Real-Time Verification of Network Properties Using Atomic Predicates

Hongkun Yang, *Student Member, IEEE*, and Simon S. Lam, *Fellow, IEEE, Fellow, ACM*

***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 Header Space and NetPlumber using datasets from the real networks.**

***Index Terms*—Automated tools, formal methods, network management, network policies and properties, protocol verification, reachability computation.**

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 and 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 and MPLS. Some boxes also modify packets, e.g., NAT. (We use "box" in this paper as a generic name for networking devices, including routers, switches and middle boxes.) In a study of large-scale Internet services [16], operator error was found to be the largest single cause of failures with configuration errors being the largest category of operator errors.

Manuscript received January 31, 2014; revised July 07, 2014, November 12, 2014; accepted January 02, 2015; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Bejerano. Date of publication March 12, 2015; date of current version April 14, 2016. This work was supported by the National Science Foundation under Grant CNS-1214239. An abbreviated version of this paper appeared in Proceedings of the IEEE International Conference on Network Protocols, October 2013.

The authors are with the Department of Computer Science, The University of Texas at Austin, Austin, TX 78712 USA (e-mail: yanghk@cs.utexas.edu; lam@cs.utexas.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2015.2398197

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.* [20]. 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], [13], [15], [12], [14], [11]. 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, and reachability via waypoints.

The network state in the data plane is determined by the forwarding and ACL rules in the network's boxes. Forwarding tables and ACLs are packet filters. They can be parsed and represented by predicates that guard input and output ports of boxes. The variables of such a predicate represent packet header bits. 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 [17], 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 computed very quickly and their sizes are surprisingly small. For example, the Stanford network [12] 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 also much more *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* [14], [11]. More specifically, if a "verifier" is placed in the communication path between a SDN's central controller and its switches, 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 [11]. 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 box. We describe how port predicates of each box are computed from rules in its forwarding table and ACLs. In Section III, we show that binary decision diagrams (BDDs) have the desirable properties for representing packet sets. Experimental results show that BDD is a highly efficient data structure for reachability verification, regardless of whether atomic predicates are used for further optimization [see Table IV(a)–(d)]. In Section IV, 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 also present statistics of three real networks [1], [17], [2] including the sizes and computation times of their atomic predicate sets. In Section V, we present algorithms for computing reachability and verifying a number of network properties. We present computation times and storage costs comparing AP Verifier with Hassel in C (the fast version used in Header Space and NetPlumber [12], [11]). In Section VI, we present algorithms for processing network state changes due to link up/down and rule insertion/deletion events as well as for checking reachability compliance in real time. We present computation times comparing AP Verifier with NetPlumber [11]. In Section VII, we discuss the impact of ACLs. In Section VIII, we discuss related work and conclude in Section IX.

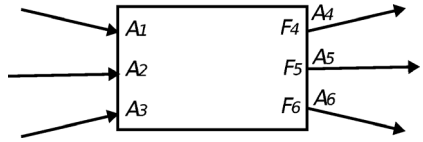


Fig. 1. Example of a box. $A_1, \ldots, A_6$ are ACL predicates and F_4, F_5, F_6 are forwarding predicates.

II. Network Model

We model a packet network as a directed graph of boxes (which may be routers, switches, or firewalls). Each box has a forwarding table as well as input and output ports guarded by access control lists (ACLs). Our model of a box is shown in Fig. 1. Observe that each input port is guarded by a predicate specified by the port's ACL. Each output port is guarded by a predicate specified by the forwarding table followed by a predicate specified by the port's ACL.

Each packet has a header of h bits. The header bits are partitioned into multiple fields. The three networks analyzed in this paper are all IP networks. Note that *our model of packet headers is general and not limited to IP headers*; for example, it can be used for any layer-2 or layer-3 protocol header.

A predicate is a Boolean formula where each variable represents one bit in the packet header. A predicate represents a set of packets for which the predicate evaluates to true. Predicate *false* specifies the empty set. *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], [18].

In what follows, we present algorithms for computing predicates that guard the ports of a box from the box's ACLs and forwarding table. We refer to these predicates as *port predicates*.

Access control lists. An ACL is a list of rules. Each ACL rule is specified by a condition and an action. For each packet, the rules are considered in sequential order. Whether the packet can pass through the filter is determined by the action of the *first rule* whose condition is satisfied by the packet's header.

To compute the predicate for an ACL that guards a port, the condition in each rule in the ACL is first converted into a predicate in bit variables (represented by a BDD). Let G_i be the predicate specifying the condition of the ith rule. An ACL with m rules is represented by the following list:

$$G_1, \text{action}_1$$
$$G_2, \text{action}_2$$
$$\ldots$$
$$G_m, \text{action}_m$$

where action_i is either *allow* or *deny*. We use Algorithm 1 to compute the port predicate that specifies the packet set allowed by the ACL.

Algorithm 1 Converting an ACL to a port predicate

Input: An ACL ($G_i, action_i$ for $i = 1, \ldots, m$)

Output: A predicate for the ACL

```
1: allowed ← false, denied ← false
2: for i = 1 to m do
3:   if action_i = deny then
4:     denied ← denied ∨ G_i
5:   else
6:     allowed ← allowed ∨ (G_i ∧ ¬denied)
7:   end if
8: end for
9: return allowed
```

Forwarding table. The forwarding table in a 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. 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.

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 Section III). 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 $2 + 2h$. Algorithm 1 remains the same for such rules.

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:

$$\begin{aligned} &\text{Pre}_1, L_1, \text{port}_1 \\ &\text{Pre}_2, L_2, \text{port}_2 \\ &\ldots \\ &\text{Pre}_m, L_m, \text{port}_m \end{aligned}$$

where L_i, $i \in \{1, \ldots, m\}$, are prefix lengths such that $L_1 \geq L_2 \geq \ldots L_m$; Pre_i denotes the predicate representing a prefix with length L_i; and $\text{port}_i \in \{1, \ldots, k\}$. Then we use Algorithm 2 to convert the sorted forwarding table to a list of forwarding 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 $\{1, \ldots, k\}$

Output: A list of predicates $\{P_1, \ldots, P_k\}$

```
1: for j = 1 to k do
2:   P_j ← false
3: end for
4: fwd ← false
5: for i = 1 to m do
6:   P_port_i ← P_port_i ∨ (Pre_i ∧ ¬fwd)
7:   fwd ← fwd ∨ Pre_i
8: end for
9: return {P_1, ..., P_k}
```

Virtual ports. A port in a 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.

III. How to Represent a Packet Set

The data structure for representing predicates specifying packet sets is crucial to both the time and space efficiency of any tool designed for reachability computation and verification. We selected binary decision diagram (BDD) as the data structure in AP Verifier after performing a comparative study of BDD versus three data structures proposed by other researchers, namely: Tuple representation (a set of tuples) [19], firewall decision diagram (FDD) [8], [13], and wildcard expression (a set of bit strings with wildcards) [12], [11]. No data structure is described for representing packet sets in the other related papers on reachability verification [20], [3], [15], [14].

In what follows, we explain why none of the other three data structures has all of the desirable properties of BDD.

Unique representation: Consider a predicate that specifies a set of packets. It has been proved that its representation as a reduced-ordered BDD [5] or as a reduced FDD [8] is unique. However, a predicate may have multiple Tuple representations or multiple wildcard expressions. It is nontrivial (time-consuming) to check that different Tuple representations, or different wildcard expressions, are equivalent and thus represent the same predicate.

Representation size for a rule: Consider an ACL rule in which the allowed values of each header field are specified by a prefix, suffix, or an interval. Both Tuple representation 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 has h bits, then a single-bit suffix is represented as a union of 2^{h-1} intervals, each of which represents a single packet. In the worst case, each field is a single-bit suffix. Let h_i denote the number of bits in header field i. In both Tuple and FDD representations, the total number of intervals is $2^{h_1-1} + 2^{h_2-1} + \cdots 2^{h_k-1}$.

An interval can be represented by multiple wildcard expressions each of which is a set of wildcard strings. In general, for header field i with h_i bits, there exists an interval that requires at least h_i wildcard strings in each of its expressions. Therefore, in the worst case, there exists a rule that requires $h_1 h_2 \cdots h_k$ wildcard strings in its expression.

The size of the BDD graph representing a rule is measured by the number of nodes in the graph. For an ACL rule, ordering variables differently may result in BDD graphs of different sizes. However, if the order of variables is the same as the order of bits in the packet header, then the number of nodes in the BDD graph for an ACL rule is $2 + 2h$ in the worst case, where h is the number of header bits. A proof of the following theorem is presented in Appendix A. Theorem 1 does not depend on the number of packet fields. The packet header constraint stated in the theorem is satisfied by every ACL rule in the datasets of real networks we have (datasets from Stanford, Purdue, and Wisconsin), and is likely satisfied by the vast majority of ACL rules in practical use.

Theorem 1: 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 number of nodes in the BDD graph representing an ACL rule is $\leq 2 + 2h$.

For forwarding rules specified by prefixes or suffixes, the number of nodes in the BDD graph representing a forwarding rule is $\leq 2 + n$, where n is the number of bits in the destination address field (cf. Fig. 9(a) and (b) in Appendix A for examples).

Logical operations: For all four data structures, conjunction (also disjunction) requires time proportional to the product of the operand sizes. 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 Tuple representation or wildcard expression is nontrivial. The negation of a Tuple representation (or a wildcard representation) might result in more tuples (or wildcard strings) than the original representation.

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

Representation size for ACL/table. 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. To evaluate such growth, we computed BDDs for the datasets of three real networks with network statistics shown in Table I. All 16 boxes in the Stanford dataset are routers. All nine boxes in the Internet2 dataset are routers. The 1646 boxes in the Purdue dataset consist of routers and switches.

We computed the number of BDD nodes used to represent an ACL versus the number of rules in the ACL. These results are shown in Figs. 2(a)–(b). For each forwarding table, we computed the total number of nodes in the BDDs representing all forwarding predicates guarding the output ports versus the number of rules in the table. These results are shown in Figs. 3(a)–(b).

TABLE I
STATISTICS OF THREE REAL NETWORKS

	Stanford	Internet2	Purdue
No. of 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

It is interesting to observe from these figures that increasing the number of rules in an ACL or a forwarding table does not always mean more BDD nodes. In three out of the four figures, the largest number of rules does not produce the largest number of BDD nodes. 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 1825 rules is represented by 5325 BDD nodes (maximum); the forwarding table with the most rules (184 908) is represented by only 1900 BDD nodes.

The computation times of the BDD for an ACL and all BDDs for a forwarding table are shown in Fig. 4 for the Stanford dataset.[1] These computation overheads are low (i.e., milliseconds for each ACL/table). Note that the computation time for a forwarding table includes the time to sort rules in the table in descending order of prefix length. It does not, however, include the preprocessing time used to reduce table size by the ORTC algorithm [6] because such preprocessing is needed for any data structure used to represent packet sets.

IV. ATOMIC PREDICATES

A. Basic Idea

Consider a set U of elements. A predicate P specifies a subset of elements. Predicate *true* specifies U. Predicate *false* specifies the empty set.

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

1) $p_i \neq$ false, $\forall i \in \{1, \ldots, k\}$.
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:

$$P = \bigvee_{i \in S(P)} p_i, \qquad \text{where } S(P) \subseteq \{1, \ldots, k\}. \tag{1}$$

5) k is the *minimum* number such that the set $\{p_1, \ldots, p_k\}$ satisfies the above four properties.

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

[1]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.

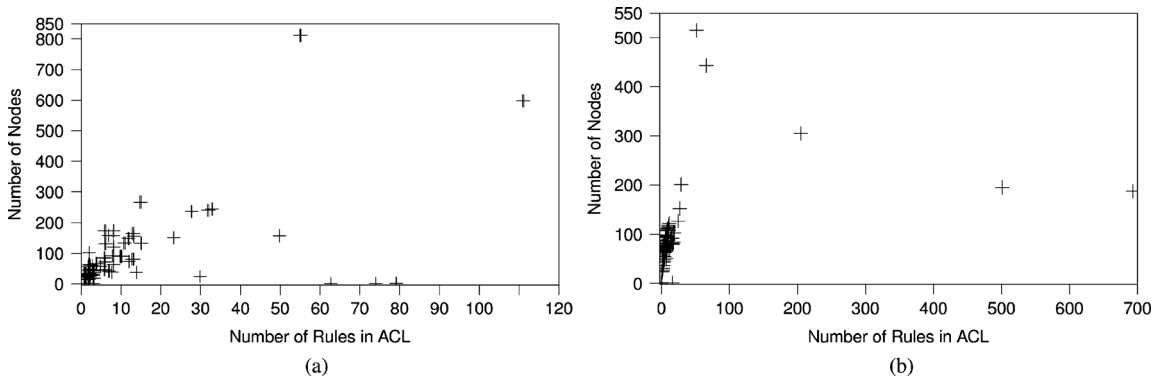


Fig. 2. Number of BDD nodes to represent an ACL. (a) Stanford dataset. (b) Purdue dataset.

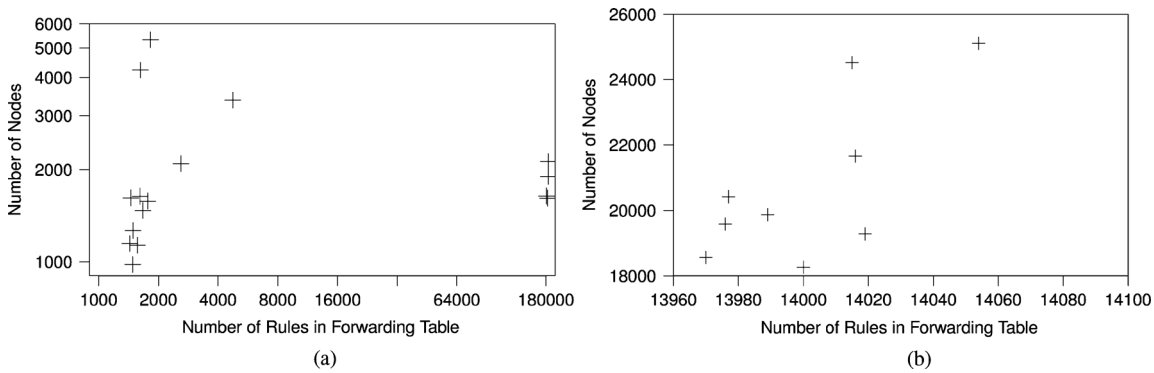


Fig. 3. Number of BDD nodes to represent all forwarding predicates of a forwarding table. (a) Stanford dataset. (b) Internet2 dataset.

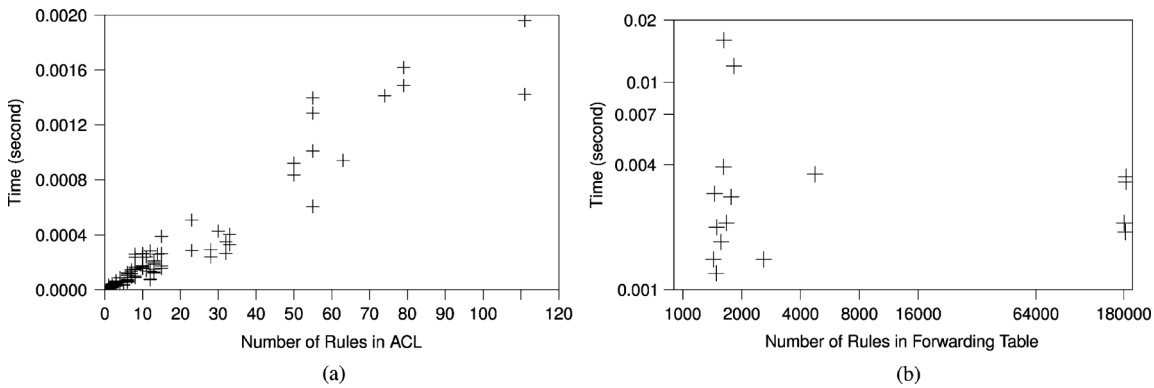


Fig. 4. Computation times of BDDs for the Stanford dataset. (a) Time to compute an ACL predicate. (b) Time to compute all forwarding predicates of a forwarding table.

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 element. 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 B).

Theorem 2: For a given set $\mathcal{P}$ of predicates, the atomic predicates for $\mathcal{P}$ specifies the equivalence classes in the set U with respect to $\mathcal{P}$.

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

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

It is easy to see that $\mathcal{A}(\{P\})$ satisfies Definition 1.

Second, let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two sets of predicates. Let $\mathcal{P}_1$'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:

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

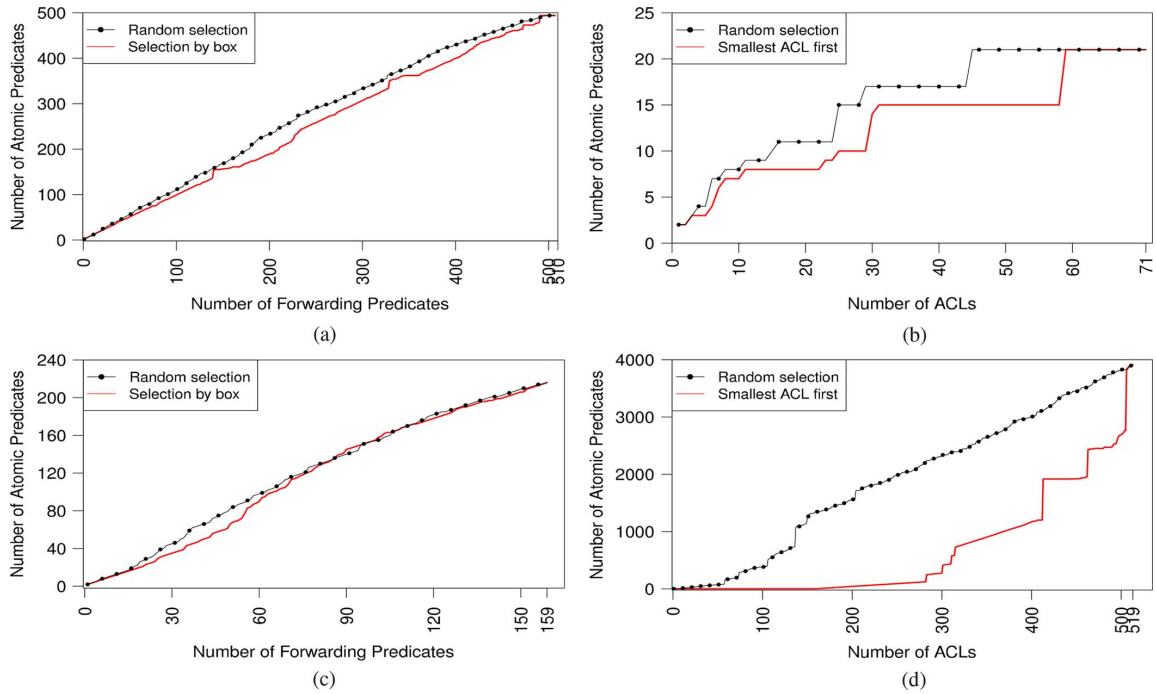


Fig. 5. Number of atomic predicates versus number of forwarding/ACL predicates. (a) Number of atomic predicates for forwarding in Stanford network. (b) Number of atomic predicates for ACLs in Stanford network. (c) Number of atomic predicates for forwarding in Internet2. (d) Number of atomic predicates for ACLs in Purdue network.

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 C).

Theorem 3: 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 (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). Algorithm 3 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.

C. Atomic Predicates in Real Networks

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.

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.

TABLE II
NUMBER OF ATOMIC PREDICATES IN THREE REAL NETWORKS

	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 III
TIME TO COMPUTE ATOMIC PREDICATES

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

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 box randomly and then select its forwarding predicates one by one randomly.

Fig. 5 shows growth of the number of atomic predicates in the three networks as the number of forwarding/ACL predicates increases. Fig. 5(a) and (c) shows that, for forwarding predicates, the number of atomic predicates grows approximately linearly with the number of forwarding predicates whichever selection method is used. Fig. 5(b) and (d) shows 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 II and Fig. 5, observe that the Stanford network has 71 ACLs with 1584 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 3917 for Purdue's 519 ACLs with 3605 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.

Table III shows times used to compute atomic predicates for the three networks. It 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 use *smallest ACL first* to compute atomic predicates for ACLs. Table III shows that the computation times for ACL atomic predicates in the Stanford and Purdue networks were 0.84 ms and 0.45 s, respectively.

We use *selection by box* to compute atomic predicates for forwarding. Table III shows that the computation times for forwarding atomic predicates in the Stanford network and Internet2 were 0.2 and 0.15 s, respectively.

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

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

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

V. COMPUTING REACHABILITY AND VERIFYING NETWORK PROPERTIES

Consider a network represented by a directed graph of 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 entrance of s to the exit of d*, i.e., the packets pass through both s and 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 [7].

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], a more efficient implementation of Header Space [12]. The results presented were computed on the same hardware (our workstation).

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 $S(A_1), \ldots, S(A_k)$.[2] In steps 1–2 of Algorithm 4, S_F and S_A represent the set of all packets that are injected into port s to test reachability. Algorithm 4 computes the set of packets that can be forwarded along the path in step 3, and it computes the set of packets that are allowed by ACLs

[2]Any predicate equal to *true* is not represented.

TABLE IV
COMPUTATION TIMES OF REACHABILITY, LOOP DETECTION, AND BLACK HOLE DETECTION

	Average (ms)	Median (ms)	Maximum (ms)
Hassel in C	233.57	48.57	2086.71
AP Verifier	0.91	0.98	1.48
AP Verifier (BDD)	2.82	2.92	11.51

(a) Port to port reachability computation for 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
AP Verifier (BDD)	1.15	0.06	10.89

(b) Port to port reachability computation for Internet2.

	Average (ms)	Median (ms)	Maximum (ms)
Hassel in C	218.22	53.45	1881.41
AP Verifier	0.95	1.03	1.38
AP Verifier (BDD)	4.23	4.21	10.67

(c) Reachability tree computation from one port (loop detection) 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
AP Verifier (BDD)	2.96	3.05	8.34

(d) Reachability tree computation from one port (loop detection) in Internet2.

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

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

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

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

along the path in step 7. If the Algorithm returns false in step 5 or in step 9, port d is not reachable from port s.

The reachability set from s to d is specified by the predicate P in (4) using S_F and S_A returned in step 11. 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 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: $S_F \leftarrow S_F \cap S(F_1) \cap \ldots \cap S(F_j)$
4: **if** $S_F = \emptyset$ **then**
5: **return** *false*
6: **end if**
7: $S_A \leftarrow S_A \cap S(A_1) \cap \ldots \cap S(A_k)$
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 IV(a) and (b). On 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. We also present the computation times of AP Verifier using BDDs without computing atomic predicates, called AP Verifier (BDD). As expected, it is slower than AP Verifier using atomic predicates, but it is still 2–3 orders of magnitude faster than Hassel in C.

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.

A search branch is terminated after visiting a port (say x) if one of the following conditions holds: 1) S_F or S_A becomes empty after visiting port x; 2) port x is an output port and there is no link connecting x to an input port; 3) port x is an input port of a box with no output port; and 4) 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.

Fig. 6 shows a small network example. The network has six atomic predicates, $f_1, f_2, f_3, f_4, f_5, f_6$, for forwarding and two 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 Fig. 7. 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. 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 both 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 Fig. 7.

Optimization techniques. AP Verifier uses several optimization techniques to reduce time for checking various network

[3]The C version of Hassel for Header Space is faster than the Python version [12] by about two orders of magnitude.

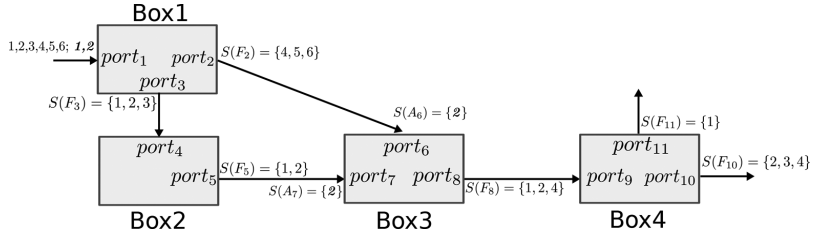


Fig. 6. Small network example.

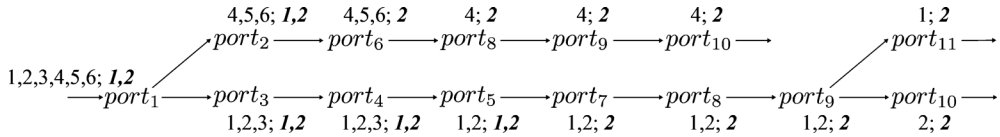


Fig. 7. Reachability tree of $port_1$.

TABLE V
STORAGE COSTS OF REACHABILITY TREES FROM PORTS. (A) STANFORD NETWORK (58 PORTS). (B) INTERNET2 (56 PORTS)

	Size (MB)
Hassel in C	323.06
AP Verifier	8.70

(a)

	Size (MB)
Hassel in C	187.60
AP Verifier	6.72

(b)

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

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 [12]. The same twelve infinite loop paths were detected by both AP Verifier and Hassel in C. For Internet2, we computed reachability trees for all ports. The same two infinite loop paths were detected by both AP Verifier and Hassel in C. Their computation times are presented in Table IV(c) and (d). On average, AP Verifier is 230 times faster than Hassel in C for the Stanford network and 2793 times faster for Internet2. We also present the computation times of AP Verifier (BDD) for comparison.

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 $S(F_1), S(F_2), \ldots S(F_k)$ be sets of identifiers of atomic predicates for output ports, $1, 2, \ldots, k$ of the box (including the special port for intentional packet drop). Let $S(true)$ be the set of identifiers of all atomic predicates for forwarding. The set of black holes is represented by the set

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

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 IV(e) and (f). On 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. (This is because every router in the Stanford network has the default route.) 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

TABLE VI
COMPUTATIONAL TIMES (IN MILLISECONDS) FOR DYNAMIC UPDATES. NETPLUMBER RESULTS ARE FROM [11]

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

(a) Link up in Stanford network.

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

(b) Link up in Internet2.

	Average	Median	Maximum
AP Verifier	0.00039	0.00027	32.64

(c) Link up in Purdue network.

	Average	Median	Maximum
AP Verifier	0.0028	0.00094	0.27

(d) Link down in Stanford network.

	Average	Median	Maximum
AP Verifier	0.0016	0.0011	0.10

(e) Link down in Internet2.

	Average	Median	Maximum
AP Verifier	0.00029	0.00023	12.69

(f) Link down in Purdue network.

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

(g) Rule insertion in Stanford network.

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

(h) Rule insertion in Internet2.

	Average	Median	Maximum
AP Verifier	0.020	0.00093	9.25

(i) Rule insertion in Purdue network.

	Average	Median	Maximum
AP Verifier	0.32	0.083	12.71

(j) Rule deletion in Stanford network.

	Average	Median	Maximum
AP Verifier	0.35	0.13	46.24

(k) Rule deletion in Internet2.

	Average	Median	Maximum
AP Verifier	0.023	0.0016	8.57

(l) Rule deletion in Purdue network.

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 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.

VI. REAL-TIME COMPLIANCE CHECK FOR NETWORK STATE CHANGES

In SDNs, a "verifier," placed in the communication path between the central controller and its switches, can intercept every network state change message and verify compliance of the state change with pre-defined network policies and properties. In this section, we describe how AP Verifier handles link up/down and rule insertion/deletion 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 [11]. 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 VI(a), (b), (d), and (e) 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 average, AP Verifier is *five to orders of magnitude* faster than NetPlumber.

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

[4]NetPlumber results are from [11]. They were computed using 6-core Xeon processors with 12 MB of L2-cache and 12 GB of DRAM.

TABLE VII
TIME TO COMPUTE INTERSECTIONS OF ACLS ALONG A PATH IN PURDUE DATASET

	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

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: 1) it checks if a port predicate is changed by the rule update; if so, it computes a new predicate for the port; 2) it updates the reachability tree using the new predicate, if any; and 3) it forks a process which runs on a second core to update the set of atomic predicates. Steps 2) and 3) occur concurrently.

Steps 1) and 2) 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 entrance. 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 $S(A_{\text{port}}) \cap S_A$, S_F, and F'_{port}, which together specify the following predicate:

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

The port's descendant nodes in the subtree are updated accordingly (more details in our technical report [21]). 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 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 Table IV(c) and (d); note that loop detection for a port is performed by computing its reachability tree).

We performed benchmark experiments [11] 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 VI(g), (h), (j), and (k). For rule insertions, the performance of AP Verifier is comparable to NetPlumber for the Stanford network; it is better than NetPlumber for Internet2.

[5]This is a configurable parameter value.

VII. PURDUE DATASET EXPERIMENTS

We performed experiments using AP Verifier and Hassel in C to compute port-to-port reachability sets for the Purdue dataset which has large ACLs but no forwarding tables. We started from a network topology including just the core routers in the dataset, and gradually increased the network size by including neighbors of boxes already chosen. For each network, we selected the shortest path in hop count that goes through core routers for each pair of boxes. We measured the times to compute intersections of ACLs along each path. The results are summarized in Table VII. On average, AP Verifier takes approximately 1–2 μs to compute intersections of ACLs along a path and is about two to three orders of magnitude faster than Hassel in C. Comparing with Table IV(a) and (b), it is noteworthy that the intersection of ACL predicates takes two to three orders of magnitude *less time* to compute than the intersection of forwarding predicates.

We performed experiments of real-time compliance check for network state changes using the Purdue dataset with 1,646 boxes and 2,376 ports. Forwarding tables were generated using shortest path routing. The results are shown in Table VI(c) and (f) for link status change and in Table VI(i) and (l) for ACL rule update.

VIII. RELATED WORK

A model for static reachability analysis of network state in the data plane was first presented by Xie *et al.* [20]. Gouda and Liu presented firewall decision diagram (FDD) for formal analysis of firewalls [8] and distributed firewalls [9]. Quarnet uses FDDs to represent ACLs in packet networks; it used tens to hundreds of seconds to compute reachability along paths with ACLs only [13].

There were two proposals to use general verification tools from other application domains. First, in ConfigChecker [3] (also FlowChecker [4] by the same first author), a BDD is used

to represent a set of *state transitions*. If n header bits are used for filtering, each BDD of ConfigChecker uses $2n$ state variables: n bit variables for packet header before state transition, and n bit variables for packet header after state transition. This BDD definition is necessitated by the symbolic model checking tool used by the authors. (In AP Verifier, each BDD represents a set of packets and requires the use of n bit variables only. Therefore, our use of BDDs is new and much more efficient than BDDs used in ConfigChecker/FlowChecker.) Second, Anteater [15] uses boolean formulas to represent policies (constraints) for packets travelling over edges in a network graph; it uses a SAT solver to check network properties. Both of these general-purpose tools are slow and operate on time scales of seconds to hours [14].

Custom-designed methods for reachability computation include Header Space/Hassel in C [12], NetPlumber [11], and Veriflow [14]. 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)* [14]. 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. Veriflow's experimental evaluation of its performance when there are multiple header fields was limited [14].

Of all these methods, only Header Space and AP Verifier compute reachability trees using different data structures (wildcard strings by Header Space, BDDs and atomic predicates by AP Verifier). Both of their reachability algorithms are based upon depth-first search. ConfigChecker and Anteater do not compute reachability trees. Veriflow computes a forward graph for each "equivalence class" of packets. To obtain port-to-port reachability and reachability trees, additional work has to be done to find relevant packet sets from different graphs and combine the results.

IX. 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 BDDs and atomic predicates to represent packet sets dramatically speeds up computation of reachability trees from ports. On average, AP Verifier is three orders of magnitude faster than Hassel in C. It also uses two to three orders of magnitude less memory than Hassel in C for computing and storing reachability trees from ports.

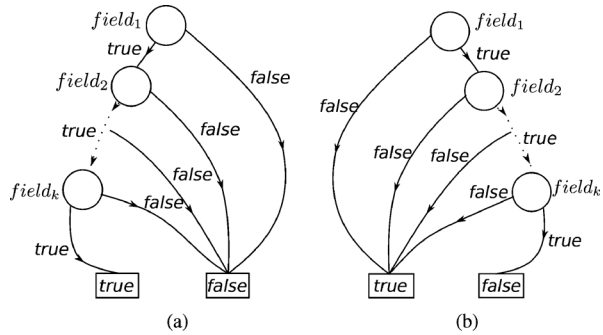


Fig. 8. BDD representation of an ACL rule. (a) $Action = allow$. (b) $Action = deny$.

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 four to five orders of magnitude smaller.

Routers and switches in a packet network perform packet filtering using one or more fields in the packet header. However, some also perform packet transformation when it changes, inserts, or removes some header field(s) of interest in verification, e.g., IP address and port number, MPLS label, or VLAN tag. To handle packet transformers, AP Verifier requires additional BDD operations and algorithms (omitted herein). Extending AP Verifier to networks with packet transformers is work in progress and the subject of a technical report under preparation.

Appendix A

Theorem 1: 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 number of nodes in the BDD graph representing 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 Fig. 8(a) and (b). 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

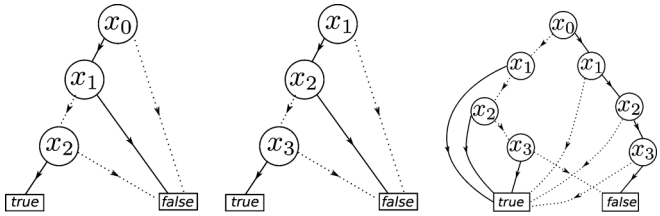


Fig. 9. BDD subgraphs representing a prefix, a suffix and an interval. The field has 4 bits represented by variables x_0, x_1, x_2, x_3. A dotted edge denotes an assignment to *false* and a solid edge denotes an assignment to *true*. (a) BDD subgraph for prefix $101*$. (b) BDD subgraph for suffix $*101$. (c) BDD subgraph for the interval from 0001 to 1110.

denies all packets, its BDD representation has only one node, the terminal node *false*.

For a nontrivial rule, there may be one or more nonterminal nodes in each circle. Let N_i be the number of nonterminal 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 Fig. 9(a) and (b) 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 [10] shows that it can be represented by at most $2h_i - 2$ prefixes. These prefixes can be divided into at most two sets such that, in each set, there is a longest prefix and all of the other prefixes can be obtained by left-shifting the longest prefix. Therefore, we can construct a BDD in which at most h_i nodes are used to represent all prefixes in each of the two sets. Thus, all prefixes of the ith field can be represented by a non-reduced binary decision diagram of at most $2h_i$ nodes (see Fig. 9(c) for a 4-bit example which has 7 nodes). So we have $N_i \leq 2h_i$.

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

APPENDIX B

Theorem 2: For a given set $\mathcal{P}$ of predicates, the atomic predicates for $\mathcal{P}$ specifies the equivalence classes in the set U with respect to $\mathcal{P}$.

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

For a predicate P and an element x, the *indicator function* $I_P(x)$ is defined as follows:

$$I_P(x) = \begin{cases} 1, & x \in \text{ the set specified by } P \\ 0, & \text{otherwise.} \end{cases}$$

Given a set $\mathcal{P}$ of predicates, two elements, x_1 and x_2 are *equivalent* w.r.t. $\mathcal{P}$ if and only if $I_P(x_1) = I_P(x_2), \forall P \in \mathcal{P}$.

The equivalence relation partitions the set U of all elements into *equivalence classes*, $\{C_1, \ldots, C_n\}$, that is, *for every pair of elements, x_1 and x_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: $I_P(C_i) = I_P(x), \forall x \in C_i$, 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, $C_i \neq \emptyset, \forall i \in \{1, \ldots, k\}$; thus Property 1 is satisfied. The equivalence classes partition the set of U of all elements; thus, the disjunction of all predicates is *true* and Property 2 is satisfied. An element 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 set specified by P be $\{x \mid I_P(x) = 1\}$. We prove Property 4 by proving that an element x' is in $\{x \mid I_P(x) = 1\}$ *if and only if* element x' is in $\cup_{I_P(C_i)=1} C_i$.

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

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

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

Lemma 2 below states that given a set $\mathcal{P}$ of predicates, the set of predicates that specify equivalence classes $\{C_1, \ldots, C_n\}$ is minimum, thus satisfying Property 5 in Definition 1. Hence, the predicates that specify $\{C_1, \ldots, C_n\}$ are the atomic predicates of $\mathcal{P}$.

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 $\{q_1, \ldots, q_m\}$ 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 set specified by q_i. This implies that $m \geq n$ which is minimum.

Proof: For any predicate $P \in \mathcal{P}$, from the assumption that $\{q_1, \ldots, q_m\}$ satisfies the fourth property of Definition 1, P can be represented by the disjunction of a subset of $\{q_1, \ldots, q_m\}$. Consider some $q_i \in \{q_1, \ldots, q_m\}$ and choose any two elements, x_1 and x_2, from the set specified by q_i. We will show that x_1 and x_2 are equivalent. There are two possibilities in the disjunction representation of P. First, q_i appears in the subset representing P, in which case, $I_P(x_1) = I_P(x_2) = 1$. Second, q_i does not appear in the subset representing P, in which case, $I_P(x_1) = I_P(x_2) = 0$. Therefore, $I_P(x_1) = I_P(x_2), \forall P \in \mathcal{P}$. Thus, x_1 and x_2 are equivalent w.r.t. $\mathcal{P}$, and $x_1, x_2 \in C_j$ for some $j \in \{1, \ldots, n\}$. Thus $C_j \supseteq$ the set specified by $q_i \in \{q_1, \ldots, q_m\}$. ■

APPENDIX C

Theorem 3: 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).

Proof: We prove the theorem 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 elements, x_1 and x_2, x_1 is equivalent to x_2 w.r.t. $\mathcal{P}_1 \cup \mathcal{P}_2$ *if and only if* there exists $i \in \{1, \ldots, k\}$ such that x_1 and x_2 belong to the set specified by a_i.

If part: Assume that there exists $i \in \{1, \ldots, k\}$ such that x_1, x_2 belong to the 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 x_1, x_2 belong to the set specified by b_{i_1} and to the set specified by d_{i_2}. From Theorem 2, b_{i_1} and d_{i_2} each specifies an equivalence class w.r.t. $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Thus, $\forall P \in \mathcal{P}_1$, $I_P(x_1) = I_P(x_2)$, and $\forall P \in \mathcal{P}_2$, $I_P(x_1) = I_P(x_2)$. Therefore, $I_P(x_1) = I_P(x_2), \forall P \in \mathcal{P}_1 \cup \mathcal{P}_2$. That is, x_1 and x_2 are equivalent w.r.t. $\mathcal{P}_1 \cup \mathcal{P}_2$.

Only if part: Assume that x_1 and x_2 are equivalent w.r.t. to $\mathcal{P}_1 \cup \mathcal{P}_2$. Then, we have $I_P(x_1) = I_P(x_2), \forall P \in \mathcal{P}_1$, and $I_P(x_1) = I_P(x_2), \forall P \in \mathcal{P}_2$. Thus, x_1, x_2 are equivalent w.r.t. $\mathcal{P}_1$, and $x_1, x_2 \in$ the equivalence class specified by b_{i_1}, for some $i_1 \in \{1, \ldots, l\}$. Similarly, we can show that x_1, $x_2 \in$ the equivalence class specified by d_{i_2}, for some $i_2 \in \{1, \ldots, m\}$. Since $x_1, x_2 \in$ 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 $x_1, x_2 \in$ the set specified by a_i.

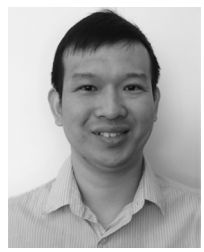
Consequently, the set $\{a_1, \ldots, a_k\}$ specifies the set of equivalence classes of U w.r.t. $\mathcal{P}_1 \cup \mathcal{P}_2$. Thus $\mathcal{A}(\mathcal{P}_1 \cup \mathcal{P}_2) = \{a_1, \ldots, a_k\}$. ■

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers of ToN for their constructive comments.

REFERENCES

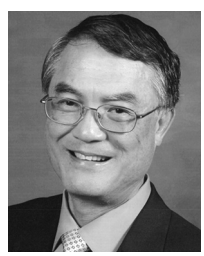
[1] Header Space Library and NetPlumber. [Online]. Available: https://bitbucket.org/peymank/hassel-public/
[2] The Internet2 Observatory Data Collections. [Online]. Available: http://www.internet2.edu/observatory/archive/data-collections.html
[3] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi, "Network configuration in a box: Towards end-to-end verification of network reachability and security," in *Proc. IEEE ICNP*, Princeton, NJ, USA, 2009, pp. 123–132.
[4] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated openflow infrastructures," in *Proc. ACM SafeConfig*, Chicago, IL, USA, 2010, pp. 37–44.
[5] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986.
[6] R. Draves, C. King, V. Srinivasan, and B. Zill, "Constructing optimal IP routing tables," in *Proc. IEEE INFOCOM*, New York, NY, USA, 1999, pp. 88–97.
[7] E. A. Emerson, , J. van Leeuwen, Ed., "Temporal and Modal Logic," in *Handbook of Theoretical Computer Science*. Cambridge, MA, USA: MIT, 1990, vol. B.
[8] M. G. Gouda and A. X. Liu, "Firewall design: Consistency, completeness, and compactness," in *Proc. IEEE ICDCS*, Tokyo, Japan, 2004, pp. 320–327.
[9] M. G. Gouda, A. X. Liu, and M. Jafry, "Verification of distributed firewalls," in *Proc. IEEE GLOBECOM*, New Orleans, LA, USA, 2008, pp. 1–5.
[10] P. Gupta, "Algorithms for routing lookups and packet classification," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 2000.
[11] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. USENIX NSDI*, Lombard, IL, USA, 2013, pp. 99–112.
[12] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. USENIX NSDI*, San Jose, California, 2012, pp. 113–126.
[13] A. R. Khakpour and A. X. Liu, "Quantifying and querying network reachability," in *Proc. IEEE ICDCS*, Genoa, Italy, 2010, pp. 817–826.
[14] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proc. USENIX NSDI*, Lombard, IL, USA, 2013, pp. 15–27.
[15] 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, pp. 290–301.
[16] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?," in *Proc. 4th Conf. USENIX Symp. Internet Technol. Syst.*, Seattle, WA, USA, 2003, pp. 1–16.
[17] Y.-W. E. Sung, S. G. Rao, G. G. Xie, and D. A. Maltz, "Towards systematic design of enterprise networks," in *Proc. ACM CoNEXT*, Madrid, Spain, 2008, pp. 1–12.
[18] A. Vahidi, "JDD, a pure Java BDD and Z-BDD library," [Online]. Available: http://javaddlib.sourceforge.net/jdd/
[19] E. Wong, "Validating network security policies via static analysis of router ACL configuration," M.S. thesis, Dept. Comput. Sci., Naval Postgraduate School, Annapolis, MD, USA, 2006.
[20] 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 *Proc. IEEE INFOCOM*, Miami, FL, USA, 2005, pp. 2170–2183.
[21] H. Yang and S. S. Lam, "Real-time verification of network properties using Atomic Predicates," Comput. Sci. Dept., Univ. of Texas at Austin, Austin, TX, USA, Tech. Rep. TR-13-15, Aug. 2013.



Hongkun Yang (S'12) received the B.S.E. (with distinction) and M.S.E. degrees from Tsinghua University, Beijing, China, in 2007 and 2010, respectively. He is currently working toward the Ph.D. degree at the Department of Computer Science, University of Texas at Austin, Austin, TX, USA.

His research interests include computer networks, protocol verification, network security, and formal methods. He has authored and coauthored research papers in a number of conferences and journals.

Mr. Yang was a recipient of the Microelectronics and Computer Development (MCD) Fellowship.



Simon S. Lam (F'85) received the B.S.E.E. degree (with distinction) from Washington State University, Pullman, WA, USA, in 1969, and the M.S. and Ph.D. degrees in engineering from the University of California, Los Angeles, CA, USA, in 1970 and 1974, respectively.

From 1971 to 1974, he was a Postgraduate Research Engineer with the ARPA Network Measurement Center, University of California, Los Angeles, CA, USA, 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, USA. Since 1977, he has been on the faculty of the University of Texas at Austin, where he is a Professor and Regents Chair in computer science and served as Department Chair from 1992 to 1994.

Prof. Lam is a Fellow of ACM and a member of the National Academy of Engineering. He was the recipient of 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. He served as Editor-in-Chief of the IEEE/ACM TRANSACTIONS ON NETWORKING from 1995 to 1999. He served on the editorial boards of the IEEE/ACM TRANSACTIONS ON NETWORKING, the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, *IEEE Transactions on Communications*, the PROCEEDINGS OF THE IEEE, *Computer Networks*, and *Performance Evaluation*. He cofounded the ACM SIGCOMM conference in 1983 and the IEEE International Conference on Network Protocols in 1993.