Real-time Verification of Network Properties using Atomic Predicates

Hongkun Yang and Simon S. Lam
Department of Computer Science, The University of Texas at Austin
{yanghk, lam}@cs.utexas.edu

TR-13-15 August, 2013

***Abstract*—Network management will benefit from automated tools based upon formal methods. Several such tools have been published in the literature. We present a new formal method for a new tool, Atomic Predicates (AP) Verifier, which is much more time and space efficient than existing tools. Given a set of predicates representing packet filters, AP Verifier computes a set of atomic predicates, which is minimum and unique. The use of atomic predicates dramatically speeds up computation of network reachability. We evaluated the performance of AP Verifier using forwarding tables and ACLs from three large real networks. The atomic predicate sets of these networks were computed very quickly and their sizes are surprisingly small.**

Real networks are subject to dynamic state changes over time as a result of rule insertion and deletion by protocols and operators, failure and recovery of links and boxes, etc. In a software-defined network, the network state can be observed in real time and thus may be controlled in real time. AP Verifier includes algorithms to process such events and check compliance with network policies and properties in real time. We compare time and space costs of AP Verifier with NetPlumber using datasets from the real networks.

I. Introduction

Managing a large packet network is a complex task. The process of forwarding packets is prone to faults from configuration errors and unexpected protocol interactions. In large packet networks, forwarding tables in routers/switches are updated by multiple protocols. Access control lists (ACLs) in routers, switches, and firewalls are designed and configured by different people over a long period of time. Links may be physical or virtual (e.g., VLAN, MPLS). Some middle boxes also modify packets (e.g., NAT). In a study of large-scale Internet services [15], operator error was found to be the largest single cause of failures with configuration errors being the largest category of operator errors.

Towards more reliable networks, formal analysis methods and automated tools have been proposed to check reachability (e.g., "a packet with certain header values cannot reach host *y*") and to verify essential network properties (e.g., "the network has no routing loop for all packets"). A model for static reachability analysis of network state in the data plane was first presented by Xie et al. [19]. They proposed a unified approach for reasoning about the effects of forwarding and filtering rules as well as packet transformations on reachability. This approach motivated subsequent development of algorithms and automated tools by other researchers [3], [12], [14], [11], [13], [10]. In these tools, the algorithm for computing reachability is the core algorithm for verifying essential network properties in the data plane, such as, loop-freedom, nonexistence of black holes, network slice isolation, reachability via waypoints, etc.

The network state in the data plane is determined by the forwarding and ACL rules in the network's middle boxes. Forwarding tables and ACLs are packet filters. They can be parsed and represented by predicates that guard input and output ports of middle boxes. The variables of such a port predicate are packet header fields.[1] Packets with identical values in their header fields are considered to be the same by packet filters. A predicate P specifies the set of packets for which P evaluates to true. The set of packets that can travel from port s to port d through a sequence of packet filters can be obtained by computing the *conjunction of predicates* in the sequence or by intersection of the corresponding packet sets.

The intersection and union of packet sets are highly computation-intensive because they operate on multi-dimensional sets which could have many allowed intervals in each dimension and arbitrary overlaps in each dimension between two packet sets. In the worst case, the computation time of set intersection/union is $O(2^n)$ where n is the number of bits in the packet header. Efficiency of these operations determines the efficiency of reachability analysis irrespective of which formal method is used to compute reachability.

In this paper, we propose a novel idea that enables very fast computation of reachability. For a given set of predicates, we present an algorithm to compute a set of atomic predicates, which is proved to be minimum and unique. Atomic predicates have the following property: Each given predicate is equal to the disjunction of a subset of atomic predicates and can be stored and represented as a set of integers that identify the atomic predicates. *The conjunction (disjunction) of two predicates can be computed as the intersection (union) of two sets of integers.* Thus, intersection and union of packet sets can be computed very quickly. Based upon this idea, we developed a formal analysis method and prototyped an automated tool, named *Atomic Predicates (AP) Verifier*, for computing reachability and checking compliance with network policies and properties in real time.

We evaluated the performance of AP Verifier using forwarding tables and ACLs from *three real networks* downloaded from Stanford University [1], Purdue University [16], and Internet2 [2]. Since forwarding rules and ACL rules have different characteristics and locality properties, AP Verifier computes two different sets of atomic predicates, one for ACL predicates and another for forwarding predicates. We found that the atomic predicate sets of the three networks can be

[*]An abbreviated version of this technical report to appear in *Proceedings of IEEE ICNP*, October 2013.

[1]We will shorten "port predicate" to "predicate" whenever the meaning is clear from context.

computed very quickly and their sizes are surprisingly small. For example, the Stanford network [11] has 71 ACLs with 1,584 rules but we found only 21 atomic predicates for these ACLs and rules. This outcome is due to the existence of *large amounts of redundancy in the forwarding and ACL rules of real networks*. *By encoding the network state in terms of atomic predicates, such redundancy is eliminated*. Therefore, AP Verifier is much more time and space efficient than other automated tools for network verification published to date.

Real networks are subject to dynamic state changes over time as a result of, for examples, rule insertion and deletion by protocols and operators, failure and recovery of links and boxes, etc. Recently, two research groups suggested that in a software-defined network (SDN), the network state can be observed in real time and thus may also be controlled in *real time* [13], [10]. More specifically, if a "verifier" is placed in the communication path between a SDN's central controller and its middle boxes, the verifier can intercept every network state change message and verify compliance of the state change with pre-defined network policies and properties. If a state change is detected in real time to be noncompliant, the verifier may raise an alarm or block the state change. We have designed algorithms for AP Verifier to perform such real-time checks. AP Verifier was found to be especially fast in checking reachability compliance of a link up/down event. Existing tools used several seconds of time to verify compliance of a link up/down event [10], [13]. AP Verifier's compliance verification times were 4 to 5 orders of magnitude smaller for a link up event (median = 50 μs, maximum = 1.5 ms) and a link down event (median = 1 μs, maximum = 27 μs).

The balance of this paper is organized as follows. In Section II, we present our models of a network and a middle box. We describe how port predicates of each box are computed from rules in its forwarding table and ACLs. In Section III, we define atomic predicates. Given a set of predicates, we present an algorithm for computing the set of atomic predicates, which is proved to be minimum and unique. We present statistics of three real networks [1], [16], [2] including the sizes of their atomic predicate sets and their computation times. In Section IV, we present algorithms for computing reachability and verifying a number of network properties. We present computation time and storage costs comparing AP Verifier with Hassel in C (the fast version used in Header Space and NetPlumber [11], [10]). In Section V, we present algorithms for processing network state changes due to rule insertion/deletion and link up/down events and checking reachability compliance in real time. We present results comparing the computation times of AP Verifier and NetPlumber [10]. In Section VI we discuss related work. We provide answers to questions from the reviewers in Section VII and conclude in Section VIII.

II. Network Model

We model a packet network as a directed graph of middle boxes. A middle box can be a switch or a router. A middle box has a forwarding table as well as input and output ports guarded by access control lists (ACLs). Each packet has a header of h bits. The header is partitioned into multiple fields. The three networks analyzed in this paper are all IP networks. However, our model of packet headers is general and not limited to IP headers.

Each ACL consists of a list of ACL rules. Each ACL rule is specified by a predicate and an action. (Our model includes firewalls, which are ACLs with large numbers of rules.) The variables of the predicate are packet header fields. AP Verifier has a parser for converting ACL rules written in Cisco IOS to predicates. *All predicates in AP Verifier are represented by binary decision diagrams (BDDs)* which are rooted, directed acyclic graphs. Logical operations on BDDs can be performed efficiently using graph-based algorithms [5]. (We use the software package JDD [17].) Consider an ACL with m rules:

$$\begin{array}{c} G_1, action_1 \\ G_2, action_2 \\ \cdots \\ G_m, action_m \end{array}$$

where G_i is the predicate for the ith rule and $action_i$ is $allow$ or $deny$.

When a packet is checked against an ACL, it is matched by the first rule whose predicate evaluates to true for the packet. From the predicates in rules, we use Algorithm 1 to compute a single predicate that specifies the packet set allowed by the ACL. (Predicate *false* specifies the empty set.)

Algorithm 1 Converting an ACL to a predicate

```
Input: An ACL
Output: A predicate for the ACL
1: $allowed \leftarrow false$, $denied \leftarrow false$
2: for $i = 1$ to $m$ do
3:     if $action_i = deny$ then
4:         $denied \leftarrow denied \vee G_i$
5:     else
6:         $allowed \leftarrow allowed \vee (G_i \wedge \neg denied)$
7:     end if
8: end for
9: return $allowed$
```

If the allowed values of each header field in an ACL rule are specified by a suffix, prefix or an interval, we proved that the predicate of an ACL rule can be represented by a BDD with $\leq 2 + 2h$ nodes, where h is the number of bits in the packet header (see Appendix). We found that this constraint is satisfied by each ACL rule in the several datasets we have (including those from Stanford and Purdue). For an ACL rule in which the allowed values of a header field are specified by multiple disjoint intervals, the number of nodes in the rule's BDD may be larger than $2 + 2h$ but *Algorithm 1 remains the same*. (Such an ACL rule can be replaced in the ACL by a sequence of rules, one for each disjoint interval of allowed values in the header field, specifying the same allowed packet set.)

The forwarding table in a middle box is also a list of rules. Each rule has an IP prefix and a port name. The port may be physical or virtual. There is also a special port for packets to be intentionally dropped. AP Verifier has parsers for converting forwarding rules written in Cisco IOS and Juniper JUNOS to predicates. We first convert each prefix to a predicate represented by a BDD. The number of nodes in the BDD is $\leq n + 2$ where n is the number of bits in an IP address.

In IP forwarding, a packet may be matched by multiple rules in the table; the packet is forwarded to the output port specified by the matched rule with the longest prefix. To compute a single predicate for each port in the forwarding table, which has k ports indexed by $\{1, \ldots, k\}$, we first sort the rules in the table in descending order of prefix length and represent the forwarding table with m rules as follows:

$$Pre_1, L_1, port_1$$
$$Pre_2, L_2, port_2$$
$$\dots$$
$$Pre_m, L_m, port_m$$

where Pre_i denotes a prefix; L_i, $i \in \{1, \dots, m\}$, are prefix lengths such that $L_1 \geq L_2 \geq \dots L_m$; and $port_i \in \{1, \dots, k\}$. Then we use Algorithm 2 to convert the sorted forwarding table to a list of predicates, one for each output port.

Algorithm 2 Converting a forwarding table to forwarding predicates

```
Input: A sorted forwarding table
Input: A set of output ports $\{1, \dots, k\}$
Output: A list of predicates $\{P_1, \dots, P_k\}$
1: for $j = 1$ to $k$ do
2:     $P_j \leftarrow false$
3: end for
4: $fwd \leftarrow false$
5: for $i = 1$ to $m$ do
6:     $P_{port_i} \leftarrow P_{port_i} \vee (Pre_i \wedge \neg fwd)$
7:     $fwd \leftarrow fwd \vee Pre_i$
8: end for
9: return $\{P_1, \dots, P_k\}$
```

From the several datasets we have (including those from Stanford, Purdue, and Internet2), we observed that the number of BDD nodes used to represent an ACL or a forwarding table increases approximately linearly with the number, m, of rules in the ACL/table up to a maximum and then decreases as m increases further. For example, in the Purdue dataset, an ACL with 52 rules is represented by 515 BDD nodes (maximum); the ACL with the most rules (693) is represented by only 187 BDD nodes. In the Stanford dataset, a forwarding table with 1,825 rules is represented by 5,325 BDD nodes (maximum); the forwarding table with the most rules (184,908) is represented by only 1,900 BDD nodes. (More results are presented in Appendix.)

A port in a middle box may be a virtual port (e.g. a VLAN port) which has a set of physical ports corresponding to it. We map the ACL and forwarding predicates of a virtual port to its set of physical ports. As a result, a physical port can have multiple ACLs; also, a physical output port can have multiple predicates computed from the same forwarding table. For reachability computation, the ACL predicate of a physical port is the disjunction of its predicates computed from all of the ACLs. The forwarding predicate of a physical output port is the disjunction of all of its predicates computed from the forwarding table.

Thus we have the model shown in Fig. 1, namely: a middle box with a set of physical input ports and a set of physical output ports. Each input port is guarded by an ACL predicate. Each output port is guarded by a forwarding predicate followed by an ACL predicate.

III. Atomic Predicates

A. Basic Idea

Definition 1 (*Atomic Predicates*)**.** Given a set $\mathcal{P}$ of predicates, its set of atomic predicates $\{p_1, \dots, p_k\}$ satisfies these five

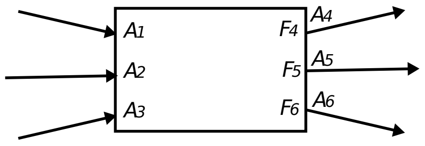


Fig. 1: An example of a middle box. $A_1, \dots, A_6$ are ACL predicates and F_4, F_5, F_6 are forwarding predicates.

properties:

1) $p_i \neq false, \forall i \in \{1, \dots, k\}$.
2) $\vee_{i=1}^{k} p_i = true$.
3) $p_i \wedge p_j = false$, if $i \neq j$.
4) Each predicate $P \in \mathcal{P}$, $P \neq false$, is equal to the disjunction of a subset of atomic predicates:
$$P = \bigvee_{i \in S(P)} p_i, \text{ where } S(P) \subseteq \{1, \dots, k\}. \quad (1)$$
5) k is the *minimum* number such that the set $\{p_1, \dots, p_k\}$ satisfies the above four properties.

Note that if $P = true$, then $S(P) = \{1, \dots, k\}$; if $P = false$, $S(P) = \emptyset$. Since $p_1, \dots, p_k$ are disjoint, the expression in equation (1) is *unique* for each predicate $P \in \mathcal{P}$.

Given a set $\mathcal{P}$, there are numerous sets of predicates that satisfy the first four properties of Definition 1. In the trivial case, these four properties are satisfied by the set of predicates each of which specifies a single packet. We are interested in the set with the *smallest* number of predicates. The meaning of atomic predicates is provided by the following theorem (proof in Appendix A).

Theorem 1. For a given set $\mathcal{P}$ of predicates, the set of atomic predicates for $\mathcal{P}$ specifies the *minimum* set of equivalence classes in the set of all packets.

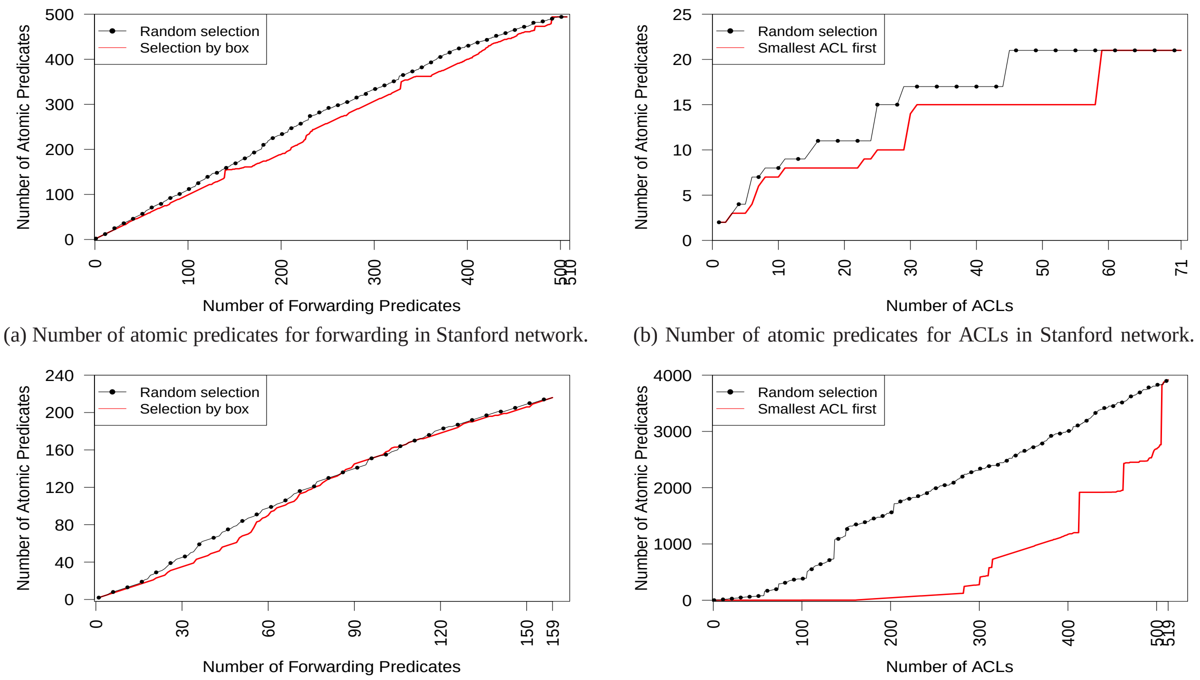
To enable fast computation of reachability in a network, AP Verifier precomputes the set of atomic predicates for all port predicates of the network. The set of atomic predicates together with the network topology preserve all network reachability information but without any redundant information in ACL rules and forwarding rules. Thus, AP Verifier is space efficient.

More importantly, the conjunction of two predicates, P_1 and P_2 in $\mathcal{P}$, can be computed by the intersection of two sets of integers, $S(P_1)$ and $S(P_2)$. Similarly, the disjunction of P_1 and P_2 can be computed by the union of two sets of integers, $S(P_1)$ and $S(P_2)$. Operations on predicates (or operations on packet sets) are highly computation-intensive because they operate on many packet header fields. Using atomic predicates, these computation-intensive operations are replaced by operations on sets of integers (i.e., identifiers of atomic predicates) with a dramatic decrease in computation time. Thus, AP Verifier is also time efficient.

B. Computing Atomic Predicates

For a given set $\mathcal{P}$ of predicates, we present an algorithm to compute its set of atomic predicates, denoted by $\mathcal{A}(\mathcal{P})$.

First, we compute the set of atomic predicates for each predicate P in $\mathcal{P}$ using equation (2) below. It is easy to see



(a) Number of atomic predicates for forwarding in Stanford network.

(b) Number of atomic predicates for ACLs in Stanford network.

(c) Number of atomic predicates for forwarding in Internet2.

(d) Number of atomic predicates for ACLs in Purdue network.

Fig. 2: Number of atomic predicates for forwarding and for ACLs in three real networks.

that $\mathcal{A}(\{P\})$ satisfies Definition 1.

$$\mathcal{A}(\{P\}) = \begin{cases} \{true\} & \text{if } P = false \text{ or } true \\ \{P, \neg P\} & \text{otherwise.} \end{cases} \tag{2}$$

Second, let $\mathcal{P}_1, \mathcal{P}_2$ be two sets of predicates, $\mathcal{P}_1$'s set of atomic predicates be $\{b_1, \ldots, b_l\}$ and $\mathcal{P}_2$'s set of atomic predicates be $\{d_1, \ldots, d_m\}$. We compute a set of predicates $a_1, \ldots, a_k$ as follows:

$$\{a_i = b_{i_1} \wedge d_{i_2} | a_i \neq false, i_1 \in \{1, ..., l\}, i_2 \in \{1, ..., m\}\} \tag{3}$$

In the worst case, the above set can have $l \times m$ predicates. However, in practice we found that most intersections in (3) are *false*. The following theorem states that $\{a_1, \ldots, a_k\}$ is the set of atomic predicates for $\mathcal{P}_1 \cup \mathcal{P}_2$ (proof in Appendix B).

Theorem 2. The set of atomic predicates for $\mathcal{P}_1 \cup \mathcal{P}_2$ is $\{a_1, \ldots, a_k\}$ where, for $i \in \{1, \ldots, k\}$, a_i is computed by formula (3).

Given a set of predicates, $\mathcal{P} = \{P_1, \ldots, P_N\}$, Algorithm 3 computes the set of atomic predicates for $\mathcal{P}$.

Algorithm 3 Computing atomic predicates

```
Input: $\{P_1, P_2, \ldots, P_N\}$
Output: $\mathcal{A}(\{P_1, P_2, \ldots, P_N\})$
1: for $i = 1$ to $N$ do
2:     compute $\mathcal{A}(\{P_i\})$ using equation (2)
3: end for
4: for $i = 2$ to $N$ do
5:     compute $\mathcal{A}(\{P_1, \ldots, P_i\})$ from $\mathcal{A}(\{P_1, \ldots, P_{i-1}\})$ and
       $\mathcal{A}(\{P_i\})$ using formula (3)
6: end for
7: return $\mathcal{A}(\{P_1, \ldots, P_N\})$
```

Algorithm 3 uses formula (3) repeatedly. Theorem 2 ensures that Algorithm 3 returns the correct set of atomic predicates. Since the set of atomic predicates is unique, it is independent of the predicates' order in the list given to Algorithm 3 as input. The computation time however is affected by the predicates' order (see next subsection). We note that the computation time can be improved by treating $\mathcal{A}(\{P_i\})$, $i = 1, \ldots, N$ as leaf nodes of a binary tree and using formula (3) to compute other tree nodes from their children until the root node is computed. However, since the computation times of Algorithm 3 are quite small even for large networks (see next subsection), we have not tried to improve it.

	Stanford	Internet2	Purdue
No. of middle boxes	16	9	1,646
No. of ports used	58	56	2,736

	Stanford		Internet2	Purdue
No. of rules	Forwarding	ACL	Forwarding	ACL
	757,170	1,584	126,017	3,605
No. of atomic predicates	494	21	216	3,917

TABLE I: Statistics of three real networks.

C. *Atomic Predicates in Real Networks*

We downloaded datasets of three real networks from Stanford University [1], Purdue University [16], and Internet2 [2]. Some network statistics are shown in Table I. All 16 middle boxes in the Stanford dataset are routers. All 9 middle boxes in the Internet2 dataset are routers. The 1,646 middle boxes in the Purdue dataset consist of routers and switches. We observed that forwarding and ACL rules have different characteristics and locality properties. Therefore we consider ACL and forwarding rules separately and compute *separate sets of atomic predicates for ACL and forwarding predicates.*

To compute atomic predicates for ACLs using Algorithm 3, we experimented with two ways for ordering the ACL predicates:

- *Random selection:* Select an ACL randomly.
- *Smallest ACL first:* Select an ACL with the smallest number of rules.

To compute atomic predicates for forwarding, we also experimented with two ways for ordering the forwarding predicates:

- *Random selection:* Select a forwarding predicate randomly.
- *Selection by box:* Select a middle box randomly and then select its forwarding predicates one by one randomly.

Figure 2 shows growth of the number of atomic predicates in the three networks versus the number of forwarding/ACL predicates. Figures 2(a) and (c) show that for forwarding predicates, the number of atomic predicates grows approximately linearly with the number of forwarding predicates whichever selection method is used. Figures 2(b) and (d) show that when ACLs are selected randomly, the number of atomic predicates grows approximately linearly with the number of ACLs. But with *smallest ACL first*, the number of atomic predicates remains low for a long time until near the end of the computation (thus requiring less computation time).

From Table I and Figure 2, observe that the Stanford network has 71 ACLs with 1,584 rules but only 21 atomic predicates for these ACLs – a surprisingly small number which indicates large amounts of redundancy in the rules as well as similarity between ACLs. The number of atomic predicates is 3,917 for Purdue's 519 ACLs with 3,605 rules; we found that the Purdue dataset contains many different rules and a sizable number of extended ACL rules. The number of atomic predicates is 494 for Stanford's 757,170 forwarding rules. The number of atomic predicates is 216 for Internet2's 126,017 forwarding rules.

atomic predicates for ACLs		
	random selection (ms)	*smallest ACL first* (ms)
Stanford	1.56	0.84
Purdue	886.21	450.31

atomic predicates for forwarding		
	random selection (ms)	*selection by box* (ms)
Stanford	210.26	201.40
Internet2	154.91	148.28

TABLE II: Time to compute atomic predicates.

Table II shows times used to compute atomic predicates for the three networks.[2] Table II shows that for ACLs *smallest ACL first* uses about 50% less time than *random selection*. For forwarding tables, the computation time of *selection by box* is slightly smaller than the time of *random selection*. We will use *smallest ACL first* to compute atomic predicates for ACLs and *selection by box* to compute atomic predicates for forwarding.

The computation times for ACL atomic predicates in the Stanford and Purdue networks were 0.84 ms and 0.45 second, respectively. The computation times for forwarding atomic predicates in the Stanford network and Internet2 were 0.2 and 0.15 second, respectively.

[2]All results in this paper were computed using just *one core* of a six-core Xeon processor with 12 MB of L3 cache and 16 GB of DRAM.

D. Packet Set Specification

The set of packets that can pass through an output port is specified by the conjunction of its forwarding and ACL predicates. For a particular port, let F and A denote the forwarding and ACL predicates, respectively. Let S_F denote the set of integer identifiers of atomic predicates for forwarding. Let S_A denote the set of integer identifiers of atomic predicates for ACLs. Then the set of packets that can pass through the output port is specified by the predicate

$$P = (\vee_{i \in S_F} f_i) \wedge (\vee_{j \in S_A} a_j) \quad (4)$$

where f_i and a_j denote atomic predicates for forwarding and ACLs, respectively.

IV. Computing Reachability and Verifying Network Properties

Consider a network represented by a directed graph of middle boxes. Any full-duplex physical link connecting two boxes is represented as two unidirectional logical links; each logical link connects the output port of one box to the input port of the other box. Each input port is guarded by an ACL predicate. Each output port is guarded by a forwarding predicate followed by an ACL predicate. If a predicate is true, any packet can pass through. If a predicate is false, no packet can pass through. (Notation: In figures in this paper, if a port is not labeled by any predicate identifier, its predicate is assumed to be *true*.)

In this section, we first present an algorithm for computing the set of packets that can travel from a port s to another port d in the network (more specifically, *from the entry point of* s *to the exit point of* d). We next describe how the algorithm is extended to compute the reachability tree from s. Such a reachability tree is labeled by sets of integer identifiers of atomic predicates. Operations on sets of integers are extremely fast. The reachability trees from ports can be computed quickly and stored efficiently. AP Verifier can be extended to check the network's compliance with most safety and temporal properties, such as, properties specified using CTL [6].

We will describe how to verify several specific network properties, namely: loop detection, black hole detection, network slice isolation, and required waypoints. Using the datasets from Stanford University and Internet2, we present computation results and compare the performance of AP Verifier versus Hassel in C [1], [10].

A. Reachability trees

We first consider a path from port s to port d. Let $F_1, \ldots, F_j$ be the forwarding predicates in the path represented by $S(F_1), \ldots, S(F_j)$. Let $A_1, \ldots, A_k$ be the ACL predicates in the path represented by $S(A_1), \ldots, S(A_k)$. (Any predicate equal to $true$ is not represented.) Algorithm 4 computes the reachability set from s to d along the path. In steps 1-2, S_F and S_A represent the set of all packets that are injected into port s to test reachability. If the Algorithm returns $false$, port d is not reachable from port s. The reachability set from s to d, represented by S_F and S_A returned in step 11, is specified by the predicate P in equation (4). If there are multiple paths from s to d, then the reachability set is the union of the reachability sets of the paths.

Note that reachability can be computed from any port to any other port in the network. The source port s does not have

	Average (ms)	Median (ms)	Maximum (ms)
Hassel in C	233.57	48.57	2086.71
AP Verifier	0.91	0.98	1.48

(a) Port to port reachability computation in Stanford network.

	Average (ms)	Median (ms)	Maximum (ms)
Hassel in C	757.73	610.80	7433.85
AP Verifier	0.26	0.29	0.48

(b) Port to port reachability computation in Internet2.

	Average (ms)	Median (ms)	Maximum (ms)
Hassel in C	218.22	53.45	1881.41
AP Verifier	0.95	1.03	1.38

(c) Loop detection from one port in Stanford network.

	Average (ms)	Median (ms)	Maximum (ms)
Hassel in C	754.19	609.14	5873.44
AP Verifier	0.27	0.29	0.45

(d) Loop detection from one port in Internet2.

	Average (ms)	Median (ms)	Maximum (ms)
AP Verifier	0.011	0.0064	0.040

(e) Black hole detection for each forwarding table in Stanford network.

	Average (ms)	Median (ms)	Maximum (ms)
AP Verifier	0.014	0.014	0.027

(f) Black hole detection for each forwarding table in Internet2.

TABLE III: Computation times of reachability, loop detection, and black hole detection.

to be an input port that accepts packets from an external host or box. The destination port d does not have to be an output port that connects to an external host or box.

Algorithm 4 Computing $s-d$ reachability along a path

```
Input: $S(F_1), \ldots, S(F_j)$, and $S(A_1), \ldots, S(A_k)$
Output: packet set specification
 1: $S_F \leftarrow \{1, \ldots, I\}$        // identifiers of atomic predicates
 2: $S_A \leftarrow \{1, \ldots, J\}$        // identifiers of atomic predicates
 3: $S_F \leftarrow S_F \cap S(F_1) \cap \cdots \cap S(F_j)$
 4: if $S_F = \emptyset$ then
 5:     return $false$
 6: end if
 7: $S_A \leftarrow S_A \cap S(A_1) \cap \cdots \cap S(A_k)$
 8: if $S_A = \emptyset$ then
 9:     return $false$
10: end if
11: return $S_F, S_A$        // packet set specification
```

We compare the computation times of AP Verifier versus Hassel in C [1] for the Stanford network and Internet2.[3] For each network, we compute reachability sets for all port pairs and measure the time used for each pair. The results are presented in Table III(a) and (b). On the average, AP Verifier is 256 times faster than Hassel in C for the Stanford network and it is 2,914 times faster than Hassel in C for Internet2.

The *reachability tree from a port* s to all other ports in the network is computed by performing a depth-first search which begins with visiting port s. The packet set injected into port s is the set of all packets (same as lines 1 and 2 in Algorithm 4). When the search visits a port, S_F and S_A are intersected with the sets representing the port's forwarding and ACL predicates, respectively (any port predicate equal to $true$ is not represented.) A search branch is terminated after visiting a port (say x) if one of the following conditions holds: (i) S_F or S_A becomes empty after visiting port x; (ii) port x is an output port and there is no link connecting x to an input port; (iii) port x is an input port of a box with no output port; (iv) port x has been visited before in the search (*loop detected*). In each case, the search backtracks and depth-first search continues until no more port can be reached. When search terminates, a reachability tree from port s to all reachable ports is created. Each node in the tree has a port number and two sets of integers, S_F and S_A, specifying the set of packets that can reach and pass through the port.

[3]The C version of Hassel is faster than the Python version [11] by about two orders of magnitude.

Figure 3 shows a small network example. The network has 6 atomic predicates, $f_1, f_2, f_3, f_4, f_5, f_6$, for forwarding and 2 atomic predicates, a_1, a_2, for ACLs. Ports that filter packets are labeled by integer identifiers of atomic predicates specifying packets allowed to pass (ports without labels allow all packets to pass). For examples, port 1 allows all packets to pass; port 3 labeled by $S(F_3) = \{1, 2, 3\}$ forwards only packets in predicate $f_1 \vee f_2 \vee f_3$. The ACL of port 6 labeled by $S(A_6) = \{2\}$ allows only packets in predicate a_2 to pass.

The reachability tree from $port_1$ is shown in Figure 4. Each node in the tree is a port with two sets of integers separated by a semicolon. Integers before the semicolon identify atomic predicates for forwarding. Integers (in bold italics) after the semicolon identify atomic predicates for ACLs. For example, the expression "1,2,3,4,5,6; ***1***,***2***" represents the set of all packets injected into port 1. As another example, port 6 is labeled by "4,5,6; ***2***" with the following meaning: packets that satisfy the predicate $(f_4 \vee f_5 \vee f_6) \wedge a_2$ can reach and pass through port 6. Note that a port can appear as nodes in different paths of the reachability tree, such as, ports 8, 9, and 10 in Figure 4.

Optimization techniques. AP Verifier uses several optimization techniques to reduce time for checking various network properties. First, AP Verifier maintains a hash table, HT, of (key, value) pairs. A key is a port number (or name). Given a key, say port number x, its value is the set of tree nodes each of which has port number x. HT can be used to query the reachability set from a source port s to some destination port d without traversing the reachability tree from s. Function $HT(d)$ returns the set of port d nodes in s's reachability tree.

Second, when computing the reachability tree from a source port s, AP Verifier stores in each tree node (say port y) the *set of ports along the path from* s *to the tree node* (y). Port set information enables fast loop detection without traversing the reachability tree.

Third, if a set of integer identifiers (such as, S_F, S_A, $S(F_i)$, or $S(A_j)$) is too large, the set's complement is stored and used instead.

B. Storage Costs of Reachability Trees

We compare the memory requirements of Hassel in C and AP Verifier for storing reachability trees computed for all ports of the Stanford network and Internet2. The results are presented in Table IV. Hassel in C required 37 times more memory for the Stanford network and 28 times more memory for Internet2 than AP Verifier. Furthermore, we monitored the maximum memory used to store intermediate data when reachability trees were computed one at a time. The maximum memory was over 400

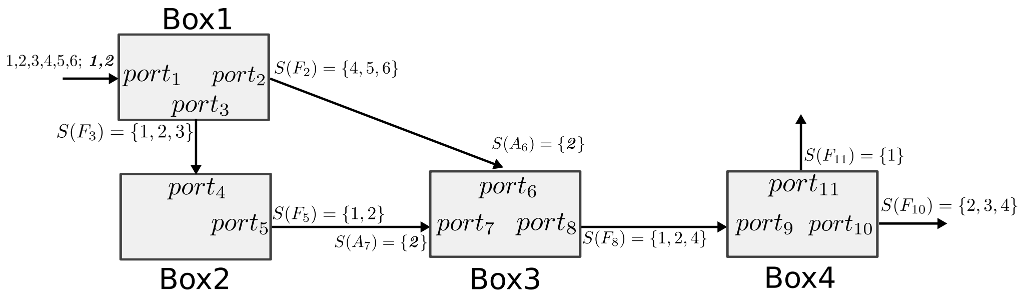


Fig. 3: A small network example.

$$\begin{tikzcd}
{} & {} & \overset{4,5,6;\ \boldsymbol{1,2}}{port_2} \arrow[r] & \overset{4,5,6;\ \boldsymbol{2}}{port_6} \arrow[r] & \overset{4;\ \boldsymbol{2}}{port_8} \arrow[r] & \overset{4;\ \boldsymbol{2}}{port_9} \arrow[r] & \overset{4;\ \boldsymbol{2}}{port_{10}} \arrow[r] & {} & \overset{1;\ \boldsymbol{2}}{port_{11}} \arrow[r] & {} \\
{} \arrow[r, "1{,}2{,}3{,}4{,}5{,}6;\ \boldsymbol{1{,}2}"] & port_1 \arrow[ur] \arrow[r] & \underset{1,2,3;\ \boldsymbol{1,2}}{port_3} \arrow[r] & \underset{1,2,3;\ \boldsymbol{1,2}}{port_4} \arrow[r] & \underset{1,2;\ \boldsymbol{1,2}}{port_5} \arrow[r] & \underset{1,2;\ \boldsymbol{2}}{port_7} \arrow[r] & \underset{1,2;\ \boldsymbol{2}}{port_8} \arrow[r] & \underset{1,2;\ \boldsymbol{2}}{port_9} \arrow[ur] \arrow[r] & \underset{2;\ \boldsymbol{2}}{port_{10}} \arrow[r] & {}
\end{tikzcd}$$

Fig. 4: The reachability tree of $port_1$.

MB for Hassel in C and was less than 1 MB for AP Verifier.

	Size (MB)
Hassel in C	323.06
AP Verifier	8.70

(a) Stanford network (58 ports).

	Size (MB)
Hassel in C	187.60
AP Verifier	6.72

(b) Internet2 (56 ports).

TABLE IV: Storage costs of reachability trees from ports.

C. Loop Detection

Loop detection is performed by computing the reachability tree for every port, as described above.

We used AP Verifier and Hassel in C to detect loops in the Stanford network and Internet2. For the Stanford network, we computed reachability trees for 30 ports as was done previously [11]. Twelve infinite loop paths were detected by both AP Verifier and Hassel in C. For Internet2, we computed reachability trees for all ports. Two infinite loop paths were detected by both AP Verifier and Hassel in C. Their computation times are presented in Table III(c) and (d). On the average, AP Verifier is 230 times faster than Hassel in C for the Stanford network and 2,793 times faster for Internet2.

D. Black Hole Detection

A black hole in the forwarding table of a box is a set of packets that are dropped due to no forwarding entry (rather than intentionally). Finding black holes in the forwarding table of a box is very easy for AP Verifier. Let $S(F_1), S(F_2), \dots S(F_k)$ be sets of identifiers of atomic predicates for output ports, $1, 2, \dots, k$ of the box (including the special port for intentional packet drop). Let $S(\mathit{true})$ be the set of identifiers of all atomic predicates for forwarding. The set of black holes is represented by the set,

$$S(\mathit{true}) - \cup_{i=1}^{k} S(F_i) \tag{5}$$

If the above set is empty, the forwarding table has no black hole.

We checked for black holes in each forwarding table in the Stanford network and Internet2. The computation times for AP Verifier are presented in Table III(e) and (f). On the average, AP Verifier took 11 μs for the Stanford network and 14 μs for Internet2. It found no black hole in forwarding tables of the Stanford network. It found black holes in every forwarding table of Internet2.

E. Slice Isolation

Network operators provide different network slices (virtual networks, e.g., VLANs) to customers/applications and must ensure that the slices do not overlap; any overlap would allow packets to leak from one slice to another. A slice can be defined by a set of ports together with a set of packets allowed in the slice.

In AP Verifier, a set of packets is represented by two sets of identifiers of atomic predicates for forwarding and ACLs. Consider two slices, $Slice_1$ and $Slice_2$. $Slice_1$ has a set, T_1, of ports and a set of packets represented by S_{F_1} and S_{A_1}. $Slice_2$ has a set, T_2, of ports and a set of packets represented by S_{F_2} and S_{A_2}. To check whether $Slice_1$ and $Slice_2$ overlap, AP Verifier first computes $T_1 \cap T_2$. If the intersection is empty, then the two slices are isolated; else, it computes $S_F = S_{F_1} \cap S_{F_2}$. If S_F is empty, then the two slices are isolated; else, it computes $S_A = S_{A_1} \cap S_{A_2}$. If S_A is empty, then the two slices are isolated; else, $Slice_1$ overlaps $Slice_2$ at ports $T_1 \cap T_2$ and the set of packets shared by both slices is specified by S_F and S_A.

F. Required Waypoints

Many networks have one or more required waypoints (e.g., firewalls) through which all packets from a source port s must pass through before reaching a specified set of destination ports. Consider a single middle box, with several input ports, which is a required wayppoint for all packets from source port s. To verify compliance with the waypoint requirement, AP Verifier traverses the reachability tree from s to check that every path in the tree passes through an input port of the waypoint before reaching any destination port in the specified set. AP Verifier returns $true$ or a set of paths that avoid the waypoint.

Checking compliance with the waypoint requirement from a set of source ports to a set of destination ports is performed by traversing the reachability tree of every source port in the specified set. It is also straightforward to check the waypoint requirement that all packets from port s pass through any member of a set of waypoints or the requirement that all packets from port s pass through several waypoints in a specified sequence before reaching specified destination ports.

V. Real-time Compliance Check for Network State Changes

In this section, we describe how AP Verifier handles rule insertion/deletion and link up/down events which change the network state. For performance comparison, we performed the same benchmark experiments for link up and rule insertion events described in the NetPlumber paper [10]. We also provide performance results for link down and rule delection events not reported in the paper. In these *benchmark experiments*, the reachability tree of a port is precomputed which satisfies a network property or reachability policy. We investigate the time used by AP Verifier to update the reachability tree when a state change event is detected. We performed one experiment for the reachability tree of each of Stanford network's 58 ports and Internet2's 56 ports.

A. Link Status Change

The sets of atomic predicates are derived from a network's forwarding predicates and ACL predicates and, therefore, do not depend on the status of any link in the network. The reachability tree of a port, however, depends on network topology and thus the status of each link. In each experiment, the reachability tree from a port and its hash table, *HT*, are precomputed. For a link up/down event, AP Verifier needs to update the reachability tree and *HT*.

Consider a link down event for a bidirectional link with two output ports. For each of the two ports, AP Verifier uses HT to locate nodes in the reachability tree identified by the two port numbers. It removes these nodes and all of their descendant nodes from the reachability tree and from the hash table.

Consider a link up event for a bidirectional link with two output ports. For each of the two ports, AP Verifier uses HT to locate nodes in the reachability tree identified by the two port numbers. From each node located, it performs a depth-first search to extend the reachability tree. It also adds new nodes from the subtrees to HT.

The benchmark performance results of AP Verifier are summarized in Table V(a)-(d) for the Stanford network and Internet2. For each link in a reachability tree, we performed two experiments for link down and link up. We measured the time to update the reachability tree. AP Verifier's results are compared with those reported for NetPlumber.[4] On the average, AP Verifier is *4-5 orders of magnitude* faster than NetPlumber.

The Veriflow paper [13] reports that its average time to verify a link failure was 1.15 seconds with a maximum of 4.05 seconds. Veriflow experiments were performed for a different network, i.e., a synthetic network with a Rocketfuel topology and BGP update traces.

B. Rule Update

When a rule is inserted into, or deleted from, a forwarding table, it may change a forwarding port predicate. (For an ACL rule update, the following description is similar and will not be repeated.) As a result the set of atomic predicates for forwarding may change. To update a port's reachability tree being used for reachability compliance check, AP Verifier running on one processor core performs these steps: (i) It checks if a port predicate is changed by the rule update; if so, it computes a new predicate for the port. (ii) It updates the reachability tree using the new predicate, if any. (iii) It forks a process which runs on a second core to update the set of atomic predicates. Steps (ii) and (iii) occur concurrently.

Steps (i) and (ii) can be completed in hundreds of μs on the first core. The updated reachability tree is correct and can be used for compliance check but is intended to be temporary. In a temporary reachability tree, nodes of the port affected by a rule update store a new predicate whose representation by atomic predicates has not been resolved. Let S_F and S_A represent the set of packets that can arrive at the port's entry point. Suppose the port's ACL predicate is A_{port} and the port's forwarding predicate, F_{port}, has been changed to F'_{port} which is unresolved. After the rule update, the set of packets that can pass through the port is represented by $S(A_{port}) \cap S_A$, S_F, and F'_{port}, which together specify the following predicate:

$$(\vee_{j \in S(A_{port}) \cap S_A} a_j) \wedge (\vee_{i \in S_F} f_i) \wedge F'_{port}$$

The port's descendant nodes in the subtree are updated accordingly (more details in Appendix).

If rule updates arrive in rapid succession, AP Verifier can keep on updating the temporary reachability tree correctly (however, computation time increases as the number of unresolved predicates in the tree increases).

The process running on the second core can compute the updated set of atomic predicates in 10 ms on the average for one rule update. If the updated set of atomic predicates is unchanged *and* there are three[5] or fewer unresolved predicates, the process running on the first core replaces each unresolved predicate in the temporary tree with its atomic predicate identifiers and converts the temporary tree to a "normal" one. Otherwise, the process deletes the temporary tree and computes a new reachability tree directly from the updated set of atomic predicates. It can do so in less than 1 ms most of the time. (See Tables III(c) and III(d); note that loop detection for a port is performed by computing its reachability tree.)

We performed benchmark experiments [10] using AP Verifier for the Stanford network and Internet2. For each network, the reachability tree of a port was first computed using 90% of rules selected at random. (For the Stanford network, the rules include ACL and forwarding rules.) The 10% of rules remaining were inserted one by one and the time for updating the reachability tree was measured. We also ran experiments for each network with 100% of the rules initially. Ten percent of the rules were then selected one by one for deletion; the time for updating the reachability tree after each rule deletion was measured. Results are presented in Table V(e)-(h). For rule insertions, the performance of AP Verifier is comparable to NetPlumber for the Stanford network; it is better than NetPlumber for Internet2.

VI. Related Work

A model for static reachability analysis of network state in the data plane was first presented by Xie et al. [19]. Gouda and Liu presented firewall decision diagram (FDD) for formal analysis of firewalls [7] and distributed firewalls [8]. Quarnet

[4]NetPlumber results were computed using 6-core Xeon processors with 12 MB of L2-cache and 12 GB of DRAM [10].

[5]This is a configurable parameter value.

	Average (ms)	Median (ms)	Maximum (ms)
NetPlumber	3020.00	2120.00	(not reported)
AP Verifier	0.16	0.037	1.55

(a) Link up in Stanford network.

	Average (ms)	Median (ms)	Maximum (ms)
NetPlumber	4760.00	2320.00	(not reported)
AP Verifier	0.027	0.0067	0.36

(b) Link up in Internet2.

	Average (ms)	Median (ms)	Maximum (ms)
AP Verifier	0.0028	0.00094	0.27

(c) Link down in Stanford network.

	Average (ms)	Median (ms)	Maximum (ms)
AP Verifier	0.0016	0.0011	0.10

(d) Link down in Internet2.

	Average (ms)	Median (ms)	Maximum (ms)
NetPlumber	0.2	0.065	(not reported)
AP Verifier	0.29	0.077	26.44

(e) Rule insertion in Stanford network.

	Average (ms)	Median (ms)	Maximum (ms)
NetPlumber	0.53	0.52	(not reported)
AP Verifier	0.35	0.19	10.40

(f) Rule insertion in Internet2.

	Average (ms)	Median (ms)	Maximum (ms)
AP Verifier	0.32	0.083	12.71

(g) Rule deletion in Stanford network.

	Average (ms)	Median (ms)	Maximum (ms)
AP Verifier	0.35	0.13	46.24

(h) Rule deletion in Internet2.

TABLE V: Computational times for dynamic updates.

uses FDDs to represent ACLs in packet networks; it used tens to hundreds of seconds to compute reachability along paths with ACLs only [12].

There were two proposals to use general verification tools from other application domains. ConfigChecker [3] uses boolean formulas to specify state transition relations of packet sets before and after a packet filter. It applies symbolic model checking to check network properties. Anteater [14] uses boolean formulas to represent packet filters and a SAT solver for checking network properties. Both of these general-purpose tools are slow and operate on time scales of seconds to hours [13].

Custom-designed methods for reachability computation include Header Space/Hassel in C [11], NetPlumber [10], and Veriflow [13]. We have compared the performance of AP Verifier versus Hassel in C and NetPlumber and showed that AP Verifier is much more time and space efficient.

Veriflow aggregates packets into equivalence classes (ECs) by first storing all rules in a *multi-dimensional prefix tree (trie)* [13]. An EC is defined by a particular choice of one of the disjoint intervals of allowed values for every header field in the trie. After tens of thousands of rules are inserted in the trie, the number of disjoint intervals for each header field is numerous. For ACL rules which specify allowed values for many header fields, the number of ECs is the product of the set sizes of disjoint intervals and is very large. The performance of Veriflow was demonstrated mostly for forwarding rules with only one header field. Veriflow used one to several seconds of time to verify the compliance of a link down event for a synthetic network. AP Verifier used 10 μs for most link failures (maximum of 0.27 ms for the Stanford network and maximum of 0.1 ms for Internet2).

VII. Answers to Questions from Reviewers

Question: *How would AP Verifier perform on backbone ISP networks?* ***Answer***: The Stanford dataset has 757,170 prefixes in forwarding tables of which 197,808 are distinct. The number of distinct prefixes in the forwarding tables of a backbone ISP network may be twice as many. We believe that AP Verifier will scale and can be used for verifying reachability properties of backbone ISP networks. (We need access to the forwarding tables and ACLs of a backbone ISP.)

Question: *Can AP Verifier be extended to include firewall rules?* ***Answer***: AP Verifier is already designed and implemented to include firewall rules. A firewall is an ACL with a large number of rules. In the Stanford dataset, there are 12 ACLs with 50 to 111 rules. In the Purdue dataset, there are 5 ACLs with 52 to 111 rules and one ACL with 693 rules. These ACLs were included in our experiments. Furthermore, our model of packet headers is general. A packet header is a sequence of h bits partitioned into multiple fields. Our parsers and AP Verifier can be extended to handle nonstandard rules.

Question: *Why is there no experimental comparison using the Purdue dataset? It is possible that AP Verifier won't bring much advantage when there is low redundancy.* ***Answer***: We performed experiments using AP Verifier and Hassel in C to compute port-to-port reachability sets for the Purdue dataset. Since the Purdue dataset does not have forwarding tables, the intersection of ACL predicates along the shortest path (in hop count) between two ports is computed. AP Verifier used 1-2 μs, on the average, for each pair of source-destination ports and is 2-3 orders of magnitude faster than Hassel in C (see Appendix). Comparing with Tables III(a) and (b), it is noteworthy that computing the intersection of ACL predicates is 2-3 orders of magnitude faster than computing the intersection of forwarding predicates.

VIII. Conclusion

We present a new formal method for a new tool, Atomic Predicates (AP) Verifier, which is much more time and space efficient than existing tools. We evaluated the performance of AP Verifier using forwarding tables and ACLs from three large real networks. The sizes of atomic predicate sets of these networks are surprisingly small. This outcome indicates that there exist large amounts of redundancy in the forwarding and ACL rules of real networks. By encoding the network state in terms of atomic predicates, such redundancy is eliminated.

The use of atomic predicates dramatically speeds up computation of reachability trees from ports. On the average, AP Verifier is 3 orders of magnitude faster than Hassel in C. It also uses 2 to 3 orders of magnitude less memory than Hassel in C for computing and storing reachability trees from ports.

Real networks are subject to dynamic state changes over time as a result of rule insertion and deletion by protocols and operators, failure and recovery of links and boxes, etc. AP Verifier includes algorithms to process such events and check compliance of network policies and properties in real time.

In particular, atomic predicates are not affected by link status (up or down). Thus while existing tools used several seconds of time to verify reachability compliance of a link up/down event, AP Verifier's compliance verification times are 4 to 5 orders of magnitude smaller.

Lastly, reachability properties of networks are affected by middle boxes that modify packets, e.g., NAT, MPLS, IPsec, etc. Our work on modeling and analysis of such "packet transformers" is presented in a related paper under preparation.

Acknowlegment. This work was sponsored by National Science Foundation grant CNS-1214239. We thank the anonymous reviewers for their constructive comments.

Appendix

A. Proof of Theorem 1

To prove Theorem 1, we first define equivalence classes of packets with respect to (w.r.t.) a given set $\mathcal{P}$ of predicates. We then prove Lemmas 1 and 2. *Theorem 1 follows directly from Lemmas 1 and 2.*

For a predicate P and a packet pkt, the *indicator function* $I_P(pkt)$ is defined as follows:

$$I_P(pkt) = \begin{cases} 1 & P \text{ evaluates to true for } pkt, \\ 0 & \text{otherwise.} \end{cases}$$

Given a set $\mathcal{P}$ of predicates, two packets, pkt_1 and pkt_2 are *equivalent* w.r.t. $\mathcal{P}$ if and only if $I_P(pkt_1) = I_P(pkt_2), \forall P \in \mathcal{P}$.

The packet equivalence relation partitions the set of all packets into *equivalence classes*, $\{C_1, \ldots, C_n\}$. That is, for every pair of packets, pkt_1 and pkt_2, they are in the same C_i, for $i \in \{1, \ldots, n\}$, if and only if they are equivalent. We can also define the indicator function on equivalence classes: $I_P(C_i) = I_P(pkt), \forall pkt \in C_i$, where $i \in \{1, \ldots, n\}$, and $P \in \mathcal{P}$.

Lemma 1. Given a set $\mathcal{P}$ of predicates, the predicates that specify $\{C_1, \ldots, C_n\}$ satisfy the first four properties in Definition 1.

Proof: We prove the four properties one by one using set notation. By the definition of equivalence classes, $C_i \neq \emptyset$, $\forall i \in \{1, \ldots, k\}$ so Property 1 is satisfied. The equivalence classes partition the set of all packets; thus the disjunction of all predicates is $true$ and Property 2 is satisfied. A packet cannot belong to two equivalence classes; therefore, the conjunction of two different predicates is $false$ and Property 3 is satisfied.

To prove Property 4, consider an arbitrary predicate $P \in \mathcal{P}$. Let the packet set specified by P be $\{pkt \mid I_P(pkt) = 1\}$. We prove Property 4 by proving that a packet pkt' is in $\{pkt \mid I_P(pkt) = 1\}$ *if and only if* packet pkt' is in $\cup_{I_P(C_i)=1} C_i$.

If part: Consider a packet $pkt' \in \cup_{I_P(C_i)=1} C_i$. Then for some i, $pkt' \in C_i$ and $I_P(C_i) = 1$. Thus $I_P(pkt') = I_P(C_i) = 1$. Hence $pkt' \in \{pkt \mid I_P(pkt) = 1\}$.

Only if part: Consider a packet $pkt' \in \{pkt \mid I_P(pkt) = 1\}$. Then $I_P(pkt') = 1$. Since $\{C_1, \ldots, C_n\}$ is a partition of the set of all packets, there exists an $i \in \{1, \ldots, n\}$ such that $pkt' \in C_i$. Thus $I_P(C_i) = I_P(pkt') = 1$. Hence $pkt' \in \cup_{I_P(C_i)=1} C_i$.

We have proved that $\{pkt | I_P(pkt) = 1\} = \cup_{I_P(C_i)=1} C_i$, which means that P is equal to the disjunction of a subset of predicates specifying equivalent classes (Property 4). ■

Lemma 2. For a set $\mathcal{P}$ of predicates, let $\{C_1, \ldots, C_n\}$ denote the equivalence classes w.r.t. $\mathcal{P}$. Consider any set of predicates $\{p_1, \ldots, p_m\}$ that satisfies the first four properties of Definition 1. Then for all $i \in \{1, \ldots, m\}$, there exists a unique $j \in \{1, \ldots, n\}$ such that $C_j \supseteq$ the packet set specified by p_i. This implies that $m \geq n$ which is minimum.

Proof: For any predicate $P \in \mathcal{P}$, from the assumption that $\{p_1, \ldots, p_m\}$ satisfies the fourth property of Definition 1, P can be represented by the disjunction of a subset of $\{p_1, \ldots, p_m\}$. Consider some $p_i \in \{p_1, \ldots, p_m\}$ and choose any two packets, pkt_1 and pkt_2, from the packet set specified by p_i. We will show that pkt_1 and pkt_2 are equivalent. There are two possibilities in the disjunction representation of P. First, p_i appears in the subset representing P, in which case, $I_P(pkt_1) = I_P(pkt_2) = 1$. Second, p_i does not appear in the subset representing P, in which case, $I_P(pkt_1) = I_P(pkt_2) = 0$. Therefore, $I_P(pkt_1) = I_P(pkt_2), \forall P \in \mathcal{P}$. Thus pkt_1 and pkt_2 are equivalent w.r.t. $\mathcal{P}$, and $pkt_1, pkt_2 \in C_j$ for some $j \in \{1, \ldots, n\}$. Thus $C_j \supseteq$ the packet set specified by $p_i \in \{p_1, \ldots, p_m\}$. ■

B. Proof of Theorem 2

Proof: Theorem 1 states that the atomic predicates for $\mathcal{P}_1 \cup \mathcal{P}_2$ specify the set of equivalence classes w.r.t. $\mathcal{P}_1 \cup \mathcal{P}_2$. We prove Theorem 2 by showing that $a_1, \ldots, a_k$ from formula (3) specify equivalence classes w.r.t. $\mathcal{P}_1 \cup \mathcal{P}_2$. That is, for any two packets, pkt_1 and pkt_2, pkt_1 is equivalent to pkt_2 w.r.t. $\mathcal{P}_1 \cup \mathcal{P}_2$ *if and only if* there exists $i \in \{1, \ldots, k\}$ such that pkt_1 and pkt_2 belong to the packet set specified by a_i.

If part: Assume that there exists $i \in \{1, \ldots, k\}$ such that pkt_1, pkt_2 belong to the packet set specified by a_i. Then there exist $i_1 \in \{1, \ldots, l\}$ and $i_2 \in \{1, \ldots, m\}$ such that $a_i = b_{i_1} \wedge d_{i_2}$. Thus pkt_1, pkt_2 belong to the packet set specified by b_{i_1} and to the packet set specified by d_{i_2}. From Theorem 1, b_{i_1} and d_{i_2} each specifies an equivalence class w.r.t. $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Thus, $\forall P \in \mathcal{P}_1$, $I_P(pkt_1) = I_P(pkt_2)$, and $\forall P \in \mathcal{P}_2$, $I_P(pkt_1) = I_P(pkt_2)$. Therefore, $I_P(pkt_1) = I_P(pkt_2), \forall P \in \mathcal{P}_1 \cup \mathcal{P}_2$. That is, pkt_1 and pkt_2 are equivalent w.r.t. $\mathcal{P}_1 \cup \mathcal{P}_2$.

Only if part: Assume that pkt_1 and pkt_2 are equivalent w.r.t. to $\mathcal{P}_1 \cup \mathcal{P}_2$. Then we have $I_P(pkt_1) = I_P(pkt_2), \forall P \in \mathcal{P}_1$, and $I_P(pkt_1) = I_P(pkt_2), \forall P \in \mathcal{P}_2$. Thus, pkt_1, pkt_2 are equivalent w.r.t. $\mathcal{P}_1$, and $pkt_1, pkt_2 \in$ the equivalence class specified by b_{i_1}, for some $i_1 \in \{1, \ldots, l\}$. Similarly, we can show that pkt_1, $pkt_2 \in$ the equivalence class specified by d_{i_2}, for some $i_2 \in \{1, \ldots, m\}$. Since $pkt_1, pkt_2 \in$ the equivalence classes specified by b_{i_1} and d_{i_2} respectively, and $b_{i_1} \wedge d_{i_2} \neq false$, there exists $i \in \{1, \ldots, k\}$ such that $a_i = b_{i_1} \wedge d_{i_2}$, and $pkt_1, pkt_2 \in$ the packet set specified by a_i.

Consequently, the set $\{a_1, \ldots, a_k\}$ specifies the set of equivalence classes of packets w.r.t. $\mathcal{P}_1 \cup \mathcal{P}_2$. Thus $\mathcal{A}(\mathcal{P}_1 \cup \mathcal{P}_2) = \{a_1, \ldots, a_k\}$. ■

C. Advantages of BDD over Other Data Structures

We selected BDD as the data structure for representing a predicate (set of packets) after performing a comparative study of BDD versus three other data structures, namely: Packet Set (PS) representation using a set of tuples [18], firewall decision diagram (FDD) [7], and wildcard expression [11]. In what follows, we explain why none of the other three data structures has all of the desirable properties of BDD and is as efficient as BDD.

Unique representation: Consider a given predicate representing a set of packets. It has been proved that its representation as a reduced ordered BDD [5] or as a reduced FDD [7] is unique. However such a predicate may have multiple PS representations or multiple wildcard expressions. It is nontrivial to check that different PS representations, or different wildcard expressions, are equivalent and thus represent the same predicate.

Logical operations: Computing the negation of a BDD or FDD is easy; it is done by swapping the two terminal nodes in the BDD or FDD. However, computing the negation of a PS representation or wildcard expression is nontrivial. The negation of a PS representation (or a wildcard representation) might result in more tuples (or wildcard strings) than the original representation. For all four data structures, conjunction (also disjunction) requires time proportional to the product of the operand sizes.

Representation size for an ACL rule: Consider ACL rules in which the allowed values of each header field are specified by a suffix, prefix, or an interval. This constraint is satisfied by every ACL rule in the datasets we have (from Stanford, Purdue, and Wisconsin) and is likely satisfied by the vast majority of ACL rules in practical use.

Both PS and FDD use intervals to represent allowed values of each field in a packet header. A prefix is a special type of interval. However, a suffix is not. If a field is specified by a suffix, an exponential number of intervals are required to represent the suffix in the worst case. For example, consider a 32-bit IP address field. A single-bit suffix (the worst case) requires 2^{31} intervals to represent. The size of a PS (also FDD) representation of an ACL rule can be measured by the number of intervals used. Consider a packet header with k fields and the ith field has h_i bits, $i = 1, \ldots, k$. In the worst case, the size of a PS (also FDD) representation is $O(2^{h_1} + 2^{h_2} + \cdots + 2^{h_k})$.

The size of a wildcard expression can be measured by the number of wildcard strings used in the expression. An interval can be represented by multiple wildcard expressions. For example, consider the interval from 001 to 110 of a 3 bit field. This interval has 6 numbers and can be represented by multiple wildcard expressions, including:

1. 001, 01*, 10*, 110
2. 0*1, *10, 10*
3. *01, 1*0, 01*
4. *01, *10, 011, 100

It is easy to see that each of the wildcard expressions above representing the interval from 001 to 110 has at least 3 wildcard strings. In general, for a field that has h_i bits, there exists an interval that requires at least h_i wildcard strings in each of its expressions. Therefore, there exists an ACL rule that requires $h_1 h_2 \cdots h_k$ wildcard strings in its expression.

The size of the BDD graph representing an ACL rule is measured by the number of nodes in the graph. For the same ACL rule, ordering variables differently may result in BDD graphs of different sizes. Let h denote the number of header bits. The following theorem shows that the size of the BDD for an ACL rule is $2 + 2h$ in the worst case, where h is the number of header bits.

Theorem 3. If the length of a packet header is h bits, and an ACL rule specifies each header field by an interval, a prefix or a suffix, then the size of the BDD representation of an ACL rule is $\leq 2 + 2h$.

Proof: The header's bit sequence is partitioned into fields. Let h_i be the number of bits of the ith field, $i = 1, 2, \ldots, k$. $\sum_{i=1}^{k} h_i = h$. Each variable in the BDD represents one bit in the packet header. In the BDD representation, each header field in a rule is specified by a BDD subgraph. The BDD graph of the rule is obtained by merging the subgraphs representing its fields. A high level representation of the BDD graph for an ACL rule is shown in Figures 5a and 5b. A circle labeled by $field_i$ indicates a BDD subgraph representing the ith field. An edge exiting the circle is labeled $true$ if the corresponding subgraph is evaluated to $true$. An edge exiting the circle is labeled $false$ if the corresponding subgraph is evaluated to $false$. For a rule that has $allow$ action, its BDD graph evaluates to $true$ if all subgraphs evaluate to true. For a rule that has $deny$ action, its BDD graph evaluates to $false$ if all subgraphs evaluate to $true$.

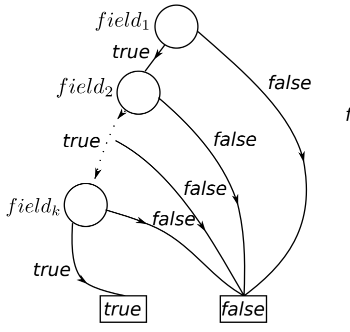
For the ACL rule that allows all packets, its BDD representation has only one node, the terminal node $true$. For the rule that denies all packets, its BDD representation has only one node, the terminal node $false$.

For a nontrivial rule, there may be one or more non-terminal nodes in each circle. Let N_i be the number of non-terminal nodes in the circle for the ith field, $i = 1, \ldots, k$. Then the total number of nodes in the BDD representation is $2 + \sum_{i=1}^{k} N_i$, where 2 counts the two terminal nodes.

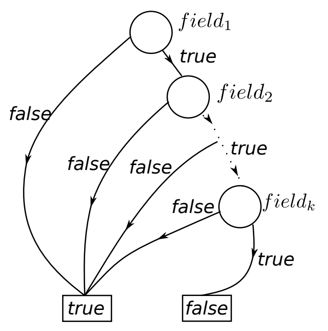
We next derive upper bounds of N_i. If the ith field is specified by a prefix or a suffix, it is straightforward to represent the field using a BDD. (See Figures 6a and 6b for 4-bit examples.) The length of the longest possible prefix or suffix is h_i for field i. Thus we have $N_i \leq h_i$ for these two cases.

If the ith field is specified by an interval, Gupta [9] shows that it can be represented by at most $2h_i - 2$ prefixes. All of these prefixes can be represented by a non-reduced binary decision diagram of at most $2h_i$ nodes (see Figure 6c for a 4-bit example). So we have $N_i \leq 2h_i$.

Therefore, we have the worst-case bound, $2 + \sum_{i=1}^{k} N_i \leq 2 + 2\sum_{i=1}^{k} h_i = 2 + 2h$. ■



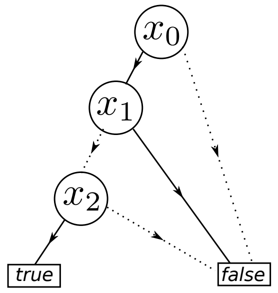
(a) $action = allow$.



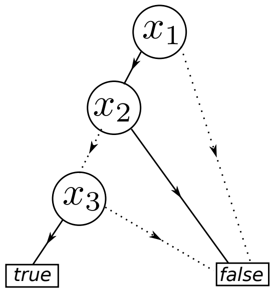
(b) $action = deny$.

Fig. 5: BDD representation of an ACL.

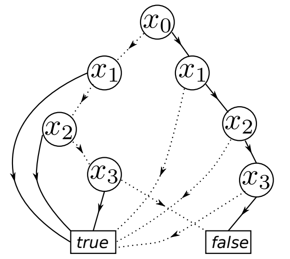
Based on the above analysis, we chose BDD as the data structure for AP Verifier since BDD representations of predicates are unique and efficient.



(a) BDD subgraph for prefix 101∗.



(b) BDD subgraph for suffix ∗101.



(c) BDD subgraph for the interval from 0001 to 1110.

Fig. 6: BDD subgraphs representing a prefix, a suffix and an interval. The field has 4 bits represented by variables x_0, x_1, x_2, x_3. A dotted edge denotes an assignment to $false$ and a solid edge denotes an assignment to $true$.

When numerous rules are grouped into an ACL or a forwarding table, we are interested in the growth of the number of BDD nodes used to represent the ACL/table as the number of rules increases. From several datasets ([11], [16], [2], [4]), we observed that the number of BDD nodes used to represent an ACL or a forwarding table increases approximately linearly with the number, m, of rules in the ACL/table up to a maximum and then decreases as m increases further (see Figures 7 and 8).

D. Performance Evaluation using Purdue Dataset

We used the Purdue dataset to compare the performance of AP Verifier and Hassel in C. We started from a network topology including just the core routers in the Purdue dataset, and gradually increased the network size by including neighbors of middle boxes already chosen. In each network, we selected the shortest path that go through core routers for each pair of middle boxes. We measured the times to compute intersections of ACLs along each path using AP Verifier and Hassel in C. The results are summarized in Table VI. We can see that on the average, AP Verifier takes about 1-2 μs to compute intersections of ACLs along a path and is about 2-3 orders of magnitude faster than Hassel in C.

E. Rule Update Algorithms

When a rule is inserted into or deleted from an ACL (or a forwarding table), it may change a port predicate and the set of atomic predicates may change also. To update a port's reachability tree being used for reachability compliance check, AP Verifier running on one processor core performs these steps: (i) It checks if a port predicate is changed by the rule update; if so, it computes a new predicate for the port. (ii) It updates the reachability tree using the new predicate, if any. (iii) It forks a process which runs on a second core to update the set of atomic predicates. Steps (ii) and (iii) occur concurrently.

To perform step (i), AP Verifier uses different approaches for ACL and forwarding table rule updates.

Port Predicate Change due to an ACL Rule Update

When an existing rule is deleted from or a new rule is added to an ACL, the ACL's predicate may change. A naive way is to run Algorithm 1 from the first rule to recompute the predicate. However, AP Verifier stores intermediate results to avoid recomputation from the first rule. When Algorithm 1 is run for the first time, predicates $allowed_i$ and $denied_i$ are

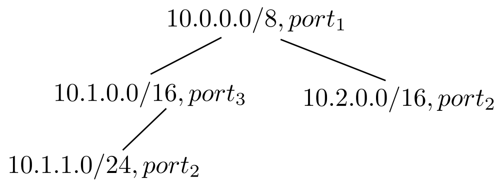


Fig. 9: Forwarding rules organized as a tree

stored for the ith rule, $i = 1, 2, \ldots, m$:

$$\begin{aligned}&G_1, action_1, allowed_1, denied_1\\&G_2, action_2, allowed_2, denied_2\\&\ldots\ldots\\&G_m, action_m, allowed_m, denied_m\end{aligned}$$

For the ith rule, $allowed_i$ is the predicate specifying the set of packets that are allowed by rules from 1 to i; $denied_i$ is the predicate specifying the set of packets that are denied by rules from 1 to i. In particular, $allowed_m$ specifies the set of packets that are allowed by the ACL.

When the ith rule is updated, AP Verifier starts from the ith rule and runs Algorithm 1 using $allowed_{i-1}$ and $denied_{i-1}$ to recompute the predicate for the ACL.

Port Predicate Change due to a Forwarding Rule Update

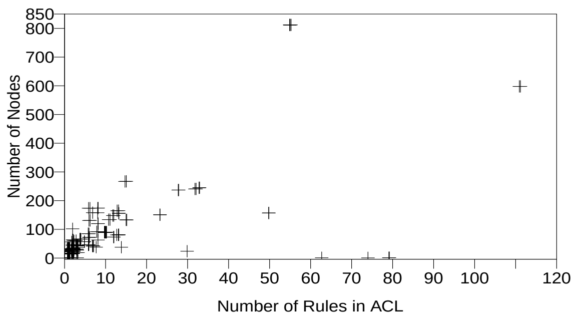
To accommodate forwarding rule updates, AP Verifier organizes rules of a forwarding table into a forest and computes forwarding port predicates using trees in the forest. Instead of Algorithm 2, AP Verifier first sorts rules in the table in *descending order of prefix length* and then uses Algorithm 5 to build trees in the forest. See Figure 9 for an example of a tree.

Algorithm 5 Organize Forwarding Rules as a Forest

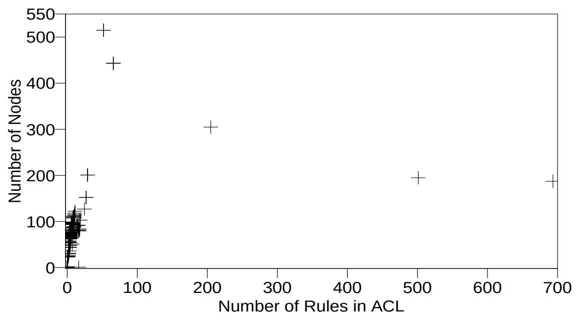
Input: A sorted forwarding table
Output: A forest of rules

```
1: for $i = 1$ to $m$ do
2:     for $j = i + 1$ to $m$ do
3:         if rule $j$'s prefix contains rule $i$'s prefix then
4:             make rule $j$ the father of rule $i$
5:             break
6:         end if
7:     end for
8: end for
```

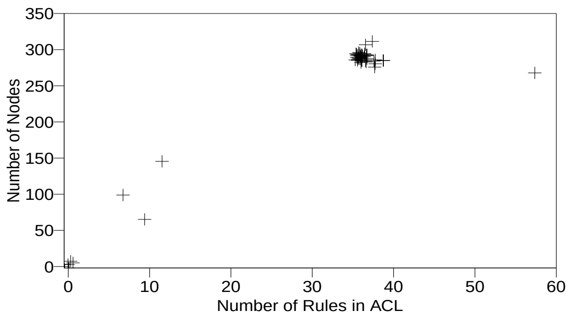
Consider a forwarding table that has m rules and k ports indexed by $\{1, \ldots, k\}$. AP Verifier stores two predicates for



(a) Stanford dataset.

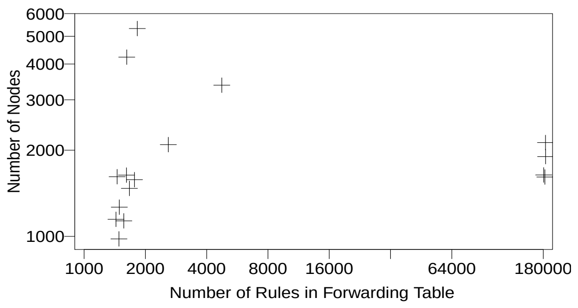


(b) Purdue dataset.

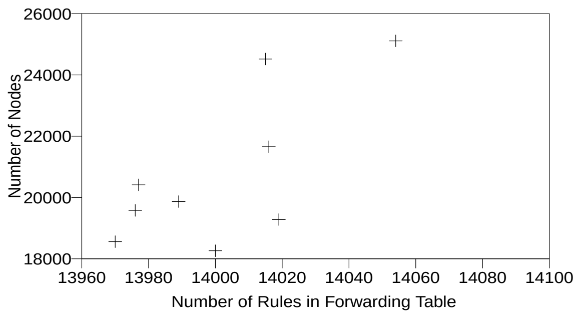


(c) Wisconsin dataset.

Fig. 7: Number of BDD nodes to represent an ACL.



(a) Stanford network.



(b) Internet2.

Fig. 8: Total number of BDD nodes to represent forwarding predicates of a forwarding table.

	Hassel in C			AP Verifier		
Network Size	Average (ms)	Median (ms)	Maximum (ms)	Average (ms)	Median (ms)	Maximum (ms)
74	0.65	0.68	0.90	0.0027	0.00049	0.038
161	0.67	0.69	1.04	0.0012	0.00019	0.012
224	0.93	0.96	1.35	0.00056	0.00012	0.042
351	0.99	0.96	2.89	0.00080	0.00012	0.034
519	0.83	0.82	1.84	0.00082	0.00012	0.040
603	0.80	0.80	1.57	0.0020	0.00086	0.056
647	0.30	0.32	0.80	0.0014	0.0022	0.038
880	1.06	1.08	1.93	0.0028	0.00090	0.053
904	0.82	0.79	1.53	0.00090	0.00017	0.0049
941	0.96	0.94	1.50	0.00071	0.00017	0.013

TABLE VI: Time to compute intersections of ACLs along a path in Purdue dataset.

each rule:

$$Pre_1, L_1, port_1, R_1$$
$$Pre_2, L_2, port_2, R_2$$
$$\cdots\cdots$$
$$Pre_m, L_m, port_m, R_m$$

where, for $i = 1, 2, \ldots, m$, predicate Pre_i represents the prefix of the ith rule, L_i is the length of the prefix, $port_i \in \{1, \ldots, k\}$ is the port number, and R_i specifies the set of packets that are actually forwarded by the rule. Predicate R_i is computed using the following equation:

$$R_i = Pre_i \wedge (\vee_{\text{rule } j \text{ is a child of rule } i} \neg Pre_j) \qquad (6)$$

The forwarding port predicate of port x, $x = 1, 2, \ldots, k$, is

$$P_x = \vee_{\text{rule } i \text{ forwards to port } x} R_i \qquad (7)$$

In what follows, we first discuss rule deletion and then rule addition. In each case, at most two port predicates are affected.

If rule i is deleted from the forwarding table, we update its tree of rules as follows. We find rule i's father, rule j, and make rule i's children into rule j's children. Predicate R_j of rule j is updated as

$$R_j \leftarrow R_j \vee R_i$$

Assume that rule i forwards packets to port x, and rule j forwards packets to port y. Then the forwarding port predicates P_x and P_y are updated using the following equation:

$$P_x \leftarrow P_x \wedge \neg R_i$$
$$P_y \leftarrow P_y \vee R_i$$

If a new rule, i, is added, AP Verifier first inserts it into the tree it belongs. Assume that rule i's father in the tree is rule j. R_i is computed using (6) and R_j is updated by

$$R_j \leftarrow R_j \wedge \neg Pre_i$$

Assume that rule i forwards packets to port x, and rule j forwards packets to port y. Forwarding port predicates P_x and P_y are updated by:

$$P_x \leftarrow P_x \vee R_i$$
$$P_y \leftarrow P_y \wedge \neg R_i$$

If no port predicate changes after a rule update, AP Verifier does not need to update the reachability tree or atomic predicates. Otherwise, AP Verifier needs to run the following steps.

Update Atomic Predicates

When port predicates are changed, the corresponding set of atomic predicates needs to be updated. Updating atomic predicates can be handled by two basic cases: add a new predicate and remove an old predicate.

Add A Predicate. It is straightforward to update the set of atomic predicates when a new predicate is added. A new set of atomic predicates will have to be computed by applying formula (3) to the existing set of atomic predicates and the newly added predicate. The mapping from predicates to sets of atomic predicates is then updated.

Remove A Predicate. If a predicate is removed, the old set of atomic predicates still satisfy the first 4 properties in Definition 1. Thus AP Verifier can still use it to represent the remaining predicates. However, the old set of atomic predicates is not necessary minimum w.r.t. the remaining predicates. To make it minimum, AP Verifier uses the following function to map each atomic predicate to a set of predicates. (AP Verifier can fork a new process which runs on another core to perform minimization.)

Definition 2 (*Reference Function*)**.** Let $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ be a set of predicates, and $\mathcal{A}(\mathcal{P}) = \{p_1, p_2, \ldots, p_k\}$ be the set of atomic predicates. The reference function ref is defined on $\mathcal{A}(\mathcal{P})$:

$$ref(p_i) = \{P_j | i \in S(P_j), j = 1, \ldots, N\}, i = 1, 2, \ldots, k \quad (8)$$

Function ref stores the information of how atomic predicates are used to represent predicates and it can be computed when the set of atomic predicates is computed.

Algorithm 6 shows how to minimize the set of atomic predicates after a predicate is removed. $\mathcal{P}$ is a set of predicates, and $\mathcal{A}(\mathcal{P})$ is its set of atomic predicates. Assume that P_j is going to be removed.

The algorithm first updates the ref function. Then it finds every pair of old atomic predicates p, p' such that $ref(p) = ref(p')$, and merges p and p' to be one new atomic predicate because $ref(p) = ref(p')$ means that the two atomic predicates have the same usage for the changed set of predicates. Note that for each old atomic predicate p, there is at most one different old atomic predicate p' such that $ref(p) = ref(p')$.

Algorithm 6 Minimize the set of Atomic Predicates

```
Input: $\mathcal{P}, P_j, \mathcal{A}(\mathcal{P})$
Output: $\mathcal{A}(\mathcal{P} - \{P_j\})$
 1: $tmp \leftarrow \mathcal{A}(P)$
 2: for all $p \in \mathcal{A}(\mathcal{P})$ do
 3:     $ref(p) \leftarrow ref(p) - \{P_j\}$
 4: end for
 5: for all $p$ used to represent $P_j$ do
 6:     if there exists $p' \in \mathcal{A}(\mathcal{P}), p \neq p'$ such that $ref(p) = ref(p')$ then
 7:         $tmp \leftarrow tmp - \{p, p'\}$
 8:         $tmp \leftarrow tmp \cup \{p \vee p'\}$
 9:     end if
10: end for
11: $\mathcal{A}(\mathcal{P} - \{P_j\}) \leftarrow tmp$
```

Update Reachability Tree

AP Verifier can first update the set of atomic predicates and compute the reachability tree directly. However, updating atomic predicates takes 10 ms on the average. For quick response to a rule update, AP Verifier forks a process which runs on another core to update the set of atomic predicates while the process running on the first core builds a temporary tree for quick response as follows.

AP Verifier updates the reachability tree using changed port predicates. The updated tree is correct and can be used for compliance check but is intended to be temporary. In the following description, we assume a forwarding rule update; the procedure is similar for an ACL rule update.

In a temporary reachability tree, each node uses a set of predicates $\mathcal{T}$ to store new port predicates that are unresolved. For example, in the top part of Figure 10, the port has forwarding port predicate F_{port} and ACL predicate A_{port}. Let S_F and S_A represent the set of packets that can reach the entrance of the port. Then $S_F \cap S(F_{port})$ and $S_A \cap S(A_{port})$

$$\begin{tikzcd} {} \arrow[r, "{S_F, S_A}"] & \underset{F_{port}, A_{port}}{port} \arrow[r, "{S_F \cap S(F_{port}), S_A \cap S(A_{port})}"] & {} \end{tikzcd}$$

$$\begin{tikzcd} {} \arrow[r, "{S_F, S_A, \{\}}"] & \underset{F'_{port}, A_{port}}{port} \arrow[r, "{S_F, S_A \cap S(A_{port}), \{F'_{port}\}}"] & {} \end{tikzcd}$$

Fig. 10: Add changed port predicates to $\mathcal{T}$

represent the set of packets that can pass through the port.

After a forwarding rule update, F_{port} is changed to F'_{port} which is unresolved (see bottom part of Figure 10). Hence AP Verifier stores F'_{port} in $\mathcal{T}$. The packet set that can pass through the port is specified by the following predicate

$$(\vee_{j \in S(A_{port}) \cap S_A} a_j) \wedge (\vee_{i \in S_F} f_i) \wedge F'_{port}$$

After multiple rule updates, set $\mathcal{T}$ can have multiple unresolved predicates.

AP Verifier locates all nodes in the tree that are affected by a rule update. For each affected node: AP Verifier copies the set S_F from its father node, and adds the new forwarding predicate to set $\mathcal{T}$. Then AP Verifier removes all its descendant nodes, and performs a depth-first search from the node to extend the tree. In the depth-first search, predicates in set $\mathcal{T}$ are propagated to each of the new descendant nodes.

If rule updates arrive in rapid succession, AP Verifier can keep on updating the temporary reachability tree correctly.

Convert to A Normal Reachability Tree

The process running on the second core can compute the updated set of atomic predicates in 10 ms on the average for one rule update. In our experiments, about 50% of rule updates did not change atomic predicates.

If the updated set of atomic predicates is unchanged *and* there are three[6] or fewer unresolved predicates, the temporary tree is converted to a "normal" one by the process running on the first core. To do this, AP Verifier traverses the temporary reachability tree. In each node, each unresolved predicate is replaced by its set of atomic predicate identifiers. If the unresolved predicate is for forwarding, its set of atomic predicate identifiers is intersected with S_F; if the unresolved predicate is for ACL, its set of atomic predicate identifiers is intersected with S_A. This step is summarized as Algorithm 7.

Algorithm 7 Convert to Normal Reachability Tree

```
Input: A temporary reachability tree
Output: A normal reachability tree
 1: for each node $n$ in the temporary reachability tree do
 2:     access node $n$'s $S_F, S_A, \mathcal{T}$
 3:     for $P$ in $\mathcal{T}$ do
 4:         if $P$ is a forwarding predicate then
 5:             $S_F \leftarrow S_F \cap S(P)$
 6:         else
 7:             $S_A \leftarrow S_A \cap S(P)$
 8:         end if
 9:     end for
10:     remove $\mathcal{T}$
11: end for
```

Otherwise, if the updated set of atomic predicates is changed *or* there are more than three unresolved predicates, the process computes a new reachability tree directly from the updated set of atomic predicates. It can do so in less than 1 ms most of the time (see Tables III(c) and III(d); note that loop detection for a port is performed by computing its reachability tree).

[6]This is a configurable parameter value.

References

[1] Header Space Library and NetPlumber. In *https://bitbucket.org/peymank/hassel-public/*.

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

[3] 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 *Proceedings of IEEE ICNP*, Princeton, New Jersey, 2009.

[4] T. Benson, A. Akella, and D. A. Maltz. Unraveling the Complexity of Network Management. In *Proceedings of USENIX NSDI*, Boston, Massachusetts, 2009.

[5] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[6] E.A. Emerson. Temporal and Modal Logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B.* MIT Press, 1990.

[7] M. G. Gouda and A. X. Liu. Firewall Design: Consistency, Completeness, and Compactness. In *Proc. of IEEE ICDCS*, Tokyo, Japan, 2004.

[8] M. G. Gouda, A. X. Liu, and M. Jafry. Verification of Distributed Firewalls. In *Proc. of IEEE GLOBECOM*, New Orleans, Louisiana, 2008.

[9] P. Gupta. *Algorithms for Routing Lookups and Packet Classification*. PhD thesis, Stanford University, 2000. Advisor: N. W. McKeown. Co-advisor: B. Prabhakar.

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

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

[12] A. R. Khakpour and A. X. Liu. Quantifying and Querying Network Reachability. In *Proc. of IEEE ICDCS*, Genoa, Italy, 2010.

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

[14] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *Proceedings of ACM SIGCOMM*, Toronto, Ontario, Canada, 2011.

[15] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, Washington, 2003.

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

[17] Vahidi, A. JDD, a pure Java BDD and Z-BDD library. In *http://javaddlib.sourceforge.net/jdd/*. 2004.

[18] E. Wong. Validating Network Security Policies via Static Analysis of Router ACL Configuration. Master's thesis, Naval Postgraduate School, 2006. Advisor: G. G. Xie.

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