Practical Network-wide Packet Behavior Identification by AP Classifier

Huazhe Wang, *Student Member, IEEE*, Chen Qian, *Member, IEEE*, Ye Yu, *Student Member, IEEE*, Hongkun Yang, *Student Member, IEEE*, and Simon S. Lam, *Fellow, IEEE, Fellow, ACM*

***Abstract*—Identifying the network-wide forwarding behaviors of a packet is essential for many network management applications, including rule verification, policy enforcement, attack detection, traffic engineering, and fault localization. Current tools that can perform packet behavior identification either incur large time and memory costs or do not support real-time updates. In this paper we present AP Classifier, a control plane tool for packet behavior identification. AP Classifier is developed based on the concept of atomic predicates which can be used to characterize the forwarding behaviors of packets. Experiments using the data plane network state of two real networks show that the processing speed of AP Classifier is faster than existing tools by at least an order of magnitude. Furthermore, AP Classifier uses very small memory and is able to support real-time updates.**

***Index Terms*—Network-wide behavior; Packet classification; Software-defined networking**

I. Introduction

MANAGING packet forwarding in a large network is a complex problem. Software defined networking (SDN) simplifies network management by decoupling the control plane from devices that forward packets, to be referred to as *boxes*.[1] More specifically, control plane applications, including routing [2], [3], traffic engineering [4], access control [5], measurement [6], and policy enforcement [7] [8], are implemented as software in a logically centralized controller. The controller specifies forwarding actions of packets by writing directly into flow tables in each box in the form of rules, through a standard API such as OpenFlow [9].

Let a *flow* be an equivalence class of packets defined on a subset of fields in the packet header, e.g., the 5-tuple consisting of source address, destination address, source port, destination port, and protocol type. All packets of a flow have the same forwarding behaviors in a network (also referred to as the flow's behaviors) when there is no data plane update. Network-wide *packet behavior identification* is a control plane function that discovers the actual forwarding behaviors of the packets in a flow (or a set of flows) including their forwarding paths, where they stop or are dropped, and which boxes they traverse, by analyzing network state in the data plane [10]. Packet behavior identification is necessary for SDN management in the following situations.

Verification of flow properties. For network flows, the control plane may specify pre-defined flow behaviors that satisfy application requirements or network policies, called *flow properties*. We highlight several typical flow properties.

- Forwarding correctness: The control plane must ensure that packets of the flow can be forwarded to the destination (e.g., a host or an egress router), or dropped if they are not allowed to reach the destination.
- Policy enforcement: Network policies may require flow packets to go through various middle boxes. For example, HTTP traffic should be forwarded through a sequence of middle boxes: firewall, IDS, and web proxy [7]. Other types of traffic may be required to traverse different sequences of middle boxes.
- Quality of service: Some applications require guaranteed flow quality. For example, a multi-tenant cloud should provide certain levels of bandwidth or latency for its users based on service level agreements [11], [12].
- VLAN isolation: A cloud provider guarantees that packets in a virtual network (VLAN) cannot travel to another VLAN.

Any data plane update could change the behaviors of a number of flows. Prior to data plane updates, the controller needs to verify that the data plane, with the new updates, can forward the packets correctly and comply with the flow properties. Such verification requires packet behavior identification for the flows that will be affected by the new rules.

Attack detection. Data plane attacks to an SDN may change the correct packet behaviors or send packets with abnormal behaviors, such as data plane DDoS attacks. An efficient data plane attack detection method should verify data plane forwarding behaviors and be aware of the behaviors that violate network policies. For example, a recent work SPHINX [13] uses flow graphs to represent network operations and detect abnormal forwarding behaviors.

Traffic engineering. Centralized traffic engineering [2] [14] [15] determines the forwarding paths for flows to maximize network throughput. When the controller is notified about a new flow, it needs to identify its packet behaviors in the current data plane and check whether they can meet application requirements such as bandwidth or latency. If not, the controller needs to modify the data plane to install a desired forward path for the flow.

Localization of network faults. When the control plane finds a flow property violation at any time, it should identify

Huazhe Wang, Chen Qian are with the Department of Computer Engineering, University of California Santa Cruz. {huazhe.wang, cqian12}@ucsc.edu. Chen Qian is the corresponding author. Ye Yu is with University of Kentucky. Hongkun Yang is with Google. Simon S. Lam is with The University of Texas at Austin. A preliminary version was published in *Proceedings of ACM CoNEXT* 2015 [1].

[1]We use "box" to refer to any network device that forwards packets, including routers, switches, and functional middle boxes such as firewalls, NATs and intrusion detection systems (IDSes).

the actual flow behaviors in the network and compare them with the expected behaviors. In this way the controller can find the part of the data plane that contains faults, called fault localization [16].

A practical packet behavior identification method must satisfy three requirements. First, it provides a high throughput in responding to packet behavior queries. According to recent measurement results [17] [18], a large data center network may see hundreds of thousands of new flows per second. SDNs should support hundreds of data plane updates per second [19] and each update may need to query multiple flows to verify correctness. Hence a desired throughput should exceed one million packet queries per second (1 Mqps). Second, the query structure should fit into a small and fast memory such as cache. Third, the query structure can be updated in real time under data plane changes to ensure that query results reflect the current network state.

Unfortunately none of the existing solutions can meet all of the requirements stated above. A straightforward approach is to maintain copies of flow tables of all boxes in the controller. However even for a medium-scale network used in [20], tens of GBs are required to store all rules [10]. Due to slow search speed among flow tables and disk I/Os, the query throughput is very low. Very recently, Inoue *et al.* [10] propose to use a multi-valued decision diagram (MDD) to classify flows to different sets of network-wide behaviors. However, an MDD cannot be updated in real time.[2]

In this work, we propose a network-wide packet behavior identification method called *AP Classifier*, where AP stands for Atomic Predicates, a concept developed in [22]. Each atomic predicate specifies a set of packets that have the same forwarding behavior in the network. The motivation of using atomic predicates is stated as follows. Existing solutions of packet behavior identification that uses forwarding table simulation or BDD-like structures are slow in processing queries and memory-inefficient because every bit of the packet header is considered. The concept of atomic predicates [22] provides a way to compress ACLs and forwarding rules to a small set of equivalence classes that can be specified efficiently. We hence develop a novel data structure, called AP Tree, to classify packets into atomic predicate which allows us to eliminate the primary cause of inefficiency using BDD-like structure to analyze packet flow behavior. The packet behavior can then be easily computed using the atomic predicate. To further increase the performance, AP Classifier employs optimized construction algorithms, so that the constructed AP Tree achieves higher query throughput. To deal with network dynamics, AP Classifier utilizes a real-time update to maintain query correctness and an AP Tree reconstruction method that periodically rebuilds the tree to optimize its performance.

We evaluated the performance of AP Classifier using the data plane network state, including forwarding tables and ACLs, from two real networks, namely: Internet2 [23] and a Stanford campus network [20]. Our results show that AP Classifier, running on a general-purpose desktop computer, only uses a few MBs memory and supports more than two millions of queries per second. In addition it can be updated in real time ($<$ 4 ms for 95% updates in Internet2 and $<$ 1 ms for 95% updates in Stanford).

The balance of this paper is organized as follows. Section II presents related work. We discuss the network model and background knowledge in Section III. We introduce the framework of AP Classifier in Section IV. The algorithms to construct an AP Tree are presented in Section V and the update and reconstruction methods of an AP Tree for dynamic networks are presented in Section VI. We present experimental results in Section VII. Finally we conclude this work in Section VIII.

II. Related Work

Network-wide packet behavior identification is equivalent to reachability computation for a specific packet. This problem is related to, but different from, network reachability analysis which has been studied for over a decade. Xie *et al.* [24] present a model for static reachability analysis of data plane network state. Quarnet [25] represents ACLs as firewall decision diagrams to compute network reachability. Header Space Analysis (HSA) [20] is custom-designed method to check network invariants but not in real time.

For real-time applications, NetPlumber [26] makes use of HSA to detect network invariant violations. Veriflow [27] stores all data plane rules in a multi-dimensional prefix tree (trie) and determines the Equivalence Classes (ECs) of packets. An EC is defined to be a set of packets that have identical forwarding actions in all boxes. Veriflow then checks network invariants by analyzing reachability graphs of ECs.

Binary Decision Diagram (BDD) [28] is an efficient structure that were used to model network properties. ConfigChecker [29] is general verification tool based on symbolic model checking. It uses a BDD to represent a set of state transitions (also flowchecker [30] by the same first author). If n header bits are used for filtering, each BDD of ConfigChecker uses $2n$ state variables which is less efficient than BDDs used in our design and [22] (In our design and AP Verifier, each BDD represents a set of packets and requires the use of n bit variables only). Anteater [31] uses boolean formulas to represent policies for packets traveling over edges in a network graph. McGeer[32] models network verification as Boolean satisfiability problems. They both use a SAT solver to check network properties. All of these general-purposes tools are slow and operate on time scales of seconds to hours [27].

All of the above methods focus on analyzing network-wide invariants (e.g., reachability, loop-freedom) but were not designed to identify the reachability of a specific packet. For example, they can determine whether it is possible to reach box B from box A but cannot tell whether a given packet can reach B. AP Verifier [22] can check whether all packets entering a port in the network pass through a waypoint (e.g., a firewall) but cannot tell whether a specific packet traverses a given waypoint.

[2]The paper [10] claims that if a data plane update does not change the existing packet behaviors, MDD update can be finished in tens of milliseconds. However from examining update traces of the Route Views Project [21], it is unlikely that a data plane update does not change the existing packet behaviors.

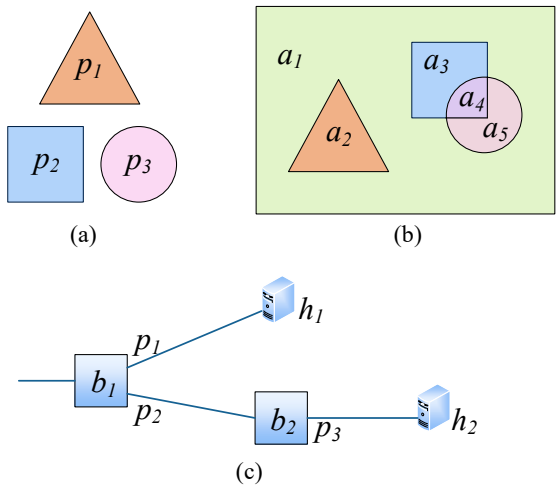


Fig. 1: (a) Three predicates. (b) The packet header space and five atomic predicates. (c) A sample network including the three predicates.

One possible solution to packet behavior identification problem is checking the packet against the set of atomic predicates calculated by AP Verifier linearly [22] which is impractically slow. Another solution is to obtain all related data plane rules of the packet by searching the trie created in Veriflow and then compute the forwarding path based on the rules. However storing all rules requires non-trivial memory cost (tens of GBs for the Stanford network) which could cause disk I/Os during query processing. As a result, using the Veriflow trie for packet behavior identification was shown to be very slow by Inoue *et al.* [10] who proposed a tool that can quickly classify a packet to an EC. Its main drawback is that their MDD structure cannot correctly represent the current network state because its does not support real-time updates, especially for SDNs where data plane updates are frequent [33]. Prefix DAG [34] employs a data structure similar to MDDs, but it focused on a simple classification problem with a single header field.

Recently, Network Optimized Datalog is proposed as a general specification language to model high-level abstraction of network beliefs and dynamism [35]. A new approach to derive data plane from network configurations is in [36].

III. Model and Background

We model a network as a directed graph of boxes, each of which has a forwarding table as well as input and output ports guarded by access control lists (ACLs). Each packet has a fixed-size header, including all fields that are evaluated by forwarding tables and ACLs in the network. A flow is then a sequence of packets that have the same values in the evaluated header fields.

Following the concepts in [22], forwarding tables and ACLs are all packet filters. Each ACL can be specified by a *predicate*. The set of packets that are allowed by the ACL are evaluated to true by the predicate. Similarly, by analyzing a forwarding table, each output port can be specified by a forwarding predicate. The set of packets that can be forwarded to the port are evaluated to true by the predicate.[3] Forwarding tables and ACLs can be converted to predicates using the algorithms in [22]. A predicate P specifies the set of packets for which P evaluates to true. Hence if a packet can travel through a sequence of packet filters, it is evaluated to true by the conjunction of predicates corresponding to the packet filters.

Given a set of predicates, we can compute a set of *atomic predicates*. Due to space limitation, we do not repeat the formal definition of atomic predicates, which can be found in [22]. A proved property of the set of atomic predicates is that they specify the minimum set of equivalence classes in the set of all packets. *The packets that are evaluated to true by the same atomic predicate have identical behaviors at all boxes.* For a set of predicates $P = \{p_1, p_2, ..., p_k\}$, each atomic predicate a_i is in the form $a_i = q_1 \wedge q_2 \wedge ... \wedge q_k$, where $q_j \in \{p_j, \neg p_j\}$. (Note that a_i in the previous sentence is an atomic predicate only if it is not false.) Every predicate is equal to the disjunction of a subset of atomic predicates. Every packet is evaluated to true by one and only one atomic predicate.

As an illustration, Fig. 1(a) shows three predicates p_1 (triangle), p_2 (square), and p_3 (circle), each of which represents a set of packets that are evaluated to true by a predicate. Each predicate specifies a set of packets that can pass the corresponding packet filter. Fig. 1(b) shows the three predicates in the packet header space. All packets in this example can be classified into five equivalence classes specified by five atomic predicates, a_1 to a_5. Each predicate is equal to the disjunction of a subset of atomic predicates. For example, $p_2 = a_3 \vee a_4$. Also, $a_4 = \neg p_1 \wedge p_2 \wedge p_3$. All packets evaluated to true by a_4 have identical behaviors: they can pass the filters of p_2 and p_3 but cannot pass p_1.

In the network shown in Fig. 1(c), Let p_1 specify the set of packets that can be forwarded at box b_1 to its output port to host h_1, p_2 specify the set of packets that can be forwarded at box b_1 to its output port to box b_2, and p_3 specify the set of packets that can be forwarded at box b_2 to its output port to host h_2. A packet specified by $a_4 = \neg p_1 \wedge p_2 \wedge p_3$ is forwarded at b_1 by the path $b_1 -> b_2 -> h_2$. A packet specified by $a_5 = \neg p_1 \wedge \neg p_2 \wedge p_3$ is forwarded to h_2 if it is at b_2, but will be dropped if it is at b_1. *An atomic predicate characterizes the behaviors of all packets it evaluates to true.*

IV. Design Framework of AP Classifier

AP Classifier is a program designed for a SDN controller. It computes the network-wide behaviors for an input packet (or flow). AP Classifier performs two-stage processing for a packet. First, using the AP Tree, it classifies the packet to the atomic predicate that evaluates to true for the packet. Second, AP Classifier determines all forwarding paths for the packet by using the atomic predicate, network information, and ingress box of the packet.

A. AP Tree

Using the algorithms presented in [22], the controller first converts each ACL to a predicate and the forwarding table of each box to m predicates, where m is the number of output ports of the box. Let $P = \{p_1, p_2, ..., p_k\}$ be the set

[3] All predicates are represented by binary decision diagrams (BDDs) [28] in our implementation of AP Classifier.

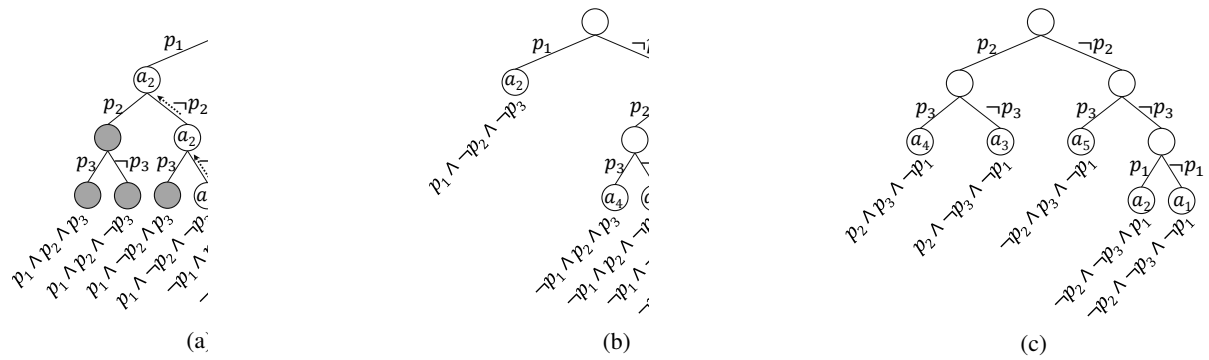


Fig. 2: AP Tree of predicates in Figure 1(b). (a) Original AP Tree. (b) Pruned AP Tree. (c) Optimized AP Tree.

of predicates of all boxes in the network. The controller constructs an *AP Tree* which is a binary tree. The root is labeled by p_1. At level i, the 2^i internal nodes are each labeled by p_i. Starting from the root, at each internal node, the input packet is evaluated by the predicate in the label. If the result is true, the packet continues to be evaluated in the left sub-tree. Otherwise it goes to the right sub-tree. An AP Tree with $(k+1)$ levels can be constructed from evaluating each of the k predicates at each level of internal nodes. A leaf node is then labeled by $q_1 \wedge q_2 \wedge ... \wedge q_k, q_i \in \{p_i, \neg p_i\}$, which specifies the set of packets reaching the leaf. Fig. 2(a) shows the AP Tree of the three predicates in Fig. 1(b). Shaded circles indicate leaf labels that are false. We will show that two sub-trees in an AP tree do not necessarily have a same predicate order in Section 5.3.

To classify a packet to an atomic predicate, AP Classifier simply searches the AP Tree by evaluating the packet until the leaf labeled by the atomic predicate is found. At each node, the packet is evaluated by checking the BDD of the predicate. Since predicates on sibling nodes are disjoint, for a given packet, the path from the root to the leaf is exclusive and determinate.

In the worst case, there could be 2^k atomic predicates and finding a leaf needs to evaluate all k predicates. However, it is found that the number of atomic predicates is surprisingly small for real networks [22]. Hence many leaves specify empty sets of packets. For example, in Fig. 2(a), $p_1 \wedge p_2 \wedge p_3$, $p_1 \wedge p_2 \wedge \neg p_3$, and $p_1 \wedge \neg p_2 \wedge p_3$ are all false according to the relationships in Fig. 1(b). Hence no packet can reach any of these three leaves. We use the following rule to "prune" the AP Tree: If no packet reaches a sub-tree, i.e., all leaves in the subtree are labeled by false predicates, the sub-tree is removed from the AP Tree. If an internal node has only one child, it is removed from the AP Tree as there is no need to check the predicate. We define the depth of a leaf to be the number of predicates evaluated to reach the leaf. After pruning, the average depth of all leaves in the AP Tree can be reduced and each node has either 0 or 2 children. Fig. 2(b) shows the pruned AP Tree has average depth $(1+3+3+3+3)/5 = 2.6$.

An important observation is the following: If predicates are placed at the levels in a different order, the average depth of the AP Tree may be different. In Fig. 2(c), the predicates are placed at three levels in the order of p_2, p_3, p_1. The average depth of all leaves in the pruned AP Tree is 2.4. *An important contribution of this work is an algorithm to find an order of predicates that substantially reduces the average depth of an AP Tree.*

For examples, each of the Internet2 and Stanford networks includes hundreds of thousands of forwarding rules, which can be converted to 161 (Internet2) or 507 (Stanford) predicates. Using our AP Tree construction algorithm, the average depth of the AP Tree is only 10.6 (Internet2) or 16.8 (Stanford). In an unpruned AP Tree, a packet needs to be evaluated by 161 or 507 predicates. AP Classifier only requires it to be evaluated by 10.6 or 16.8 predicates, on average, thus improving the query throughput by more than an order of magnitude. The detailed algorithm design of AP Tree construction is presented in Section V.

B. Computing packet behaviors

The second stage of AP Classifier determines the network-wide behaviors of the queried packet from the network information, the ingress box, and the atomic predicate determined in the first stage.

Since the atomic predicate is in the form $q_1 \wedge q_2 \wedge ... \wedge q_k, q_i \in \{p_i, \neg p_i\}$, for any predicate p_j, AP Classifier can easily check whether the predicate evaluates to true or false for the packet. Recall that p_j represents a packet filter of an ACL or output port. Hence AP Classifier can determine at any box whether the packet is dropped and which port it is forwarded to. Starting from the ingress box, i.e., the box that sees the packet first in the network, AP Classifier finds the output port to which the packet is forwarded and then determines the next-hop box. If the packet is a multicast packet, it may be forwarded to multiple ports. AP Classifier continues to find the forwarding ports on the next-visited boxes until the packet reaches the destination or is dropped. The packet behaviors are thus obtained.

Fig. 3 shows an example to illustrate how to compute network-wide forwarding paths for a given packet. Consider a packet which arrives at the ingress box b_1 and it is classified to atomic predicate a_4 by searching the AP Tree. The representation, $\neg p_1 \wedge p_2 \wedge p_3$, of a_4 shows that the packet is forwarded to b_2 because p_1 is false and p_2 is true for the packet. Similarly at b_2, the packet is forwarded to h_2 because p_3 is true for the packet.

We ran experiments to evaluate the speed of the above approach on a general-purpose desktop computer. We found

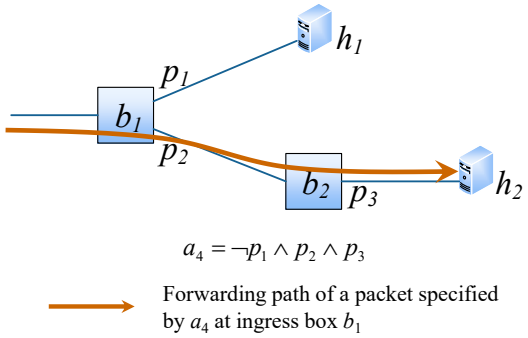
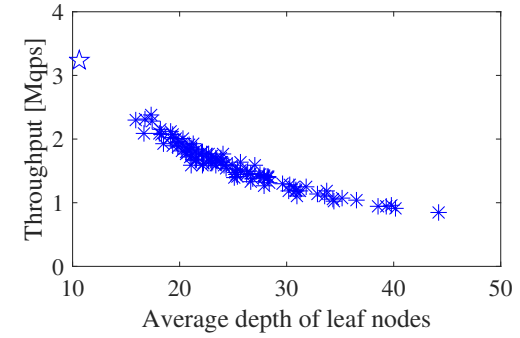
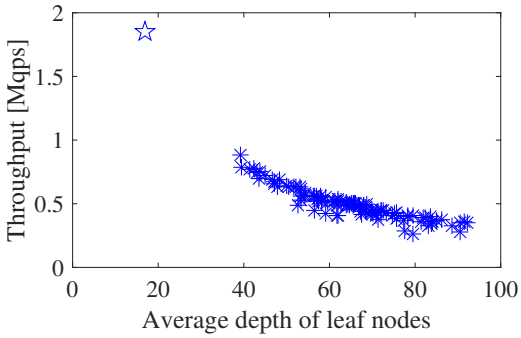


Fig. 3: Computing forwarding path for a packet in a_4



(a) Internet2



(b) Stanford

Fig. 4: Query throughput versus average depth of leaves

that, for the Internet2 and Stanford datasets, the throughput is greater than 15M and 10M packets per second, respectively. Note that this throughput is much higher than the throughput in the first stage. Therefore, the main effort of this work is to optimize the construction, search, and update of the AP Tree.

V. AP Tree Optimization

The most challenging problem in designing AP Classifier is to construct an AP Tree with minimized average depth, which can support dynamic updates.

A. Query throughput versus average depth

To reduce the query time and improve the query throughput, the optimization goal of AP Tree construction is to reduce the average depth of leaves. We conduct a set of experiments to justify the correlation of reducing the average depth and improving the throughput. We use the Internet2 network containing 161 predicates and the Stanford network containing 507 predicates. In each experiment, we randomly order the k predicates for placement at levels of the AP Tree. Then we query the generated tree using sample packets and measure the query throughput. In Fig. 4, we show the relationship between query throughput and average depth for 100 random generated AP Trees for each network. After pruning, the average depth of the AP Tree of Internet2 varies from 15.9 to 44.2, and the average depth of the AP Tree of Stanford varies from 39.1 to 92.5. From the two sub-figures in Fig. 4, it is obvious that an AP Tree with smaller average depth provides higher query throughput. The star in each figure represents the performance of the AP Tree constructed by AP Classifier. The query throughput of AP Classifier is 3.35 Mqps (Internet2) and 1.82 Mqps (Stanford), substantially higher than any random construction.

B. Quick-Ordering algorithm

The number of atomic predicates for a network is determinate if there is no update. That is, for a network, its AP Tree has a fixed number of leaves. A more balanced binary tree results in smaller average leaf depth. Compare the two AP Trees in Fig. 2(b) and (c) whose average depths are 2.6 and 2.4, respectively. The one in Fig. 2(c) is more balanced and hence has less average depth. The reason for the imbalance in Fig. 2(b) is that p_1 is placed at a higher level of the tree. According to properties of atomic predicates, every predicate is equal to the disjunction of a subset of atomic predicates. The number varies from one to the number of all atomic predicates. In this example, p_1 is a predicate that is equal to a single atomic predicate. Hence the left child of the node labeled as p_1 must be a leaf representing the atomic predicate. However, the right sub-tree may include more levels, causing the imbalance.

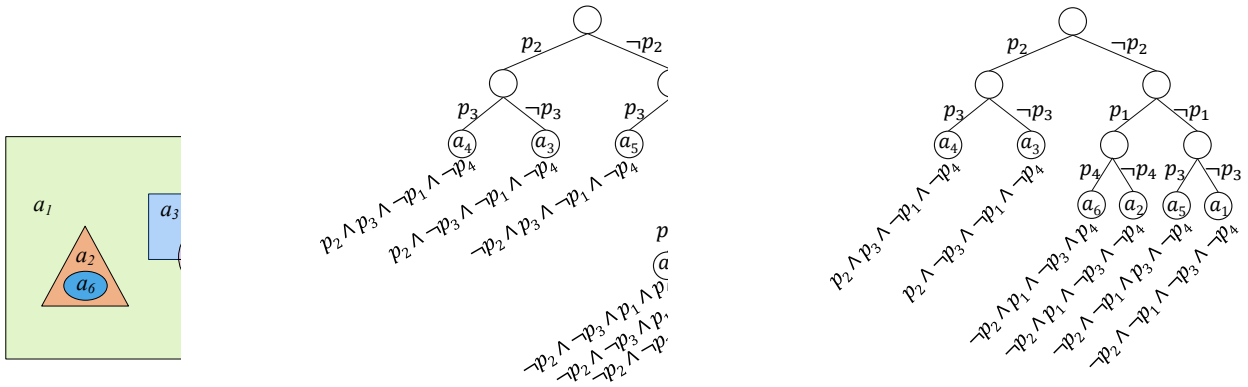
In fact, an analysis of the two real network data planes shows that many predicates are equal to a single atomic predicate. One fast yet effective ordering of predicates is to place those predicates at lower levels. For example, in Fig. 2(c), p_1 is placed at the lowest level.

Notation. Let $R(p)$ denote the subset of atomic predicates whose disjunction is p. $|R(p)|$ denotes the cardinality of $R(p)$.

In the *Quick-Ordering algorithm*, $|R(p_i)|$ is counted for each predicate p_i. Then the AP Tree is constructed by placing all predicates onto the tree in descending order of $|R(p_i)|$.

C. Optimized AP Tree construction

To develop a more sophisticated ordering method, one important observation is that, for two sub-trees whose roots are siblings, their predicate orders can be different. In the example of Fig. 5(a), we now have four predicates p_1 (triangle), p_2 (square), p_3 (circle), and p_4 (ellipse), which determine six atomic predicates, a_1 to a_6. If the predicates are added in the order p_2, p_3, p_1, p_4, the pruned AP Tree is shown in Fig. 5(b). However, for the sub-tree rooted at the right child of the root, its subtree is more balanced if the predicate order is p_1, p_3, p_4, as shown in Fig. 5(c).



(a) (b) (c)

Fig. 5: Additional example. (a) Five predicates. (b) Pruned AP Tree. (c) Optimized AP Tree.

For a given set of predicates $P = \{p_1, p_2, ..., p_k\}$, the atomic predicates $A = \{a_1, a_2, ..., a_n\}$ is determined. The number of leaves of the AP Tree is n, because each leaf corresponds to an atomic predicate. We define $F(Q,S)$ as the *minimal sum of leaf depths* of the subtree (which is a part of the AP Tree) whose nodes include the set of predicates Q and leaves are the set of atomic predicates S. In the example of Fig. 5(c), let $Q = \{p_1, p_3, p_4\}$ and $S = \{a_1, a_2, a_5, a_6\}$, $F(Q,S) = 8$. $F(Q,S)$ can be calculated recursively using the following equations. Let $H(Q,S,p)$ be the minimal sum of leaf depths if the root of the sub-tree is p. If $S \cap R(p) \neq \varnothing$ and $S \cap R(\neg p) \neq \varnothing$, $H(Q,S,p)$ is the sum of three components: $F(Q - \{p\}, S \cap R(p))$ and $F(Q - \{p\}, S \cap R(\neg p))$ are recursive computing for the left and right sub-trees and extra $|S|$ needs to be added because the depth of every leaf increments by 1. We have

$$H(Q,S,p) = F(Q - \{p\}, S \cap R(p)) + F(Q - \{p\}, S \cap R(\neg p)) + |S|$$

If $S \cap R(p) = \varnothing$, the left sub-tree will be pruned. The internal node with only one child is also removed and the leaf depths do not increase. Hence,

$$H(Q,S,p) = F(Q - \{p\}, S \cap R(\neg p))$$

Similarly, if $S \cap R(\neg p) = \varnothing$, we have,

$$H(Q,S,p) = F(Q - \{p\}, S \cap R(p))$$

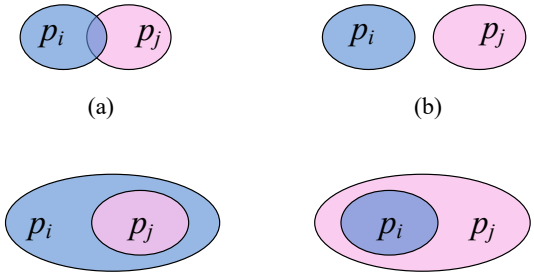
In addition, we have the following recursive equation.

$$F(Q,S) = \begin{cases} 0 & \text{if } |S| = 1 \\ \min_{p_i \in Q} H(Q,S,p_i) & \text{otherwise} \end{cases} \quad (1)$$

When $|S| = 1$, it is easy to see that the sub-tree contains only one leaf, hence $F(Q,S) = 0$. Otherwise, the predicate $p_i \in Q$ is selected as the root of the sub-tree such that p_i minimizes $H(Q,S,p_i)$.

Using the above formula, it is possible to compute $F(P,A)$. By recording the selection of p_i at each recursion, the optimized AP Tree can also be constructed.

However, the time complexity of solving this recursion is as high as $O((2^k) * k!)$, where k is the cardinality of P. We need to propose an efficient heuristic algorithm to simplify the recursion. At a level of recursion, we need to find the predicate p_i that minimizes $H(Q,S,p_i)$. Instead of trying all predicates, we propose an easier way to decide which predicate to select.



(a) (b) (c) (d)

Fig. 6: Relationships of two predicates. (a) Neither $P_i \wedge P_j$ nor $\neg P_i \wedge \neg P_j$ is false. (b) $P_i \wedge P_j$ is false. (c) $\neg P_i \wedge P_j$ is false. (d) $Pi \wedge \neg Pj$ is false.

We define a pair-wise relation between two predicates that implies which one is better to select. If $H(Q,S,p_i) < H(Q,S,p_j)$, we say that p_i is *superior* to p_j and p_j is *inferior* to p_i, denoted as $p_i \xrightarrow{S} p_j$. If $H(Q,S,p_i) = H(Q,S,p_j)$, we say p_i and p_j are in the same order, denoted as $p_i \overset{S}{\sim} p_j$.

We compare two predicates in four cases based on their logical relationships, as shown in Fig. 6. Here, p_i and p_j refer to predicates which are equal to union of atomic predicates in $S \cap R(p_i)$ and $S \cap R(p_j)$ respectively. $H(Q,S,p)$ is calculated based on the first three equations of section V-C for all four cases as follows:

1) Packets specified by p_i intersect with those of p_j (Fig. 6(a)). If we place p_i to the root and p_j to the children of the root, we get a full sub-tree since $R(p_i) \cap R(p_j)$, $R(p_i) \cap R(\neg p_j)$, $R(\neg p_i) \cap R(p_j)$ and $R(\neg p_i) \cap R(\neg p_j)$ are all non-empty. Hence, we have

$$\begin{aligned} H(Q,S,p_i) &= |S| + F(Q - \{p_i\}, S \cap R(p_i)) \\ &\quad + F(Q - \{p_i\}, S \cap R(\neg p_i)) \\ &= |S| + F(Q - \{p_i, p_j\}, S \cap R(p_i) \cap R(p_j)) \\ &\quad + F(Q - \{p_i, p_j\}, S \cap R(p_i) \cap R(\neg p_j)) \\ &\quad + |S \cap R(p_i)| \\ &\quad + F(Q - \{p_i, p_j\}, S \cap R(\neg p_i) \cap R(p_j)) \\ &\quad + F(Q - \{p_i, p_j\}, S \cap R(\neg p_i) \cap R(\neg p_j)) \\ &\quad + |S \cap R(\neg p_i)| \end{aligned}$$

If we place p_j to the root and p_i to the children, we can get $H(Q,S,p_j)$ similarly. Since $|S\cap R(p_i)|+|S\cap R(\neg p_i)| = |S\cap R(p_j)|+|S\cap R(\neg p_j)| = |S|$, $H(Q,S,p_i) = H(Q,S,p_j)$. We have $p_i \overset{S}{\sim} p_j$.

2) Packets specified by p_i disjoint with those of p_j (Fig. 6(b)). $p_i \wedge p_j$ is false. If we place p_i to the root and p_j to the children of the root, the sub-tree representing $R(p_i)\cap R(p_j)$ will be pruned. The child representing $R(p_i)\cap R(\neg p_j)$ will replace its parent node and leaf depths do not increase. However, the sub-tree representing $R(\neg p_i)\cap R(p_j)$ and $R(\neg p_i)\cap R(\neg p_j)$ are both non-empty, so the total leaf depths increase by $|S\cap R(\neg p_i)|$. Hence

$$\begin{aligned} H(Q,S,p_i) = |S| &+ F(Q-\{p_i,p_j\}, S\cap R(p_i)\cap R(\neg p_j)) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(\neg p_i)\cap R(p_j)) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(\neg p_i)\cap R(\neg p_j)) \\ &+ |S\cap R(\neg p_i)| \end{aligned}$$

Similarly, if we place p_j to the root and p_i to the children,

$$\begin{aligned} H(Q,S,p_j) = |S| &+ F(Q-\{p_i,p_j\}, S\cap R(p_j)\cap R(\neg p_i))) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(\neg p_j)\cap R(p_i)) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(\neg p_j)\cap R(\neg p_i)) \\ &+ |S\cap R(\neg p_j)| \end{aligned}$$

Despite of the same terms, if $|S\cap R(\neg p_i)| < |S\cap R(\neg p_j)|$, $p_i \overset{S}{\rightarrow} p_j$. If $|S\cap R(\neg p_i)| = |S\cap R(\neg p_j)|$, $p_i \overset{S}{\sim} p_j$. Otherwise $p_j \overset{S}{\rightarrow} p_i$.

3) Packets specified by p_j are a subset of those of p_i (Fig. 6(c)). $\neg p_i \wedge p_j$ is false. If we place p_i to the root and p_j to the children of the root, the sub-tree representing $R(\neg p_i)\cap R(p_j)$ will be pruned. The child representing $R(\neg p_i)\cap R(\neg p_j)$ will replace its parent node and leaf depths do not increase. The sub-tree representing $R(p_i)\cap R(p_j)$ and $R(p_i)\cap R(\neg p_j)$ are non-empty, so the total leaf depths increase by $|S\cap R(p_i)|$. Hence

$$\begin{aligned} H(Q,S,p_i) = |S| &+ F(Q-\{p_i,p_j\}, S\cap R(p_i)\cap R(p_j)) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(p_i)\cap R(\neg p_j)) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(\neg p_i)\cap R(\neg p_j)) \\ &+ |S\cap R(p_i)| \end{aligned}$$

If we place p_j to the root and p_i to the children of the root, the sub-tree representing $R(p_j)\cap R(\neg p_i)$ will be pruned.

$$\begin{aligned} H(Q,S,p_j) = |S| &+ F(Q-\{p_i,p_j\}, S\cap R(p_j)\cap R(p_i))) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(\neg p_j)\cap R(p_i)) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(\neg p_j)\cap R(\neg p_i)) \\ &+ |S\cap R(\neg p_j)| \end{aligned}$$

Therefore if $|S\cap R(p_i)| < |S\cap R(\neg p_j)|$, $p_i \overset{S}{\rightarrow} p_j$. If $|S\cap R(p_i)| = |S\cap R(\neg p_j)|$, $p_i \overset{S}{\sim} p_j$. Otherwise $p_j \overset{S}{\rightarrow} p_i$.

4) Packets specified by p_i are a subset of those of p_j (Fig. 6(d)). Similar to the above cases, we can get if $|S\cap R(\neg p_i)| < |S\cap R(p_j)|$, $p_i \overset{S}{\rightarrow} p_j$. If $|S\cap R(\neg p_i)| = |S\cap R(p_j)|$, $p_i \overset{S}{\sim} p_j$. Otherwise $p_j \overset{S}{\rightarrow} p_i$.

We then design the key criterion of predicate selection for each level of recursion, namely: *We select a predicate that is not inferior to any other predicate*. The ***algorithm*** is presented as follows: For each level of recursion, a predicate p_s is maintained, initially being p_1. A linear scan is performed from p_2 to p_k. For a predicate p_i, if $p_i \overset{S}{\rightarrow} p_s$, then p_s is set to p_i. At the end, p_s is selected as the root node of the subtree for this level of recursion.

To prove the correctness of the above algorithm, we need to show that p_s is indeed not inferior to any other predicate. A sufficient condition is that the superior/inferior relation is acyclic, i.e., there are no three predicates p_a, p_b, p_c such that $p_a \overset{S}{\rightarrow} p_b$, $p_b \overset{S}{\rightarrow} p_c$, and $p_c \overset{S}{\rightarrow} p_a$. We have proved the acyclic property by exhaustion. Our proof is not shown herein due to space limitation.

Time efficiency of AP Tree construction. In the AP Tree construction algorithm presented above, we avoid the time-intensive operation of computing the conjunction of two predicates represented as BDDs. Instead, our algorithm computes the intersection of two sets of integers that are identifiers of atomic predicates, as suggested in [22]. Intersections of integer sets can be computed much more quickly than conjunctions of BDDs. Each predicate is represented as a set of integers, so the time complexity of determining relationship between two predicates is $O(n)$, where n is the number of atomic predicates. For each level of recursion, a linear scan needs $O(k'n)$ time, where k' is the number of predicates in the current level. The overall complexity of building an AP Tree depends on the number of levels as well as the balance of the tree. Here we only provide the complexity analysis for a balanced AP Tree. For a balanced AP Tree, there are 2^l nodes at level l. For each node, $k' \leq (k-l)$. Hence at level l, the time complexity is at most $2^l(k-l)n$. Since $l \leq \log_2 n$, $2^l(k-l)n < kn^2$. Since there are $\lceil \log_2 n \rceil$ levels, the overall time complexity is upper-bounded by $O(kn^2 \log n)$.

D. Optimization for packet distribution

In the proposed algorithms, we assume that, for a packet query, leaf nodes (atomic predicates) have equal probability to be visited. Therefore minimizing the average depth of leaf nodes maximizes the query throughput. However, practical network flows may not be distributed uniformly with respect to the set of atomic predicates. For example, if many queried packets may eventually visit a leaf in a very deep position and leaves close to the root are rarely visited, the throughput decreases. To improve the query throughput for uneven packet distribution, we assign weights to atomic predicates such that leaf nodes that are visited frequently will be placed relatively close to the root.

To estimate the packet distribution, AP Classifier maintains a counter for each leaf node (atomic predicate), which records the number of visits by queries in a past period of time. The value of a counter is then converted to the weight of the corresponding atomic predicate after reduction of a fraction. When using the optimized algorithm presented in Section V-C, every occurrence of $|R(p_i)|$ is replaced by the sum of weights of all atomic predicates in $R(p_i)$, rather than its cardinality.

For example, suppose AP Classifier is choosing the root of a subtree by comparing two predicates p_i and p_j whose

relationship is as shown in Figure 6(c). If the atomic in set $R(p_j)$ have been queried by many packets, to place p_j before p_i in order to get smaller dep leaf nodes labeled by the atomic predicates in $R(p$ weights help to get $H(Q,S,p_j) < H(Q,S,p_i)$ and superior to p_i.

E. Dealing with packet header changes.

Today's networks rely on a wide range of m (e.g., firewalls, intrusion detection and preventio and proxies) which achieve performance and securi Some middleboxes may modify packet headers of traffic. When middleboxes modify packet header warding behaviors of these packets on downstream b be determined by the new header fields. For exam a Network Address Translation (NAT) middlebox an external address to an internal one, AP Classifi aware of such translation and compute the remain behaviors using the internal address.

We consider three types of packet header changes by middleboxes, namely 1) deterministic based on packet headers, 2) deterministic based on packet payload, and 3) probabilistic.

For Type 1 changes, a change is completely determined by the header of an incoming packet. In AP Classifier, we model these middlebox operations as a flow table. Each packet that enters a middlebox passes through a flow table. A flow table contains entries consisting of three components: match fields, instructions, and a new atomic predicate. Match Fields are used to select packets that match the predicates in the fields. Instructions specify new packet headers if a match occurs. The atomic predicate fields store atomic predicates calculated for new packet headers.

For Type 1 changes, given the packet header before a change, the atomic predicate after the change can be easily determined based on the flow table. Therefore when AP Classifier finds that a packet passes a middlebox, at the behavior computing stage (second stage of AP Classifier), it checks the flow table whether the packet header has been modified based on the middlebox policies. If the packet has a new header, AP Classifier will read a new atomic predicate and compute forwarding behaviors for the new header based on the new atomic predicate. Such process may repeat multiple times until the packet is dropped or the forwarding path ends at the packet's destination.

To see how this works, we use an extensional version of the example from IV-B in Fig. 7. The topology in the figure is a part of the whole network. Packets passing box b_1 are firstly processed by the flow table at middlebox MB_1 and then by b_1's forwarding table. The flow table of MB_1 contains three entries that modify packet headers and one default entry. Consider a packet enters box b_1 and matches the third entry of the flow table at MB_1. Its corresponding packet header fields are changed to 172.16.146.2 and its atomic predicate is changed to a_4. The yellow line, in Fig. 7, shows that the packet is forwarded to box b_2 and then host h_1 after header modification.

For Type 2 changes, the packet header after a change can be determined only after the packet payload is known. Hence

The flow table at MB_1

Match fields	Instructions	New atomic predicates
10.10.50.0/24	172.16.178.230	$a_2 = p_1 \wedge \neg p_2 \wedge \neg p_3$
10.10.60.0/24	172.16.158.49	$a_3 = \neg p_1 \wedge p_2 \wedge \neg p_3$
10.10.70.0/24	172.16.146.2	$a_4 = \neg p_1 \wedge p_2 \wedge p_3$
Others	None	Unchanged

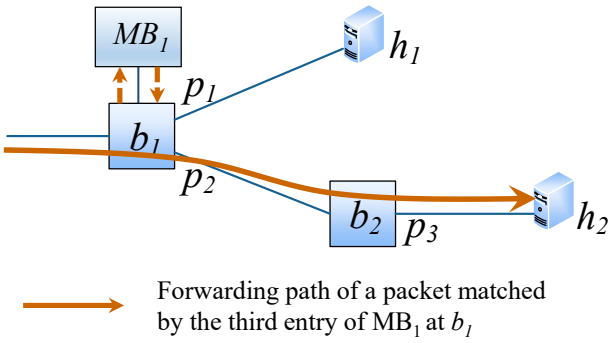


Fig. 7: Computing forwarding path with header modifications

it is not possible to pre-compute a flow table that stores the atomic predicate after packet header changes. AP Classifier needs to search the AP Tree again using the new header to find a new atomic predicate. This process may repeat multiple times. Probabilistic changes (Type 3) can be treated similarly. However, AP Classifier may output multiple possible network-wide behaviors for a given packet.

VI. AP Tree update and reconstruction

An important requirement of practical packet behavior identification is to support dynamic network changes, including link and rule changes, both of which require addition and deletion of predicates. We design fast AP Tree update methods for adding a predicate and deleting a predicate while maintaining tree correctness. However, after a large number of updates, an AP Tree will experience performance degradation. Hence we also design an AP Tree reconstruction method that periodically rebuilds the tree to optimize its performance while performing packet query processing at the same time. In this section, we assume that each atomic predicate is equally weighted.

A. Real-time update of an AP Tree

The SDN data plane of a network is frequently updated by rule installation and deletion. When a rule is inserted into or removed from a forwarding table or an ACL, it may change one or more predicates. The set of atomic predicates may change as well. We use the method presented in [37] to convert a rule insertion or deletion to predicate change. If there is no predicate change after a rule update, AP Classifier does not need to update the AP Tree. Otherwise, AP Classifier performs the methods presented below to remove the old predicate and add the updated predicate in the AP Tree. These methods are also used after addition/deletion of a network link which requires addition/deletion of predicates.

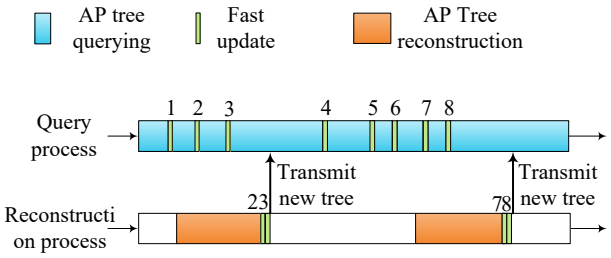


Fig. 8: Real-time update and query processing

Add a predicate. When a new predicate p is added, for each leaf node representing an atomic predicate a in the current AP Tree, AP Classifier computes $a \wedge p$ and $a \wedge \neg p$. If none of them is false, two children are added to the leaf node, representing $a \wedge p$ and $a \wedge \neg p$ respectively. If one and only one of the two conjunctions is false, the label of the leaf node is replaced by the other conjunction. If both conjunctions are false, AP Classifier does nothing to this leaf node.

Delete a predicate. To delete an existing predicate p from the AP Tree, AP Classifier does not remove all internal nodes labeled by p. This is because after the removal of a node, merging the two sub-trees rooted at its children is very difficult. Instead, we still keep p in the AP Tree, but mark it as "deleted" in the list of all predicates. A query packet is still processed by the AP Tree to find its leaf node representing its atomic predicate. It is still evaluated by the deleted predicates to determine which sub-tree to visit next. However, in the second stage of AP Classifier, i.e., computing packet behaviors, AP Classifier just ignores all predicates that have been deleted.

B. Parallel reconstruction of an AP Tree

Although, the AP Tree updates in AP Classifier are fast and maintain correctness of packet behavior identification, the AP Tree is no longer optimized and the query throughput will degrade over time. Hence AP Classifier also reconstructs the AP Tree to optimize it from time to time. To enable query processing at the same time as tree reconstruction, AP Classifier runs two processes in parallel, called the *query process* and *reconstruction process*, executing on two different cores. The start of a reconstruction is triggered by an event, e.g., query throughput is lower than a threshold or the number of updates on the current AP Tree is higher than a threshold. During reconstruction, the query process still maintains the old AP Tree by performing updates, and responds to queries. After the reconstruction process has built a new tree, the new tree needs to be updated for data plane changes that have occurred during the reconstruction period, if any. The updated new tree is then transmitted to the query process to replace the old tree.

Fig. 8 shows an example of the parallel reconstruction of an AP Tree. The query process performs AP Tree search to respond to queries as well as updates when data plane changes happen. In this example, the first reconstruction starts shortly after the change that requires update 1, which is included in the construction of a new tree. However, when the new tree is finished, two changes that require updates 2 and 3 have occurred during the reconstruction period. The new tree does not reflect these two updates. Thus the reconstruction process also applies these two updates to the new tree. Then the updated new tree is sent to the query process to replace the old AP Tree. Similarly the second reconstruction begins after changes that require updates 4, 5, and 6. The new tree constructed needs to be updated for changes (that require updates 7 and 8) which occur during the reconstruction period, before it can be sent to the query process. Note that if there is no data plane change during a reconstruction period, the new AP Tree is optimized.

If network dynamics change weights of atomic predicates, current AP Tree constructed using previous configurations should be rearranged to provide the best performance. It is hard to adjust AP Tree in the real time update process which should be finished very quickly. However, rearranging AP Tree needs to compare relationships of several predicates which may cost beyond the time scale of milliseconds. To regain the optimized performance of AP Tree, AP Classifier reconstructs AP Tree with the new weights of atomic predicates periodically.

TABLE I: Statistics of the two real networks

	Stanford		Internet2
No. of rules	Forwarding	ACL	Forwarding
	757170	1584	126017
No. of predicates	507	71	161
No. of atomic predicates	494	21	216

VII. Experimental Evaluation

We have implemented and evaluated AP Classifier on a general purpose desktop computer with quadcore@3.2G and 16GB memory. Our implementation and evaluation include all functional components for packet behavior identification from scratch, including computing atomic predicates, classifying packets using the AP Tree, and computing packet behaviors. (In comparison, prior work on this problem only implements and evaluates a single function, namely: classifying packets to equivalence classes [10].) For our experimental evaluation, we use forwarding tables and ACLs from two real networks: Internet2 [23] and Stanford network [20]. As shown in Table I, Internet2 includes 126,017 forwarding rules and the Stanford network includes 757,170 forwarding rules and 1,584 ACL rules. The predicates and atomic predicates are computed using the method in [22]. We compare AP Classifier with possible solutions by utilizing two state-of-art tools, namely Header Space Analysis (HSA) [20] and AP Verifier [22]. We do not compare AP Classifier with MDD [10] because it relies on a special method for MDD construction and the source code is not publicly available. Furthermore, its method does not support dynamic updates.

A. Depths of leaf nodes

In this set of experiments, we show the depths of leaf nodes in an AP Tree, which can demonstrate effectiveness of the proposed tree construction algorithms. We evaluate and compare three methods, Best from Random, Quick-Ordering,

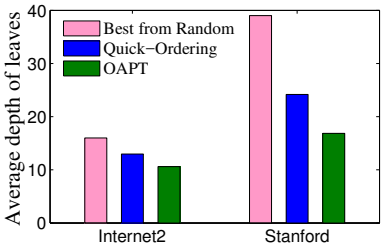
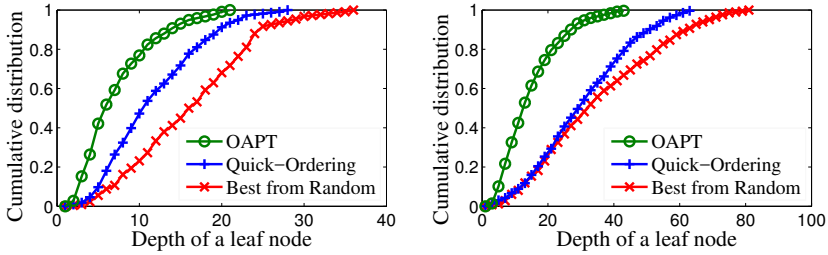


Fig. 9: Average depth of leaves



(a) Internet2 (b) Stanford

Fig. 10: Cumulative distribution of the depths of leaf nodes in AP Trees

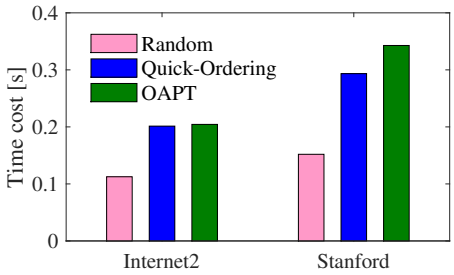


Fig. 11: Overall construction time cost of AP Classifier

and Optimized AP Tree construction (OAPT), for both Internet2 and Stanford networks. The Best from Random method generates a random order of predicates for placement on levels of an AP tree and performs pruning. It constructs 100 AP trees and chooses the tree with the minimal average depth of leaf nodes. Quick-Ordering is presented in Section V-B and OAPT is presented in Section V-C.

Fig. 9 shows the average depth of of leaf nodes in an AP tree. For Internet2, the average depth of Best from Random is 16.0, worse than those of Quick-Ordering (13.0) and OAPT (10.6). OAPT reduces the average depth by 34% compared to Best from Random and 19% compared to Quick-Ordering. For the Stanford network, Best from Random also has the highest average depth (39.0), followed by Quick-Ordering (24.2) and OAPT (16.9). OAPT shows significant improvement: It reduces the average depth by 57% compared to Best from Random and by 30% compared to Quick-Ordering.

Fig. 10 shows the cumulative distribution of depths of leaf nodes in an AP Tree. For Internet2, the leaf depths of Quick-Ordering are clearly smaller than Best from Random. However for the Stanford network such improvement is not very significant. OAPT has clearly smaller depths for all percentiles compared to the other two methods. For Internet2 80% of the leaf nodes in the OAPT tree have a depth less than 11 and for Stanford this number is 21. The maximum depths are 24 and 46 for Internet2 and Stanford, respectively.

B. Memory Usage

After construction, AP Classifier only stores one copy of all predicates and atomic predicates as BDDs and also, for each predicate, a set of integer identifiers of atomic predicates. In the AP Tree a node only stores a pointer to the labeled predicate or atomic predicate. Since pointers use very little memory, the memory costs of different methods are very close. Hence we only show the memory cost of AP Classifier using OAPT. In our implementation, we use JDD library [38] to construct BDDs and their logical operations. Each node in a BDD has a fixed size. The memory consumption of a BDD is determined by the number of nodes in the BDD. It is interesting to observe that more rules in a network do not always mean more BDD nodes. When there exist much more similarities among rules of a network, a BDD of the network is more likely to be simple with a smaller number of nodes. The memory cost for the network is prone to be lower.

The total memory cost of AP Classifier for Internet2 is 4.79 MB and that for Stanford is 2.15 MB. Although Internet2 has fewer predicates than Stanford, it requires more memory because BDDs of the Internet2 predicates are more complex than those of Stanford. Unlike the results of [10] that only show memory cost of the search structure, our memory costs account for all components for packet behavior identification, including the network topology, predicates, atomic predicates, and AP Tree. We found that AP Classifier uses very small memory and can be stored in cache.

C. AP Tree construction time

Fig. 11 shows times to construct AP Trees using the three methods for the two networks. Note that the time cost is the overall construction time that includes the times for computing atomic predicates as well as for AP Tree construction. The Random method costs the least time but it is only for one random construction. To find the best AP Tree from a large number of random constructions takes substantially longer time. Quick-Ordering and OAPT have similar time costs, 201.36 ms and 204.39 ms, for Internet2. For the Stanford network, OAPT requires 342.77 ms for Stanford, a little longer compared to Quick-Ordering (293.36 ms).

D. Query throughput for static networks

In this set of experiments, we measure the throughput of AP Classifier to process packet queries, in number of queries per second (qps). Packet headers used for queries in the experiments are generated randomly with respect to the atomic predicates. The throughput results for static networks are shown in Fig. 12. For Internet2, AP Classifier using OAPT can achieve 3.4 Mqps, higher than Best from Random by 102% and Quick-Ordering by 52%. For Stanford network, AP Classifier using OAPT can achieve 1.8 Mqps, higher than

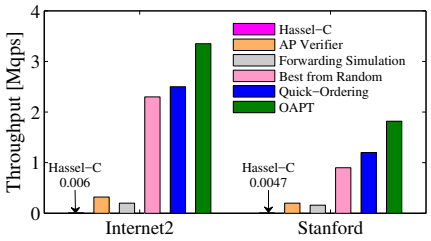
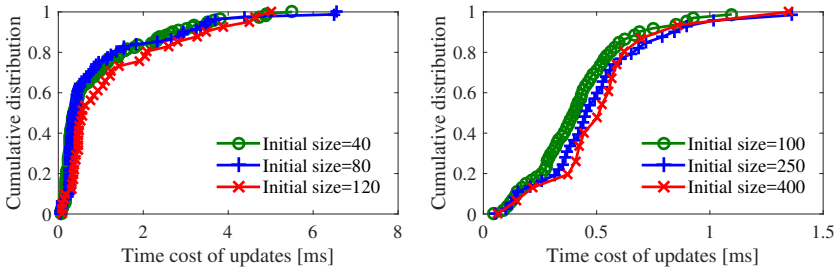


Fig. 12: Query throughput for static networks



(a) Internet2 (b) Stanford

Fig. 13: Cumulative distributions of time cost for adding a predicate.

Best from Random by 46% and Quick-Ordering by 34%. For both networks, the throughput of AP Classifier is much higher than 1 Mqps, which is enough to satisfy most application requirements in SDN.

For static networks, we can use the open-source tool Hassel-C [39] that implements HSA [20] to perform packet behavior identification for a specific packet. By providing the input port and a specific query packet, Hassel-C computes the reachability tree of the query packet. (For a unicast packet, the reachability tree is a forward path to the packet's destination.) The query throughputs of using Hassel-C to perform packet behavior identification are 6 Kqps and 4.7 Kqps for Internet2 and Stanford, respectively, which are about 1000 times slower than the query throughputs of AP Classifier. They are also plotted in Fig. 12 but they are very small and barely visible. We also compare AP Classifier with AP Verifier [22]. We first use AP Verifier to compute all atomic predicates, and perform a linear search of all atomic predicates for the query packet until the packet matches an atomic predicate. Results in Fig. 12 show that AP Verifier is also much slower, though its throughput is improved a lot compared to Hassel-C.

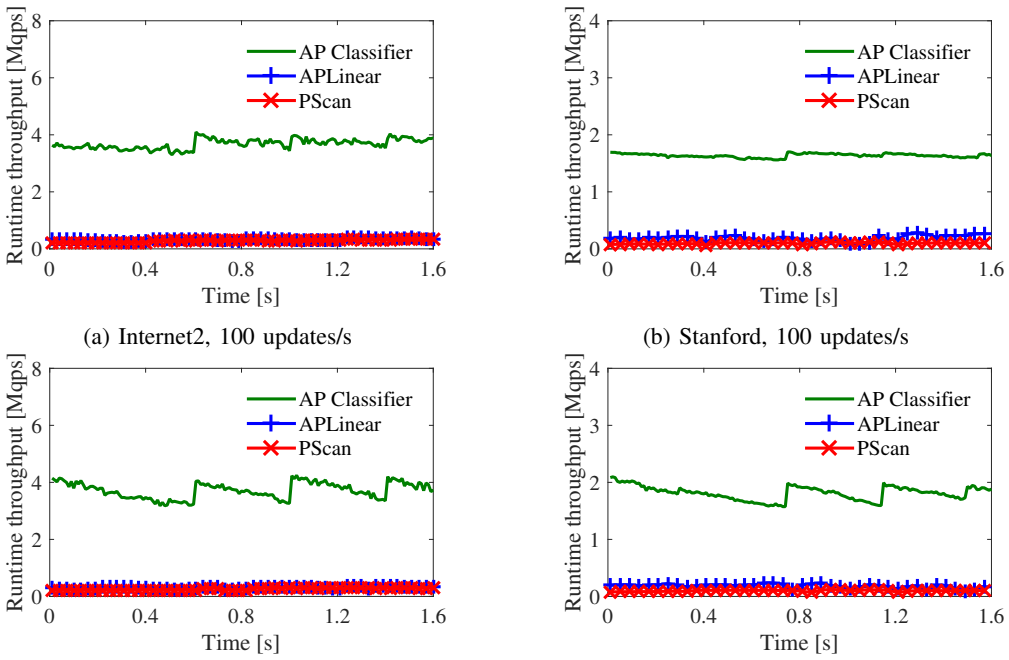
In addition we use a method of Forwarding Simulation, i.e., determining the forwarding behavior of the packet at a box, then checking the forwarding behavior on the next-hop box, until the packet stops. At each box, a packet is checked using the predicates at the box linearly until a match occurs. In our experiments using Forwarding Simulation, the average number of predicates checked is 96.8 and 232 for Internet2 and Stanford, respectively. The corresponding throughput is 0.2 Mqps and 0.16 Mqps as shown in Fig. 12. In contrast, only 10.6 and 16.8 predicates are needed to be checked on average using AP Classifier.

E. Dynamic Networks

In this set of experiments, we first construct the AP Tree using a number of predicates and then keep adding new predicates. We measure the time cost to add each new predicate and update the AP Tree. Fig. 13 (a) shows the cumulative distribution of time cost for adding a predicate in the Internet2 network. The initial number of predicates is set to 40, 80, and 120 for three different experiments. From the figure we find that about 80% of the predicate additions are finished in 2 ms. It may take 5-6 ms in worst cases. We do not observe obvious differences when the initial numbers of predicates are different. Fig. 13 (b) shows the results of similar experiments for Stanford. The initial number of predicates is set to 100, 250, and 400 for three different experiments. Over 90% of the predicate additions are finished in 1 ms. Deleting a predicate does not require extra computation, hence there is no result for deletions.

Query throughput for dynamic networks. We also evaluate the throughput of AP Classifier in practical environments where additions and deletions of rules and predicates happen over time. At the beginning of each experiment, a number of predicates are chosen randomly from the set of predicates of a network to construct the initial AP Tree. Starting from time 0, the arrivals of change events requiring the addition or deletion of predicates are modeled by a Poisson process. Each update operation can be adding a new predicate or deleting an existing predicate. In all experiments, equal numbers of additions and deletions are inserted to the event queue. A reconstruction is triggered every 0.4 s. During every reconstruction, AP Classifier answers queries and performs updates as explained in Section VI-B. We compare AP Classifier with two possible methods, APLinear and PScan, APLinear utilizes AP Verifier [22] to compute atomic predicates and performs a linear search for the query packet until the packet matches an atomic predicate. Note that BDDs of atomic predicates are more complex than those of predicates. Hence APLinear is not efficient. PScan performs a scan on all predicates using the query packet and decides whether the packet is filtered by the predicate. Both methods can be used to identify packet behaviors.

Fig. 14 shows the throughputs of AP Classifier, APLinear, and PScan in dynamic networks. The x-axis is time and the y-axis is throughput measured in Mqps. We conduct two sets of experiments whose update rates are 100 updates/s and 200 updates/s. From all subfigures in Fig. 14, we find that AP Classifier is faster than the other two methods by an order of magnitude. Note that starting from time 0, the throughput of AP Classifier slowly decreases as an increasing number of updates make the AP Tree less optimized. The first reconstruction starts at time 0.4 s and finishes at about 0.6 s in Fig. 14(a) and (c), and 0.7s in Fig. 14(b) and (d). When a reconstruction finishes, the throughput immediately goes back to a high value (4 Mqps in (a) and (c), and 2 Mqps in (b) and (d)). Furthermore, the throughput does not degrade in the long-term view. Comparing results of the two different update rates,



(a) Internet2, 100 updates/s

(b) Stanford, 100 updates/s

(c) Internet2, 200 updates/s

(d) Stanford, 200 updates/s

Fig. 14: Query throughput for dynamic networks. The number of updates per second is 100 in (a) (b) and 200 in (c) (d)

we find that the average throughput of AP Classifier does not drop much even after the update rate is doubled. Hence AP Classifier is fast and robust for practical dynamic networks.

F. Impact of packet distribution

To evaluate the performance of AP Classifier under various packet distributions, we generate new sets of test traces which are unevenly distributed with respect to the atomic predicates. The number of packets corresponding to the atomic predicates are chosen by sampling from a Pareto distribution. The probability density function for the Pareto distribution can be expressed as:

$$f_X(x) = \begin{cases} \frac{\alpha {x_m}^{\alpha}}{x^{\alpha+1}} & x \geq x_m \\ 0 & x < x_m \end{cases} \quad (2)$$

Where x_m is the minimum possible value of X, and α is a positive parameter, which is known as the tail index. In our experiments, we chose $x_m = 1$, $\alpha = 1$. About half of atomic predicates have 1,000 packets, but some have more than 20,000 packets.

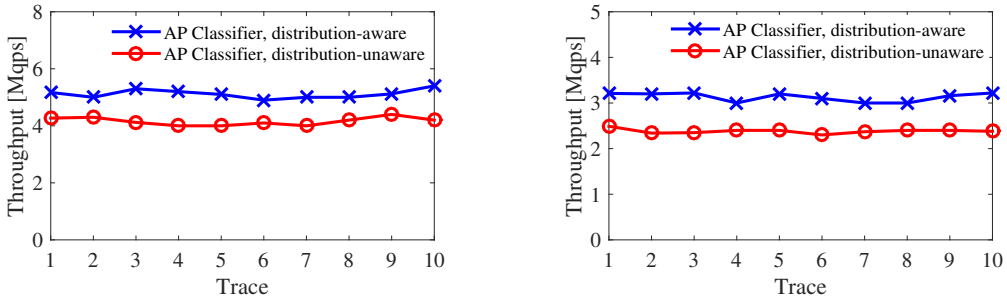
We generated 10 sets of traces for each network. If we still use the AP Trees constructed without the consideration of packet distributions (distribution-unaware), the average depth of all queries is 10.65 for Internet2 and 16.2 for Stanford network. Then we construct new distribution-aware AP Trees using the method described in Section V-D. The average depth of all queries is reduced to 8.09 (Internet2) and 11.3 (Stanford). The corresponding values of throughput are shown in Fig. 15. We can see that, if AP Classifier measures the packet distribution and assigns different weights to atomic predicates, the throughputs in all cases have notable improvements compared to the distribution-unaware method. The average query throughput increases from 4.2 Mqps to 5.2 Mqps for Internet2 and from 2.4 Mqps to 3.2 Mqps for Stanford.

G. Dealing with packet header changes

In this set of experiments, we evaluate the throughput of computing packet behaviors when there exist middleboxes modifying packet headers. We use the topologies of Internet2 and Stanford networks. In each experiment, one to three of switches are chosen as boxes connecting to middleboxes that may change packet headers. Due to lack of available middlebox policy data, we create ten entries for each flow tables of middleboxes. Match fields of flow tables are produced by dividing the packet header space into ten disjointed sets. We obtain match fields by grouping all atomic predicates into ten predicates. So every incoming packet can match an entry. When incoming packets match these entries, AP Classifier computes the remaining forwarding behaviors of packets using new atomic predicates. However for some packets, the new packet headers cannot be determined in advance. AP Classifier needs to search the AP Tree for the second time to find an atomic predicate for the new header. The process of computing packet behaviors ends until the packet is dropped or reaches the destination.

We measure the throughput of packet behavior computation under these circumstances. Packets used in the experiments are generated randomly with respect to atomic predicates.

Table. II illustrates throughput of computing packet behaviors for Internet2 and Stanford datasets in different scenarios. We define the deterministic ratio as the portion of middlebox rules that can determine the atomic predicates of packets after packet header changes. When the deterministic ratio is 0.9, the throughput does not downgrade much as number of middleboxes increases since most packets have new atomic



(a) Internet2 (b) Stanford

Fig. 15: Query throughput of AP Classifier for different packet distributions

TABLE II: Throughput with packet header changes

	Throughput(Mpps)		
No. of middleboxes	One	Two	Three
Internet2	13	10.2	9.8
Stanford	10	8.6	7.4

(a) Deterministic ratio = 0.9.

	Throughput(Mpps)		
No. of middleboxes	One	Two	Three
Internet2	11.2	9.8	8.5
Stanford	8.9	7.9	7

(b) Deterministic ratio = 0.5.

	Throughput(Mpps)		
No. of middleboxes	One	Two	Three
Internet2	8.7	6.9	3.2
Stanford	7.1	4.9	2.1

(c) Deterministic ratio = 0.

predicates stored in the flow tables, as shown in Table. II (a). Compared with Table. II (a), the corresponding throughput values in Table. II (b) and (c) are lower since more packets passing through a middlebox require searching the AP Tree for a second time. In the worst case, the throughput of computing packet behaviors is still 3.2 M and 2.1 M packets per second respectively, which is much higher than using other methods.

VIII. Conclusion

We propose AP Classifier for network-wide packet behavior identification that can be utilized by many important network management applications. We design algorithms to construct the AP Tree for a network, which can be used to quickly classify a packet to an atomic predicate. Each atomic predicate represents the network-wide forwarding behaviors of a set of packets. Experimental results using the datasets of two real networks show that the proposed AP Tree construction algorithm can optimize the average depth of leaf nodes. AP Classifier can process millions of packet queries per second. The speed is faster than existing tools by at least an order of magnitude. Furthermore, it uses only a few MBs memory. It can be updated in real time and is robust under dynamic data plane changes.

Acknowledgement

Huazhe Wang, Chen Qian, and Ye Yu were supported by NSF grants CNS-1701681 and CNS-1717948. Hongkun Yang and Simon S. Lam were supported by NSF grant CNS-1214239. We thank the anonymous reviewers for their comments.

References

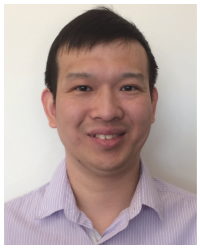
[1] H. Wang, C. Qian, Y. Yu, H. Yang, and S. S. Lam, "Practical Network-wide Packet Behavior Identification by AP Classifier," in *Proc. of ACM CoNEXT*, 2015.
[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *Proc. of USENIX NSDI*, 2010.
[3] Q. Chen, C. Qian, and S. Zhong, "Privacy-preserving cross-domain routing optimization-a cryptographic approach," in *Proc. of IEEE ICNP*, 2015.
[4] S. Agarwal, M. Kodialam, and T. Lakshman, "Traffic engineering in software defined networks," in *Proc. of IEEE INFOCOM*, 2013.
[5] A. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic access control for enterprise networks," in *Proc. of ACM WREN*, 2009.
[6] Y. Yu, C. Qian, and X. Li, "Distributed collaborative monitoring in software defined networks," in *Proc. of ACM HotSDN*, 2014.
[7] Z. A. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using SDN," in *Proc. of ACM SIGCOMM*, 2013.
[8] X. Li and C. Qian, "An NFV Orchestration Framework for Interference-free Policy Enforcement," in *Proceedings of IEEE ICDCS*, 2016.
[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.
[10] 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.
[11] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks." in *Proc. of ACM SIGCOMM*, 2011.
[12] X. Li and C. Qian, "Traffic and failure aware vm placement for multi-tenant cloud computing," in *Proc. of IEEE IWQoS*, 2015.
[13] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks." in *NDSS*, 2015.
[14] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proc. of ACM CoNEXT*, 2011.
[15] H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic engineering with forward fault correction," in *Proc. of ACM SIGCOMM*, 2014.
[16] H. Zeng, P. Kazemiany, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proc. of ACM CoNEXT*, 2012.
[17] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proc. of ACM IMC*, 2009.
[18] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. of ACM IMC*, 2010.
[19] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," *Proc. of ACM SIGCOMM*, 2013.
[20] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. of USENIX NSDI*, 2012.
[21] "University of oregon route views project, accessed on Jun. 2015" http://www.routeviews.org.

[22] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," in *Proc. of IEEE ICNP*, 2013, extended version in *IEEE/ACM Transactions on Networking*.
[23] "The internet2 observatory data collections, accessed on Oct. 2013." http://www.internet2.edu/observatory/archive/data-collections.html.
[24] 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.
[25] A. R. Khakpour and A. X. Liu, "Quantifying and querying network reachability," in *Proc. of IEEE ICDCS*, 2010.
[26] 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.
[27] 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.
[28] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.
[29] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi, "Network configuration in a box: Towards end-to-end verification of network reachability and security," in *Proc. of IEEE ICNP*, 2009.
[30] E. Al-Shaer and S. Al-Haj, "Flowchecker: Configuration analysis and verification of federated openflow infrastructures," in *Proc. of ACM SafeConfig*, 2010.
[31] 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.
[32] R. McGeer, "Verification of switching network properties using satisfiability," in *Proc. of IEEE ICC*, 2012.
[33] M. Kuzniar, P. Peresini, and D. Kostic, "What you need to know about SDN flow tables," in *Proc. of PAM*, 2015.
[34] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, "Compressing ip forwarding tables: towards entropy bounds and beyond," in *Proc. of ACM SIGCOMM*, 2013, extended version in *IEEE/ACM Transactions on Networking*.
[35] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proc. of USENIX NSDI*, 2015.
[36] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *Proc. of USENIX NSDI*, 2015.
[37] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," The Univ. of Texas at Austin, Dept. of Computer Science, Tech. Rep. TR-13-15, Aug. 2013.
[38] A. Vahidi, "Jdd, a pure java bdd and z-bdd library," http://javaddlib.sourceforge.net/jdd/index.html, 2004.
[39] "Hassel-C," http://bitbucket.org/peymank/hassel-public/.

Huazhe Wang (M'15) is a third year Ph.D. student at Department of Computer Engineering, University of California Santa Cruz. He received the B.Sc degree from Bejing Jiaotong University in 2011, the M.Sc degree from Beijing University of Posts and Telecommunications in 2014. His research interests includes software defined networking and network security. He is a student member of IEEE and ACM.

Chen Qian (M'08) is an Assistant Professor at the Department of Computer Engineering, University of California Santa Cruz. He received the B.Sc. degree from Nanjing University in 2006, the M.Phil. degree from the Hong Kong University of Science and Technology in 2008, and the Ph.D. degree from the University of Texas at Austin in 2013, all in Computer Science. His research interests include computer networking, network security, and Internet of Things. He has published more than 50 research papers in highly competitive conferences and journals. He is a member of IEEE and ACM.

Ye Yu (M'13) is a Ph.D. student at the Department of Computer Science, University of Kentucky. He received the B.Sc. degree from Beihang University. His research interests including data center networks and software defined networking.



Hongkun Yang (M'12) received the Ph.D. degree in the Department of Computer Science, University of Texas at Austin in 2015, where he is a recipient of the MCD Fellowship. He received the B.S.E. degree with Distinction and the M.S.E. degree from Tsinghua University in 2007 and 2010, respectively. His research interests include computer networks, protocol verification, network security, and formal methods. He has published research papers in a number of conferences and journals including IEEE ICNP, IEEE INFOCOM, IEEE Transactions on Mobile Computing. He is a student member of IEEE.

Simon S. Lam (F'85) received the B.S.E.E. degree with Distinction from Washington State University, Pullman, in 1969, and the M.S. and Ph.D. degrees in engineering from the University of California, Los Angeles, in 1970 and 1974, respectively. From 1971 to 1974, he was a Postgraduate Research Engineer with the ARPA Network Measurement Center at UCLA, where he worked on satellite and radio packet switching networks. From 1974 to 1977, he was a Research Staff Member with the IBM T. J. Watson Research Center, Yorktown Heights, NY. Since 1977, he has been on the faculty of the University of Texas at Austin, where he is Professor and Regents Chair in computer science, and served as Department Chair from 1992 to 1994.

He served as Editor-in-Chief of *IEEE/ACM Transactions on Networking* from 1995 to 1999. He served on the editorial boards of *IEEE/ACM Transactions on Networking*, *IEEE Transactions on Software Engineering*, *IEEE Transactions on Communications*, *Proceedings of the IEEE*, *Computer Networks*, and *Performance Evaluation*. He co-founded the ACM SIGCOMM conference in 1983 and the IEEE International Conference on Network Protocols in 1993.

Professor Lam is a Member of the National Academy of Engineering and a Fellow of ACM. He received the 2004 ACM SIGCOMM Award for lifetime contribution to the field of communication networks, the 2004 ACM Software System Award for inventing secure sockets and prototyping the first secure sockets layer (named Secure Network Programming), the 2004 W. Wallace McDowell Award from the IEEE Computer Society, as well as the 1975 Leonard G. Abraham Prize and the 2001 William R. Bennett Prize from the IEEE Communications Society.