determined by interactions of multiple protocols to address routing policies and various service requirements (e.g., VLAN, QoS). Access control lists (ACLs) in routers, switches, and firewalls are designed and configured by different network operation units over a long period of time. It is difficult to ensure that every data plane rule is installed and executed correctly. Many debugging services from network vendors require specialized management tools and hence bring high diagnosis cost to enterprises.

Towards reliable networks, automated tools [18]–[20], [23], [27], [28] that are implemented in software have been proposed to check network reachability (i.e., if any packet from device x can reach another device y in the network) and verify essential network properties such as loop-freedom, (non-)existence of black holes, and network slice isolation. For example, three recent solutions MDD Classifier [15], AP Classifier [26] and Veriflow [20] enable fast identification of the network-wide behavior for certain packet headers. While these approaches can find logic errors in the control plane, such as the controller in Software Defined Networks (SDNs), they are not able to detect faults in the data plane, where problems such as hardware failures, soft errors and switch implementation bugs can still result in erroneous data plane behaviors, including incorrect packet forwarding and processing [25]. A recent survey conducted for enterprise and campus networks [30] shows that data plane software errors and hardware failures are among the most common causes of network outages.

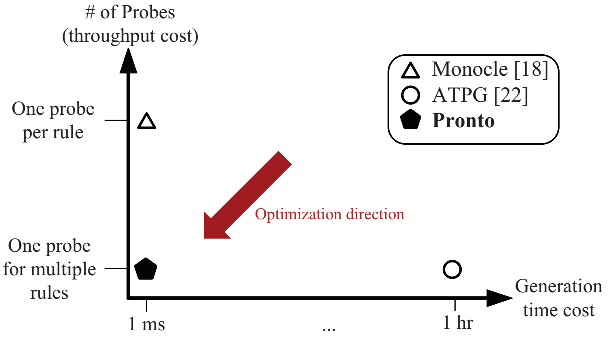


Fig. 1: Pronto vs. existing solutions

This paper focuses on efficient fault detection for the data plane, where the faults include both software errors and hardware failures. The basic approach is to send test packets (also called "probes") and determine the correctness of the data plane based the behavior of the packets. The problem studied in this research is how to generate test packets for efficiently testing data plane rules. We identify three fundamental requirements of a practical test packet generation tool for the data plane.

1) **Completeness.** Every data plane rule should be tested by at least one probe. Completely testing all rules is necessary to locate and resolve data plane faults. This problem is more challenging in current SDNs, such as OpenFlow [24]. This is because the matching fields of OpenFlow rules have more packet header fields than the destination IP. Also, OpenFlow rules may have different priority [21]. A probe may not successfully test a rule if other rules with higher priority also match the probe.
2) **Fast generation and update.** Networks especially SDNs experience frequent rule changes. Recent measurement results show that large SDNs should support hundreds of data plane updates per second [16]. The test packet generation tool should compute and update

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.55

the set of probes in a short time so that the probes can always be consistent with the current data plane rules.

3) **Bandwidth efficiency.** The probes cost network bandwidth since they are transmitted in regular data paths and may compete with other traffic. To reduce the bandwidth overhead, it is desired that one probe can test multiple rules at different switches.

Unfortunately, none of the existing solutions can meet all three requirements stated above. Traditional methods (e.g., ping, traceroute, SNMP) that have been widely used by network operators [1] cannot use arbitrary headers for test packets. Hence some rules cannot be tested by these methods [25], [30]. For example, ATPG [30] is a tool for automatic end-to-end test packet generation by utilizing the header space analysis method [19]. ATPG reads router configurations and generates a minimum set of probes to exercise every rule and link in the network. However, due to the complexity of computing network reachability, ATPG takes tens of minutes to hours to compute the probes for all rules and does not support frequent rule update. The long computation delay prohibits ATPG from being practical for dynamic SDNs. To deal with frequent network changes, a recent work Monocle [25] reduces the scope of diagnosis to a single switch. However, Monocle needs to generate an individual probe for every rule and may cause huge bandwidth cost for large networks. Fig. 1 shows the disadvantages of the two methods: ATPG is computationally slow and does not support rule update, whereas Monocle generates a large number of probes that can induce expensive network throuphput.

We propose an automatic and efficient test packet generation tool called Pronto, for dynamic networks. Pronto utilizes an efficient algorithm to determine which rules can be tested by a given probe, based on the concept of *Atomic Predicate* (AP) [28]. An AP is a concept that specifies a set of packets, which have same forwarding decisions at all switches/routers in the network. Pronto can quickly determine which set of packets is able to traverse a sequence of switches and can be used to test a number of rules at the switches. A reachability table is then constructed to enable a single probe to test multiple rules. To further improve the efficiency of test packet generation, we develop an efficient minimum set cover algorithm to compress the reachability table that allows us to minimize the number of test packets while retaining the coverage of rules. We also propose a probe update approach, focusing on rules that are affected due to network changes, to reduce the number of probes that must be generated on a changed network. We evaluated the performance of Pronto using the data plane rules from two real networks [5] [4]. Experimental results show that Pronto only takes a few seconds to determine the probes for hundreds of thousands of rules and less than one millisecond to change the probes to incorporate each rule update. The brief performance comparison of Pronto and existing work is shown in Fig. 1. In fact, Pronto is not a simple tradeoff solution, but can achieve both minimum number of probes and short computation time without any additional cost.

In summary, this paper makes the following contributions:

- To the best of our knowledge, Pronto is the first data plane testing tool that can simultaneously meet two objectives: *low* probe computation cost and *minimum* probing packets number. A novel and efficient algorithm is proposed and implemented by taking advantages of atomic predicates.
- Pronto provides a probe update approach that can efficiently re-validate the networks in the presence of frequent rule updates. The key idea is to identify rules that are changed or affected by the network updates and then generate *minimum* probes to test *only* these rules. We know of no existing work that can achieve this objective.
- Unlike existing methods that often send packets from the hosts at the network edge, Pronto is intended to *maximize the utilization* of available testing resources (i.e., ports) to test the maximum number of rules while keeping the probe generation cost low. Therefore, Pronto is scalable, transparent, and capable of dealing with complex traditional and software-defined networks.
- Two real network data plane rule sets are used to evaluate Pronto. The results show that Pronto is *faster* than a state-of-the-art tool (i.e., ATPG) by more than two orders of magnitude. In addition, Pronto can generate probing packets for the *updated rules* within 1ms while existing methods are not capable of handling rule updates.

The remainder of this paper is organized as follows. First, we present the system model and the concept of AP in Section II. Then, we present the detailed method and algorithms in Section III and Section IV. Next, we present our experimental evaluation of the new method in in Section V. Finally, we review related work in Section VI, and give our conclusions in Section VII.

II. System modeling and Background

A. System architecture

The network discussed in this work is modeled as an interconnection of switches. End hosts are connected to switches. Each switch has a number of *ports* through which packets can be forwarded to other switches or end hosts. Each switch has a rule table including forwarding, ACL, and VLAN *rules* that determines the forwarding actions applied to an incoming packet. Each rule includes two main parts: *matching fields* and *actions*. The matching fields define a set of packets that match the rule, specified as a region of header space [19]. The actions define the forwarding actions to the packets matching the rule, such as forwarding them to a certain port or dropping them. Rules have *priorities*. If a packet matches multiple rules, the rule with the highest priority will determine the actions applied to the packet. If a packet matches a multiple output actions as a signal rule, the packets will be forwarding to all ports of the rule. There is a manager that can obtain the network topology and all rule tables in the network. To the simplicity of presentation, we assume an SDN model where the manager is an SDN controller. This work may also be applied to legacy IP networks.

Pronto is positioned as an agent program between the controller and network devices (switches and hosts). Pronto can request end hosts and OpenFlow switches to send test packets (probes), which have already been implemented by existing work [30] [25]. Pronto plans the packet header of every probe. Each probe can test a number of rules at switches along the forwarding path of the probe. If the destination – an end host or an OpenFlow switch, receives the probe and informs Pronto, Pronto confirms that these rules work correctly. We call the probe "covers" these rules. Pronto may choose to test all existing rules, a particular set of rules, or newly installed rules.

A data plane fault can be a switch failure, link failure, or incorrect rule. An incorrect rule can be a rule that has a wrong action or a rule that is generated by the controller but does not exist in the data plane. Similar to existing work [30] [25], we consider only the errors in the action fields because they cover a large number of failures. Detection of errors in the matching fields will be future work.

In summary, Pronto provides at least three benefits. First, Pronto efficiently generates probes to exercise all rules in the data plane. Two classes of rules are considered infeasible to be covered and thus eliminated by Pronto: a) a rule that has low priority such that all packets matching it will exercise rules with higher priority and b) a rule that exists in a hardware switch and thus may not be exercised by probes with any headers. Second, Pronto minimizes the number of probes while retaining the coverage of all rules for saving network bandwidth. Third, Pronto not only quickly computes the initial probes but also efficiently updates probes to test rule changes.

B. Atomic Predicates

Following the concepts in [28], each switch has a number of packet filters. Each rule in a switch table matches a subset of packets. The set of packets that can be matched by a rule r_x is defined as the header space of this rule, denoted by ψ_x. In the example of Fig.2, r_{a3}'s header space ψ_{a3} is {10.0.4.0, 10.0.4.1, ... 10.0.4.255}. Each outgoing port is a filter that allows only a subset of packets to forward through, based on the forwarding rules. On each port, the ACL can be viewed as two filters determining the allowed packets for coming in and going out, respectively. If a rule controls the set of packets forwarded through a port, we call the rule *guards* the port. Each filter can be specified by a *predicate*. The set of packets that are allowed to foward through by the filter is evaluated to `true` by the predicate. We represent a predicate as a binary decision diagram (BDD) [7] whose input is a packet header and output is `true` or `false`. A predicate p specifies the set of packets for which p evaluates to `true`. Hence if a packet can travel through a sequence of ports, it is `true` by the conjunction of the predicates on the packet filters of the ports.

Given a set of predicates, we can compute a set of *atomic predicates* (APs), which specifies the minimum set of equivalence classes in the set of all packets [28]. The packets that are evaluated to `true` by the same AP have identical behaviors at all switches. For a set of predicates $\{p_1, p_2, ..., p_k\}$, each AP a_i is in the form $a_i = q_1 \wedge q_2 \wedge ... \wedge q_k$, where $q_j \in \{p_j, \neg p_j\}$ and $j \in \{1, \ldots, k\}$. Every predicate is equal to the disjunction of a subset of APs. Every packet is evaluated to `true` by one and only one AP.

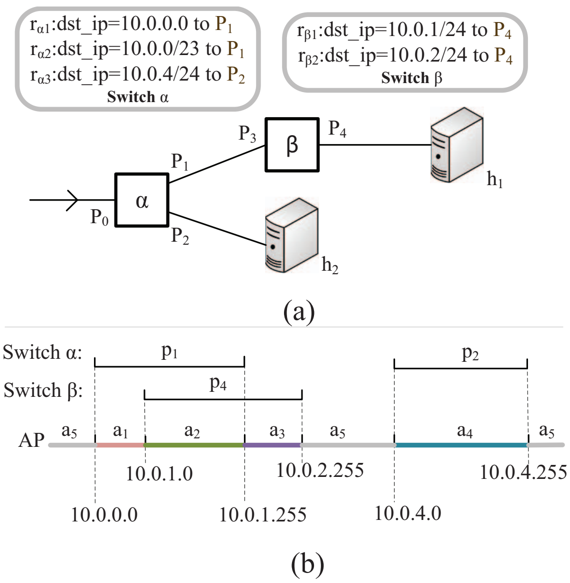


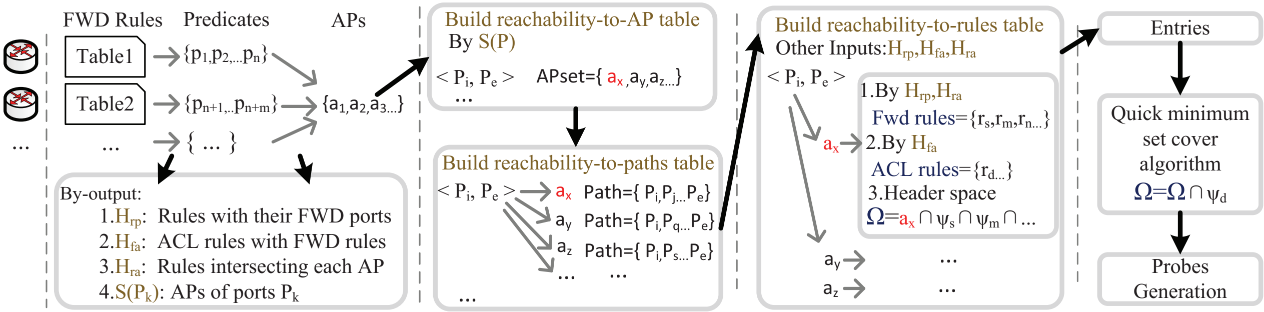
Fig. 2: A sample network including two switches

In the example of Fig.2(a), there are two switches α and β and two hosts h_1 and h_2. The switches α and β include three and two rules, respectively. For simplicity, we assume each rule defines only one matching field (i.e., destination IP). The three outgoing ports P_1, P_2, and P_4 correspond to the predicates p_1, p_2, and p_4, respectively. We can then determine five APs a_1 to a_5, which separate the matching fields into multiple sub-spaces as shown in Fig.2(b). For example, p_1 denotes the set of packets whose destination address is between 10.0.0.0 and 10.0.1.254, and these packets can be forwarded by α to Port P_1. AP $a_2 = p_1 \wedge \neg p_2 \wedge p_4$ indicates that the set of packets can pass Ports P_1 and P_4 but not P_2, whose destination is between 10.0.1.0 and 10.0.1.255. Each predicate is a disjunction of a set of APs, e.g., $p_4 = a_2 \vee a_3$. As such, we can easily represent the packets that pass a port as a union of APs. For example, the set of packets for port P_4 can be represented by $S(P_4) = \{a_2, a_3\}$. Finally, the network reachability of a path can be efficiently computed by intersecting the AP sets of all ports along that path. In the example of Fig. 2, to compute the reachability from α to h_1, we start with the set of all APs $Z = \{a_1, ..., a_5\}$ and then compute the intersection of the AP sets P_1 and P_4, i.e., $Z \cap S(P_1) \cap S(P_4) = \{a_2\}$. Therefore, packets of a_2 can reach h_1 from α, traveling through the outgoing ports P_1 and P_4.

III. Probing Packet Generation

Pronto has two major modules: *probe generation* and *probe update*. In this section, we focus on the probe generation. We will describe the probe update approach in Section IV.

The probe generation module consists of four major steps – see Figure 3. These steps include i) Converting the for-



(a)Converting Rules into the Predicates and APs

(b)Computing the Reachability-to-paths Table

(c)Computing the Reachability-to-rules Table

(d)Minimizing the Number of Probes

Fig. 3: Overview of the probes generation of Pronto

TABLE I: Reachability-to-AP table for Fig. 2

Ingress port	Egress port	AP set
P_0	P_1	a_1, a_2
P_0	P_2	a_4
P_0	P_4	a_2
P_3	P_4	a_2, a_3

TABLE II: Reachability-to-path table for Fig. 2

Ingress port	Egress port	AP set	Path
P_0	P_1	a_1	$P_0,\ P_1$
P_0	P_1	a_2	$P_0,\ P_1$
P_0	P_2	a_4	$P_0,\ P_2$
P_0	P_4	a_2	$P_0,\ P_1,\ P_3,\ P_4$
P_3	P_4	a_2	$P_3,\ P_4$
P_3	P_4	a_3	$P_3,\ P_4$

warding and ACL rules into predicates and APs to construct reachability-to-AP table, ii) Computing reachability-to-path table, iii) Constructing reachability-to-rules table, and iv) Generating probes to cover all rules in the reachability-to-rules table.

Step 1. Pronto first computes the predicates of all ports as well as the APs of the networks. Note that our computation is guided by the rules that guard each port, whereas existing AP computation approaches [28] do not consider rules. As such, Pronto can determine whether a packet can travel from one incoming port to another outgoing port in the network, i.e., the reachability information. The computation of reachability for all pairs of ports in the network can be done in a very short time because it is computed by AP set intersection rather than header space intersection used by the existing method [30]. Note that the AP space is much smaller than the packet header space ($\approx 2^{10}$ versus at least 2^{32}). Table I shows the reachability-to-AP table computed from the example of Fig. 2. This table includes all pairs of ports that allow certain packets to travel, where the ingress port is the incoming side of a port and the egress port is the outgoing side of a port. A pair of ports that do not allow any packets to travel will not appear in the table.

Step 2. Pronto leverages the information in the reachability-to-AP table to compute a reachability-to-path table that associates each AP with a specific path (i.e., the sequence of ports). Table II shows the reachability-to-path table computed from Table I.

TABLE III: Reachability-to-rules table for Fig. 2

Entry #	Ingress	Egress	Header space Ω	Rule list
1	P_0	P_1	10.0.0.0	$r_{\alpha 1}$
2	P_0	P_1	10.0.0.0/23 ∧(¬10.0.0.0)	$r_{\alpha 2}$
3	P_0	P_2	10.0.4/24	$r_{\alpha 3}$
4	P_0	P_4	10.0.2/24	$r_{\alpha 2},\ r_{\beta 2}$
5	P_3	P_4	10.0.1/24	$r_{\beta 1}$
6	P_3	P_4	10.0.2/24	$r_{\beta 2}$

Step 3. For each entry of the reachability-to-path table, Pronto considers the sequence of ports and the rules that guard these ports, and then generates a reachability-to-rules table (e.g., Table III). Each table entry has a header space indicating that any packet in this space can travel from the ingress port to the egress port and cover the listed rules. For example, a packet with the header 10.0.2/24 can travel from P_0 to P_4 and cover rules $r_{\alpha 2}$ and $r_{\beta 2}$.

Step 4. The reachability-to-rules table has two important properties: i) It includes the complete set of all rules that can be covered by probes (i.e., the fifth column). ii) For each single entry, one probe can cover all rules in the entry. Note that a rule may appear in multiple entries. For example, if Pronto uses a probe with the header 10.0.2/24 to cover $r_{\alpha 2}$ and $r_{\beta 2}$ in Entry 4, there is no need to send probes to cover the two rules in Entries 2 and 6. Hence, minimizing the number of probes is actually the minimum set cover problem, one of Karp's 21 NP-complete problems [17]. In this example, the minimum set of probes to cover all rules are four probes in header spaces of Entries 1, 3, 4, and 5. The ingress and egress ports of the probes are also given in the table. Fig. 3 illustrates the workflow of our approach according to more general cases.

Pronto has a tremendous advantage in computation efficiency: it only uses 1.9 seconds to determine probes while ATPG uses 3525 seconds for the same network.

We describe the details of the Pronto algorithm in the sections that follow.

A. Converting Rules into the Predicates and APs

The first step of Pronto is to convert the rules into predicates and APs. Existing AP computation approaches (e.g., AP verifier [28]) only compute the reachability information between two ports. In contrast, Pronto considers the relationship between rules and APs.

Computing predicates. The algorithm of converting rules into the predicates is shown in Algorithm 1. The algorithm takes as input a set of forwarding tables with sorted rules and outputs a set of predicates for all ports in the forwarding tables. These predicates are used to compute APs. A rule's predicate defines the set of packets that can match this rule. We use a BDD to represent a rule [7]. The algorithm iterates through all ports of each forwarding table t and initializes the predicate of each port i to *false*, indicating that no packets can match i' rules yet (lines 2-4). It also creates a variable ψ' to record the header spaces of rules (line 5). However, the predicate may not be equal to the header space of the matching fields. This is because some packets matching the rule can be processed by other rules with higher priority. In the example of Fig. 2, $r_{\alpha 1}$ and $r_{\alpha 2}$ overlap and $r_{\alpha 1}$ has higher priority. Hence the header space of $r_{\alpha 1}$ is 10.0.0.0 and that of $r_{\alpha 2}$ is $[10.0.0.1, 10.0.1.255]$. To compute the header spaces of rules, Pronto first scans the rules from the highest priority to the lowest and records them into ψ'. Next, for each rule Pronto checks the header space of the matching fields and removes the part that overlaps with ψ' (line 9). The remaining header space is the header space of the rule to be added to ψ' (line 11). The time complexity is $O(m)$ for m rules. Note that if the header space of a rule is always `false`, i.e., this rule is impossible to test and can be discarded.

Once we obtain the predicates of all rules, we compute the predicate for each port k (line 10). A port may have two types of rules: forwarding rules and ACL rules. To compute the ports' predicates for the forwarding rules, the algorithm uses a hash table H_{rp} to associate rules with their forwarding ports, in which the key is a port and the value is a set of rules (line 8). Note that a port k may associate with more than one rules. The algorithm then computes the conjunction of the header spaces of the rules (i.e., ψ_j) in k and the address range in p_k to obtain the k's predicate (line 10). Fig.3(a) illustrates the predicates computed from the forwarding rules of each forwarding table. This algorithm also takes $O(m)$ time.

Pronto treats ACL rules differently because they only have two types of actions: *permit* and *deny*. Pronto maintains another hash table H_{fa} whose key is a forwarding rule and value is a set of ACL rules. The rationale behind this is that a probe used to test a forwarding rule can also be used to test the ACL rules by retrieving the key in H_{fa}. To build such a relationship between forwarding rules and ACL rules, for each permit rule r_p, the algorithm computes whether its predicate intersects with the predicate of any forwarding rule r_f. If the result is true, r_p is added to the entry associated with the key r_f in H_{fa}. The deny rules cannot be considered as predicates because they do not allow any packet. We will discuss this problem at the end of this section.

Algorithm 1 Converting Rules into the Predicates

```
Input: A set of forwarding tables $FTB$
Output: A set of predicates $\{p_1, p_2 ... p_n\}$
 1: for each $t \in FTB$ do
 2:     for each $i \in t$.getAllPorts() do
 3:         $p_i \leftarrow false$ /*no packets match the rules for port $i$*/
 4:     end for
 5:     $\psi' \leftarrow false$
 6:     for $r_j \in t$.getAllRules() do
 7:         $k \leftarrow r_j$.port /*retrieve the port of $r_i$*/
 8:         $H_{rp}$.set(k, $r_j$)
 9:         $\psi_j \leftarrow \psi_j \wedge \neg\psi'$
10:         $p_k \leftarrow \psi_j \vee p_k$
11:         $\psi' \leftarrow \psi_j \vee \psi'$
12:     end for
13: end for
```

Computing atomic predicates. Leveraging the method of AP Verifier [28], we compute the APs of the network, $A(\mathcal{P})$, where $\mathcal{P}$ is the set of predicates of all ports, $A()$ is a function to compute the predicates of ports to atomic predicates. For example, $\{a_0, a_1 ... a_n\} = A(\{p_x, p_y\})$ means that the set of APs $\{a_0, a_1 ... a_n\}$ are computed from the predicates of ports P_x and P_y. $A(\mathcal{P})$ can be computed using the following recursion.

$$\begin{cases} A(\{p_i\}) = \{p_i, \neg p_i\} \\ A(\{p_0, p_1 ... p_i\}) = A(\{p_i\}) \sqcap A(\{p_0, p_1 ... p_{i-1}\}) \end{cases} \quad (1)$$

where $\sqcap$ is an operator defined as follows: $\{b_1, b_2, ..., b_l\} \sqcap \{d_1, d_2, ..., d_{l'}\}$ is equal to:

$$\{w_i = b_{i_1} \wedge d_{i_2} | w_i \neq \texttt{false}, i_1 \in \{1, ..., l\}, i_2 \in \{1, ..., l'\}\}$$

The set of APs can be computed very efficiently because in practical networks most conjunctions are always `false`.

If an AP a_i includes a term p_j, we may conclude that packets defined by a_i can pass the port P_j whose predicate is p_j, where j is the port ID. For each port P_i, Pronto stores the its disjunction of a subset of APs, denoted as $S(P_i)$. From the $S(P_i)$, we can easily retrieve the set of ports that allow the packets defined by the AP to pass, denoted as $T[a_i]$. For each port in $T[a_i]$, Pronto checks whether the intersection of any rules guarding the port and the header space of a_i is null, by computing the conjunction of each rule's header space ψ and a_i. We use a hash table H_{ra} to store the result of the intersection (if not null), where the key is an AP a_i and the value is a set of rules resulting from the intersection.

All ACL *deny* rules are tested separately from other rules. This is because when a probe matches a deny rule, it is dropped by the switch and disappears in the network. In this case, Pronto generates a probe to cover some forwarding rules and a deny rule. It first sends the probe to ensure that the forwarding rules work well. Then, if the packet disappears, Pronto concludes that the deny rule is correct.

B. Computing the Reachability-to-path Table

The reachability-to-AP table stores the information related to whether there are some packets allowed to travel from an ingress port P_i to another egress port P_e in the network. Here, we use Table I as an example. If P_e is reachable from P_i, the pair $< P_i, P_e >$ is added to the table (e.g., Columns 1-2 in Table I). The reachability information is computed by the intersection of AP sets as the example $Z \cap S(P_1) \cap S(P_4) = \{a_2\}$ discussed in Sec. II-B. The table also stores the set of APs (e.g., Column 3 in Table I) that define the set of packets that can travel from P_i to P_e. The $S(P)$ used to compute APs is computed in Sec. III-A.

We employ the following optimization strategy during the table construction. For different ingress ports at the same switch, it is very likely that the ACL rules guarding the incoming side of these ports are the same. Some ports may not have ACL rules. In this case, the reachability information will not change even if we vary these ingress ports. Therefore, for ingress ports that are at the same switch and guarded by the same set of ACL rules, we only need to compute the reachability information from one of them to the remaining network. This optimization has reduced the time of reachability analysis by over 50%.

Note that one probe can match at most one AP, because APs define disjoint sets of packets. Thus, if an entry of the reachability-to-AP table includes multiple APs, we need to examine them separately. For this reason, we generate the reachability-to-path table, as shown Table II, in which each entry includes the ingress and egress ports $< P_i, P_e >$ and exact one AP a_i. All packets defined by a_i traveling from P_i to P_e will be forwarded by the same path, shown in the fourth column as a sequence of ports. This path can be easily computed based on reachability-to-AP computing process. Notice that, between the ingress and egress ports $< P_i, P_e >$, a network may have more than one paths. Because AP represents packets' behavior in networks, all packets defined by a_i will only be forwarded by the same path.

C. Computing the Reachability-to-rules Table

The reachability-to-rules table T_r is constructed based on the information from the reachability-to-path table T_p. The algorithm is shown in Algorithm 2. It takes as input the T_p table, as well as the H_{rp}, H_{fa} and H_{ra} obtained from Section III-A. Each entry e in the T_r indicates that there exists at least one probe that can pass a sequence of ports within a header space to cover a list of rules.

There are two key elements in T_r: header space and historical rules (i.e., rule list). Fig. 4 illustrates the computation of the header space and a list of rules for each entry e. The arrow represents the header space. Suppose there are three ports P_1, P_2, and P_3 for the AP a_1 in an entry. By searching the hash table H_{rp} and H_{ra} (lines 5-6), there are four rules r_1, r_2, r_3, and r_4 that intersect a_1, each of which guards a port. The goal is to compute a header space, by which a probe can travel through the path specified in e. The shaded area in Fig. 4 is the header space Ω, which is the intersection of the matching fields of the three rules r_2, r_3, and r_4. If the return value of the intersection is not false (line 8), Pronto can generate a new entry for the T_r (lines 9-14), including the ingress and egress ports, the header space Ω, and the rules r_2, r_3, and r_4. This entry means that if Pronto sends a probe from P_1 to P_3 whose header is in Ω, it can cover three rules r_2, r_3, and r_4 at the same time. Note that there could be multiple shaded areas indicating that the header spaces that can pass all ports on a path. In this case, each header space is a new entry in T_r.

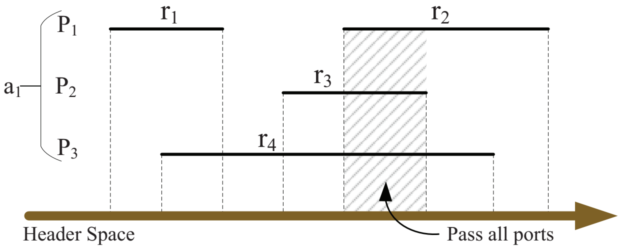


Fig. 4: Compute the header space that can pass all three ports of an AP a_1

Algorithm 2 Computing the Reachability-to-rules Table

```
**Input:** Reachability-to-path Table $T_p$, $H_{rp}$, $H_{fa}$, $H_{ra}$
**Output:** Reachability-to-rules Table $T_r$
1: **for** each table entry $e_p \in T_p$ **do**
2:     $ap \leftarrow e_p$.getAP()
3:     $P \leftarrow e_p$.getPort()
4:     **for** each port $i \in P$ **do**
5:         $R_i \leftarrow H_{rp}$.get($i$) /*the rule set for port $i$*/
6:         $B_i \leftarrow H_{ra}$.get($ap$) $\wedge R_i$ /*a subset of $R_i$*/
7:     **end for**
8:     **if** $\{\psi_a \wedge \psi_b \wedge ... \neq false | r_a \in B_1, r_b \in B_2, \dots\}$ **then**
9:         $e_r \leftarrow T_r$.createNewEntry()
10:        $e_r$.setPorts($e_p$.InPort, $e_p$.OutPort)
11:        $e_r.\Omega \leftarrow ap \wedge \psi_a \wedge \psi_b \wedge ...$
12:        $e_r$.setHeaderSpace($\Omega$)
13:        $e_r$.setFowardRuleSet($\{r_a, r_b ...\}$)
14:        $e_r$.setACLRuleSet() $\leftarrow$ $H_{fa}$.getRule($r_a$) $\vee$
           $H_{fa}$.getRule($r_b$) $\vee$...
15:    **end if**
16: **end for**
```

To compute the header spaces, a simple exhaustive combination would require $O(u^v)$ time complexity for u rules and v ports on a path. We design a novel and more efficient algorithm that takes only $O(Nu \log u)$ time for N matching fields of the header space. Specifically, for each field d, the algorithm first sorts all start and end points of d for all rules. Next, the algorithm scans these points in ascending order and inserts a rule when encountering a start point and removes a rule when encountering an end point. After each insertion, the algorithm checks if the current value of d matches at least one rule of each port. For each field d, we need $O(u \log u)$ time to compute the rule sets and their corresponding range in the field. Therefore, for all N fields we need to compute their common rule sets. Hence the overall time is $O(Nu \log u)$.

The rules we have discussed so far are all forwarding rules. We are aware that a probe covering the forwarding rules in an entry can also test some ACL rules. Therefore, we use the hash table H_{fa} defined in Section III-A to retrieve a set of ACL rules that intersect the forwarding rules. We then add the ACL rules that can also be covered by the probe into the rule list of this entry (lines 13).

D. Minimizing the Number of Probes

For each entry in the reachability-to-rules table, one probe can cover all rules associated with that entry. As such, for a table with L entries, Pronto needs to send at most L probes to cover all rules in the networks. However, a rule may appear in multiple entries. In this case, if the rules in an entry have already been covered by the probes used to test other entries, there is no need to send a probe for this entry.

To minimize the number of probes for saving network bandwidth, Pronto selects the minimum set of entries while still being able to cover all rules that appear in the reachability-to-rules table. This problem can be formulated into a minimum set cover problem – one of Karp's 21 NP-complete problems [17]. However, the traditional greedy heuristic algorithm for minimum set cover [10] is inefficient. Specifically, in each iteration, the set that includes most items is selected and the items are removed from all other sets. The algorithm repeats until no item is left for any set. In the worst case, this algorithm takes $O(L^2)$ time for L sets (i.e., entries in our context). Our experiment using the real forwarding rules of Internet 2 [5] shows L can be as large as 210,000. Therefore, we need to design a more efficient algorithm.

Pronto uses a new minimum set cover algorithm whose idea is from a recent work [11]. For L entries we have L rule sets, R_1, R_2, ..., R_L. For each set R_i, we put it to Level t if $p^t \leq |R_i| \leq p^{t+1}$, where $p > 1$ is a pre-specified value. Let C be the covered rule set. For each round, we pick the set R_i from the highest level such as $|R_i \setminus C|$ is the largest among all sets in the level, and add its rules to C. For another set R_j in the highest level K, if $|R_j \setminus C| \leq p^K$, R_j will be dropped down to a lower level. The algorithm repeats until C includes all rules. The time complexity of the algorithm is $O([1 + 1/(p-1)]\Sigma_i|R_i|)$.

Pronto employs a different method to handle ACL rules. Since different ACL rules in an entry may be disjunct with each other, computing one header space to detect all possible ACL rules in a probe is infeasible. Thus, in pronto, one probe detects only one ACL rule. Among all entries at the same level t of the minimum set cover algorithm, entries that have ACL rules are tested first. Otherwise, these entries may be dropped down to a lower level without being tested. To do this, we randomly pick a uncovered ACL rule from each entry to generate a probe. The final header space of the probe is $\Omega = \Omega \wedge \psi_d$, where ψ_d is the header space of the selected ACL rule.

We have evaluated the efficiency of Pronto' algorithm. For the forwarding rules of Internet 2 [5], the traditional greedy algorithm [10] takes more than one hour to finish and generates more than 32,000 probes to cover all rules for 210,000 table entries. On the other hand, Pronto takes only one second to generate almost the same number of probes to cover all rules in the same table.

Here, the ACL *deny* rules are treated separately because a probe cannot test more than one *deny* rules. To address this problem, Pronto selects the minimum set of packets that cover at least one forwarding rule and one *deny* rule. Similar to the method described in Section III-A, Pronto first tests the forwarding rules, and then checks if the test packet disappears to determine the correctness of the *deny* rule.

IV. Probe Update

The second key component in Pronto is the probe update module. In many SDNs, rules undergo numerous updates to perform traffic engineering and congestion avoidance. Network engineers must ensure that the updates do not cause any unintended impact to the dynamic networks. To ensure consistency of the data plane after rule updates, Pronto updates the set of probes such that the probes can still cover all existing rules, including rules that have been removed, modified and newly added. We call this type of probe generation *probe update*. However, rather than starting the probe generation from scratch using the approach described in Section III, the key idea here is to identify rules that are changed or affected by the changed rules, and then update the existing probes that exercise these rules. To our best knowledge, we know of no existing approaches that use this idea to perform efficient probe generation under frequent rule changes.

We propose two strategies to support efficient probe update: quick update and optimal update. *Quick update* aims to compute a set of probes for covering all affected rules. However, quick update does not intend to optimize the overall number of probes. As such, if a network operator needs to periodically re-validate all existing rules, using quick update may lead to many probes. To address this problem, we propose *optimal update*, which is designed to minimize the set of probes to cover all rules after the update. In our evaluation, quick update takes less than 1ms to update probes for the two real network data planes, hence it can immediately react to frequent network updates. Optimal update takes a little longer, but only 0.5s and 0.05s. In practice, the network operator may choose to use quick update as the response to the rule changes in a short time (e.g., every 0.1 second), and periodically run optimal update to obtain a minimized set of probes (e.g., every 5 minutes).

A. Quick Update

The quick update method handles every changed rule. We consider two types of changes: rule removals and rule additions. In the case of rule modifications, we treat each modification as the removal of the old rule and addition of a new rule.

Rule removal. Pronto maintains a coverage history for a given rule to record the probes exercising that rule. For each removed rule r, we find the set of probes T that covers it by querying the reachability-to-rules table. If the rule list of a table entry covered by a $t \in T$ contains only a single r, then t is not used. If the rule list R has other rules R' besides r, we check whether each $r' \in R'$ can still be tested by the probe t' after r's removal. If not, we consider r' being *affected* by r's removal. Therefore, Pronto updates probe t' to ensure that r' is tested. In fact, the removal of r can affect not only the rules in the current table entry but also rules in other entries. Specifically, if the affected rule r' also appears in another entry e and the rule list R of e is not covered by any existing probes, Pronto generates a new probe for e. On the other hand, if R is covered by a probe t but $t' \notin T$, no action is needed. The above process is repeated until all affected rules by r's removal are tested.

Rule addition. When a new rule is added to the network, simply generating a new probe to test it is not enough. This is because the added rule may affect the header spaces of other rules with lower priority. For example, suppose a probe t is generated to test an existing rule r. If a new rule r' with higher priority is added, which also matches t, then t has no chances to exercise r. To address this problem, for every rule r sitting at the same switch with r', Pronto evaluates whether the intersection of the header spaces of r and r' is empty. If not, we perform actions to handle the following two cases: i) The probe t that is used to cover r matches r' rather than r. In this case, we further check if r is covered by another existing probe. If not, Pronto generates a new probe for r. ii) The probe t still matches r but does not match r'. If all rules that intersect r' fall into this case, Pronto generates a new probe for r'. The ingress and egress port of the probe should be on the same switch of this rule, and the header space of the probe should not intersect with other rules.

B. Optimal Update

The optimal update method also considers two types of changes – rule removals and rule additions. To obtain the minimum set of probes, optimal update needs to re-compute predicates and APs according to the rule changes. However, rather than re-computing them from scratch, Pronto focuses on only affected predicates and APs. For example, if an update occurs at the switch α, the predicates related to other switches do not need to be re-computed. To do this, Pronto first identifies the set of affected predicates that need to be re-computed based on the updated rules. The result is a set of removed predicates $\mathcal{P}_r$ and a set of new predicates $\mathcal{P}_n$. Pronto then uses the following method to re-compute the set of APs in the network. Recall that each AP a_i is in the form $a_i = q_1 \wedge q_2 \wedge ... \wedge q_k$, where $q_j \in \{p_j, \neg p_j\}$. We remove the term of p_j if $p_j \in \mathcal{P}_r$. After removing all terms of predicates in $\mathcal{P}_r$, we have a new set of conjunctions A'. Then the new set of APs is computed as $A' \sqcap A(\mathcal{P}_n)$.

After computing the new set of APs, Pronto compares them to the previous set of APs and identifies only APs that are different from the previous ones. This is because not every AP is affected after a number of rule updates. For every affected AP, Pronto re-computes the corresponding entries in the reachability-to-path table, and hence the reachability-to-rules table is also updated. Next, Pronto invokes the minimum set cover algorithm to obtain the new set of probes.

Changes of the ACL rules are handled differently. Here, we consider four types of ACL updates: permit expansion, permit shrink, deny expansion, and deny shrink.

1) Permit expansion means that the header space of an ACL *permit* rule is enlarged. In this case, Pronto generates a new Probe that matches the enlarged header space. The probe also matches some forwarding rules because ACL rules can only be tested along with forwarding rules.
2) Permit shrink means that the header space of an ACL *permit* rule becomes smaller. In this case, Pronto needs to ensure that the probe used to test this rule is still valid. Otherwise, this probe is updated by re-computing the intersection of the new ACL rule and corresponding forwarding rules. Detailed description is skipped.
3) For a deny expansion that may potentially block more packets, Pronto identifies the affected forwarding rules at the same switch and determines whether there exists any probe blocked by this expansion. New probes are generated to replace the blocked probes.
4) A deny shrink can be handled easily. If the previous probe used to test this shrink rule still matches the rule, we are done. Otherwise a new probe is generated to cover this rule.

V. Performance Evaluation

To evaluate Pronto we consider two research questions:

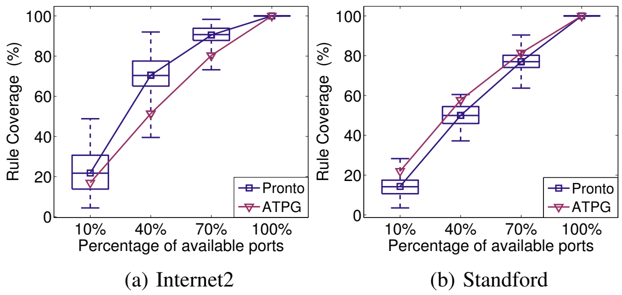
RQ1: How does the efficiency of Pronto, considering generating probe packets from scratch, compare to that of the ATPG approach?

RQ2: How efficient is Pronto at updating probes, and to what extent do the choices of using quick update and optimal update in Pronto affect its efficiency?

We have implemented a prototype of Pronto using Java. We conducted our experiments on a desktop computer with eight Intel i7-4790 processors running at 3.6GHz and 32GB RAM. To address our research questions, we applied Pronto to two real networks: Internet2 [5] and Stanford backbone network [4]. The Internet2 topology has nine routers with 126,017 forwarding rules. The Stanford topology includes 16 routers, 14 are edge routers while 2 are core routers. The data plane state of Stanford network contains 757,170 forwarding rules and 1,584 ACL rules (including about more than 100 VLANs). Some rules in two real networks are unreachable, we do not add them into the evluation.

In practice, it is possible that certain ports cannot be used to send probes for testing and debugging purpose. For example, private organizations may not be willing to use certain ports for sending probes due to security concerns. In traditional networks, it is also hard to send probes directly from a port inside a network [30]. In this paper, the ports that can be used to send probes are defined as *available ports*.

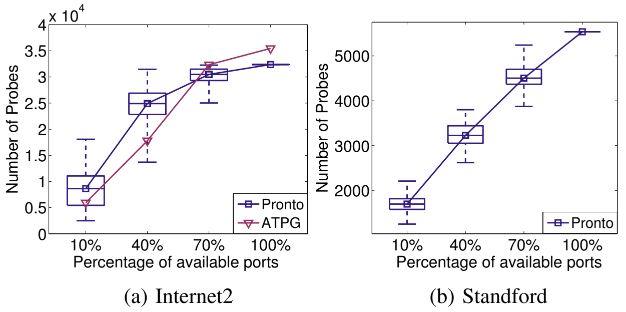
To answer our research questions, we compare Pronto to ATPG [30]. We choose ATPG as a baseline because it achieves



(a) Internet2

(b) Standford

Fig. 5: Rule coverage versus percentage of available ports



(a) Internet2

(b) Standford

Fig. 6: Number of probes vs. percentage of available ports. ATPG for Stanford is not shown because ATPG does not handle the VLAN rules

a similar objective as Pronto – generating minimum number of probes to cover multiple rules. We do not compare Pronto to Monocle because Monocle does not aim to optimize the number of probes. In fact, the number of probes used by Monocle is equivalent to the number of rules.

A. RQ1: Efficiency of Probe Generation from Scratch

To answer RQ1, we evaluate the efficiency of Pronto for probe generation when the percentage of available ports increases from 10% to 100%. We randomly choose a certain percentage of ports as available ports and calculate the subset of rules that can be tested. Fig. 5 shows the rule coverage for both Internet2 and Stanford network when the percentage of available ports is equal to 10%, 40% and 70%. For Pronto, we show the values of average, minimum, maximum, 25th, and 75th percentile, among all production runs. From the figure, we can see that Pronto has similar rule coverage rates on Internet2 and Stanford, and the average values are higher than those of ATPG in all cases.

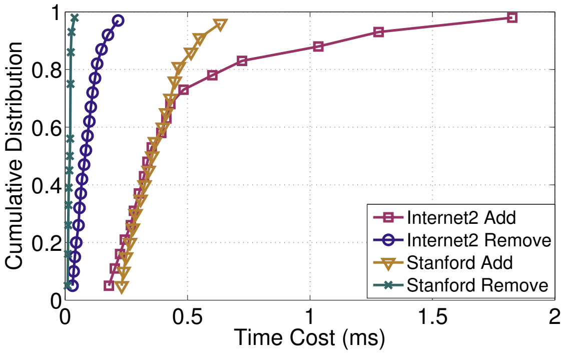


Fig. 7: CDF of the time cost for quick update.

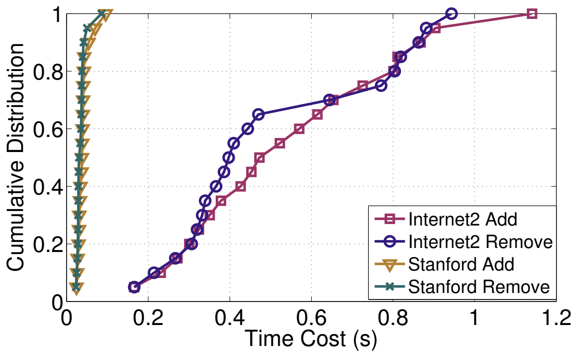


Fig. 8: CDF of the time cost for optimal update.

Fig. 6 shows the number of probes for both networks when varying the percentage of available ports. We compare the number of probes generated by Pronto and that by ATPG on Internet 2 in Fig. 5a. The results show that the two methods achieve almost equal effectiveness in terms of numbers. We do not perform this comparison on the Stanford, because ATPG does not handle VLAN rules on the Stanford, whereas Pronto does. Specifically, a VLAN rule takes multiple output actions as a signal rule, and Pronto generates probes for each action. The results show that, comparing to Internet 2, Stanford requires substantially fewer probes to cover all rules. Fig. 6 also indicates that the number of probes increases as more ports are used in Pronto. The gap penalty is smaller than the linear relationship. This observation implies that it is more likely to take advantage of minimum set cover when the size of rule set becomes larger.

The 2-3th rows in Table. IV and Table. V summarize the results on the number of generated probes and rule coverage across the percentage of available ports on both networks.

Next, we measure the cost of Pronto in terms of time. The results on the two networks are shown in the 4-5th rows of the Table. IV and Table. V, respectively. The fourth row of each Table reports the time for AP computation, i.e., 5.4s on Internet 2 and 1.57s on Standford. These times are consistent for each network regardless of the number of ports. The fifth row of each Table reports the total time for probe generation, including AP computation, construction of the reachability tables, and determining the probes. On the Internet 2, the probe generation time ranges from 5.8s to 6.9s. On the Stanford network, the total time is always less than 2s. These results indicate that the majority of time is spent on AP computation. On the other hand, the remaining steps are more efficient and cost less than 1.5s across all percentages of available ports. When comparing Pronto to ATPG [30] under the same experiment settings, ATPG takes 33.3 minutes and 46.8 minutes to generate probes for the Internet 2 and the Stanford, respectively, which are higher than those of Pronto by 300 to 1400 times.

In summary, the above results imply that: 1) *Pronto only*

TABLE IV: Probe generation results for Internet2.

Internet 2	10%	40%	70%	100%
Number of probes	9270	26749	30172	32379
Rule coverage	22.1%	65.2%	88.4%	100%
AP computation time	5.4s			
Probe generation	5.81s	6.23s	6.83s	6.9s

TABLE V: Probe generation results for Stanford.

Stanford	10%	40%	70%	100%
Number of probes	1649	2931	4828	5540
Rule coverage	12.1%	40.1%	80.5%	100%
AP computation time	1.57s			
Probe generation	1.64s	1.74s	1.87s	1.91s

takes a few seconds to generate probes and is thus efficient enough to be used in practice; 2) *Pronto is substantially more efficient than the existing method ATPG, while retaining the rule coverage effectiveness of this method.*

B. RQ2: Efficiency of Probe Update under Rule Updates

To answer RQ2, we evaluate the efficiency of Pronto on dealing with probe update. Here, we do not compare Probe to ATPG because ATPG does not generate probes to handle rule updates. We conduct 100 production runs for each of the rule addition and removal operations on the Internet 2 and Stanford. Fig. 7 shows the cumulative distribution of the time cost of Pronto when using the quick update method to react to rule updates, for the two networks. We find that in all cases of the rule addition operation on Internet 2 and rule addition and removal operations on Stanford, the time cost is always lower than 0.7ms. However, the time cost for the rule addition on Internet 2 has a (relatively) long tail: more than 10% rule additions requires greater than 0.1ms for probe update. In general, quick update is very fast.

Fig. 8 shows the cumulative distribution of the time cost of Pronto when using the optimal update method, which aims to minimize the number of probes. The results show that the optimal update for Stanford is very fast and all updates are completed within 0.1s. Pronto on Internet 2 requires more time for the optimal update, but in all cases, the updates are completed within 1.2s.

VI. RELATED WORK

There has been much work on testing and verifying network control planes [9], [18]–[20], [22], [28], [29]. For example, Header Space Analysis (HSA) [19] verifies essential properties for static networks. HSA relies on computing the intersection of packet header sets, which is highly computational-intensive. A later work [18] extends HSA to deal with dynamic network changes. AP Verifier [28] is a more time and space efficient tool to calculate the reachability of a network, compared to HSA. It proposes the concept of AP. We have described the details of AP calculation in Section II-B. These control plane verification tools cannot be used to debug the data plane.

There are several existing techniques for testing and verifying network data planes. For example, BUZZ [12] is a testing framework to ensure that the stateful data plane elements meet complex context-dependent policies. However, this technique focuses on data plane functions and cannot be used to test the rules considered in our work. SDN traceroute [6] measures packet data paths using tag-matching. It does not consider test packet generation to cover all rules. RuleScope [8] is designed to detect and diagnose data plane faults related to the priority of rules. This technique, however, is restricted to priority faults and cannot handle real-time rule updates.

The two closely related techniques to Pronto are ATPG [30] and Monocle [25]. They both can generate probes to test data plane rules. ATPG makes use of HSA and spends substantial amount of time ($>$ 1 hour) on generating probes. In addition, ATPG does not support real-time rule updates and thus needs to recompute all probes when network dynamics occur. Given the network update frequency, ATPG may not be able to generate complete probes to verify the current data plane state at a given moment. The SDN-based solution Monocle [25] restricts fault detection to a single switch. It uses a SAT solver to determine headers of probes. However, Monocle may incur large bandwidth overhead since each single rule in the data plane needs to be covered by an individual probe. In contrast, Pronto achieves the benefits and overcomes the limitations of the two techniques: it minimizes the number of probes and supports fast probe generation and update.

Fault localization for network data and control planes has also been studied. For example, NetSight [13] locates faults of the data plane by analyzing the packet processing histories on each switch. These techniques can be used as a complement for Pronto to locate and resolve faults when a test packet fails.

VII. CONCLUSION AND FUTURE WORK

We design and implement Pronto, a fast test packet generation tool for dynamic network data planes. Pronto uses the concept of atomic predicate to quickly compute the forwarding and ACL rules that can be tested by a probe sent from a port and received at another port in the network. Pronto employs an efficient minimum set cover algorithm to determine the set of probes that can cover all rules. Experiments using a single-thread implementation show that Pronto only takes several seconds to generate all probes from scratch and less than 1ms to update the probes for each changed rule, on two real network data plane rule sets that consist of hundreds of thousands rules.

Currently, Pronto can detect only forwarding rules and deny rules in the data plane. As part of the future work, we plan to develop approaches to handle modification rules. One option is to leverage the APT (Atomic Predicates for Transformers) method [14], which has been successfully used to handle the modification rules in the control plane.

ACKNOWLEDGEMENTS

This work was supported in part by NSF grants CNS-1701681 and CCF-1464032.

References

[1] A Survey on Network Troubleshooting. http://yuba.stanford.edu/~peyman/docs/atpg-survey.pdf.

[2] Comcast Outages Anger Thousands Across US. http://money.cnn.com/2016/02/15/news/companies/comcast-service-outage/.

[3] Dish Network Outage or Service down? Current Outages and Problems. http://downdetector.com/status/dish-network.

[4] Header Space Library and NetPlumber. http://bitbucket.org/peymank/hassel-public/.

[5] The Internet2 Observatory Data Collections. http://www.internet2.edu/observatory/archive/data-collections.html.

[6] K. Agarwal, E. Rozner, C. Dixon, and J. Carter. SDN traceroute: Tracing SDN Forwarding without Changing Network Behavior . In *Proc. of ACM HotSDN*, 2014.

[7] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *Computers, IEEE Transactions on*, 1986.

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

[9] M. Canini, D. Venzano, P. Pereni, D. Kosti, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *Proc. of NSDI*, 2012.

[10] V. Chvatal. A Greedy Heuristic for the Set-covering Problem. *INFORMS MOR*, 1979.

[11] G. Cormode, H. Karloff, and A. Wirth. Set Cover Algorithms for Very Large Datasets. In *Proc. of ACM CIKM*, 2010.

[12] S. K. Fayaz, Y. Tobioka, S. Chaki, and V. Sekar. BUZZ: Testing Context-Dependent Policies in Stateful Data Planes. In *Proc. of USENIX NSDI*, 2016.

[13] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *Proc. of USENIX NSDI*, 2014.

[14] Y. Hongkun and L. Simon. Scalable Verification of Networks with Packet Transformers Using Atomic Predicates. Technical report, Technical Report TR-15-09, The University of Texas at Austin, Computer Science Department, 2015.

[15] T. Inoue, T. Mano, K. Mizutani, S. Minato, and O. Akashi. Rethinking Packet Classification for Global Network View of Software-Defined Networking . In *Proc. of IEEE ICNP*, 2014.

[16] S. Jain et al. B4: Experience with a Globally-Deployed Software Defined WAN. *Proc. of ACM SIGCOMM*, 2013.

[17] R. M. Karp. Reducibility among Combinatorial Problems. In *Complexity of computer computations*. 1972.

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

[19] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking For Networks. In *Proc. of USENIX NSDI*, 2012.

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

[21] M. Kuźniar, P. Perešíni, and D. Kostić. What You Need to Know about Sdn Flow Tables. In *International Conference on Passive and Active Network Measurement*. Springer, 2015.

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

[23] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Proc. of ACM SIGCOMM*, 2011.

[24] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. In *Proc. of ACM SIGCOMM*, 2008.

[25] P. Peresini, M. Kuzniar, and D. Kostic. Monocle: Dynamic, Fine-grained Data Plane Monitoring. In *Proc. of ACM CoNEXT*, 2015.

[26] H. Wang, C. Qian, Y. Yu, H. Yang, and S. S. Lam. Practical Network-wide Packet Behavior Identification by AP Classifier. In *Proc. of ACM CoNEXT*, 2015.

[27] G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In *Proc. of IEEE INFOCOM*, 2005.

[28] H. Yang and S. S. Lam. Real-time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Trans. on Networking*, 2016.

[29] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Proc. Symposium on Security and Privacy*, pages 15–pp. IEEE, 2006.

[30] H. Zeng, P. Kazemiany, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *Proc. of ACM CoNEXT*, 2012.