Scalable Verification of Networks With Packet Transformers Using Atomic Predicates

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

***Abstract*— Packet transformers are widely used in ISPs, datacenter infrastructures, and layer-2 networks. Existing network verification tools do not scale to large networks with transformers (e.g., MPLS, IP-in-IP, and NAT). Toward scalable verification, we conceived a novel packet equivalence relation. For networks with packet transformers, we first present a formal definition of the packet equivalence relation. Our transformer model is general, including most transformers used in real networks. We also present a new definition of atomic predicates that specify the *coarsest equivalence classes* of packets in the packet space. We designed an algorithm for computing these atomic predicates. We built a verifier, named *A*tomic *P*redicates for *T*ransformers, and evaluated its performance using four network data sets with MPLS tunnels, IP-in-IP tunnels, and NATs. For a provider cone data set with 11.6 million forwarding rules, 92 routers, 1920 duplex ports, and 40 MPLS tunnels which use 170 transformers, APT used only 0.065 s, on average, to compute the reachability tree from a source port to all other ports for all packets and perform loop detection as well. For the Stanford and Internet2 data sets with NATs, APT is faster than HSA (Hassel in C implementation) by two to three orders of magnitude. By working with atomic predicates instead of individual packets, APT achieves verification performance gains by orders of magnitude.**

***Index Terms*— Network verification, reachability analysis, packet transformers, formal methods, algorithm design.**

I. Introduction

THE process of forwarding packets in networks is prone to faults from configuration errors and unexpected protocol interactions. Active probes (pings and traceroutes) are widely used to measure data plane reachability. These tools, however suffer from major limitations and biases [8].

Automated tools based upon formal methods have been developed in recent years for verifying reachability properties in the data plane: such as, "a packet with certain header values cannot reach host z," "the network has no forwarding loop for any packet," "all packets from specified input ports must pass through a given sequence of firewalls." Substantial progress has been made in developing formal methods with efficient algorithms for verifying networks of packet filters, representing forwarding tables and access control lists (ACLs), but not much progress so far for networks with packet transformers.

We observe that tunnels and NATs are widely used in packet networks. A measurement study shows that all tier-1 ISPs and more than half of large ISPs used MPLS tunnels [20]. The Internet also has IP-in-IP tunnels used by IPsec and for the co-existence of IPv4 and IPv6. Furthermore, researchers continue to propose new applications of IP-in-IP tunnels [11].

The entry and exit routers of a tunnel perform header encapsulation and de-encapsulation, respectively. Each transit router in a MPLS tunnel performs label switching. All of them perform packet transformations. NATs rewrite packets and are packet transformers. Therefore, verifying reachability properties using a network model with packet filters only without including packet transformers would not be useful for many networks in the real world.

An abstract framework for addressing the network verification problem using static reachability analysis was proposed by Xie et al. [25]. In this framework, a network consists of packet filters and transformers. A *packet filter* is specified by a ***predicate*** which represents a set of packets for which the predicate evaluates to true and can pass through the filter. A *packet transformer* maps a set of packets to another set of packets in the set of all packets, namely: the *packet space*.

Designing and building an automated tool for verifying reachability properties of a large network with packet transformers as well as filters present major challenges. State-of-the-art verification tools fail to meet these challenges for one of two reasons: (i) they are computationally efficient but do not model packet transformers; (ii) they can, *in theory*, model packet transformers but are computationally inefficient and not scalable. In this paper, we present a verification tool, **APT**, based upon a novel packet equivalence relation, for verifying networks with many different packet transformers as well as filters; the tool is computationally efficient enough to scale to networks with millions of forwarding rules and large numbers of routers, ports, and packet transformers.

A. Related Work

In recent years, researchers have proposed network verification methods and tools by following one of two different approaches: (i) custom design new data structures and algorithms to compute reachability sets directly [12]–[14], [26], [28]; (ii) reformulate the network verification problem within the context of verification tools previously designed for other problem domains [5], [6], [15], [16], [22].

Manuscript received August 19, 2016; revised March 7, 2017 and May 2, 2017; accepted May 26, 2017; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor A. Bremler-Barr. Date of publication July 17, 2017; date of current version October 13, 2017. This work was supported in part by the National Science Foundation under Grant CNS-1214239. *(Corresponding author: Simon S. Lam.)*

H. Yang is with Google, Mountain View, CA 94043 USA (e-mail: hkyang@google.com).

S. S. Lam is with the Department of Computer Science, The University of Texas at Austin, Austin, TX 78712 USA (e-mail: lam@cs.utexas.edu).

Digital Object Identifier 10.1109/TNET.2017.2720172

In the first approach, HSA [13] presents a new data structure, wildcard expressions, for representing packet sets together with methods for computing reachability trees to verify network properties. NetPlumber [12], based upon HSA, and Veriflow [14] were designed for software defined networking with a controller to perform incremental verification after a network configuration change. The network model of Veriflow has packet filters only. AP Verifier [26], [28], also designed for packet filters only, uses atomic predicates computed from the packet filters to substantially reduce both computation time and space for network verification.

In the second approach, among methods that model some packet transformations, ConfigChecker [5] uses symbolic model checking; Anteater [16] uses a SAT solver; Spinoso et al. [21] use a SMT solver; NoD [15] uses Datalog. SymNet [22] uses symbolic execution. The general verification tools employed have expressive modeling languages which can be used to specify packet transformers; some were also used to model and specify stateful NATs [21], [22]. However, since the data structures and algorithms of these tools were originally designed for other problem domains, they run very slowly for network verification.

The designers of NoD replaced the native data structure of Datalog by a new data structure more suited to representing packet sets. As a result, [15, Table 3] shows that NoD runs much faster than model checkers and SAT/SMT solvers but it is still 20 times slower than Hassel (the optimized version of HSA written in C [3]). Symnet is reported to be 50% slower than Hassel [22]. Lastly, AP Verifier was shown to be 2-3 orders of magnitude faster than Hassel [26], [28].

For networks with *packet filters only*, packet equivalence is intuitive and not hard to define. This is because when a packet arrives at a filter, it either exits unchanged or is filtered. Informally, two packets are equivalent if and only if they are treated identically by every filter in the network. This idea was applied to speed up network verification in Veriflow [14] and in AP Verifier [26], albeit these tools use very different data structures and algorithms. AP Verifier is much more efficient than Veriflow because its algorithm computes the coarsest equivalence classes of packets in the packet space (i.e., the number of equivalence classes is smallest). Recently, Plotkin et al. [18] presented a different approach based upon bisimulation and modal logic to speed up network verification and wrote: "A side effect of all this machinery is the ability to formalize earlier concepts such as slicing and a (generalized version of) Yang-Lam equivalence" where "Yang-Lam equivalence" refers to atomic predicates defined for a network of packet filters [26], [28].

B. Contributions of This Paper

Towards scalable verification of packet networks in the real world, we set out to develop a general theory of packet equivalence for networks with both transformers and filters. When a packet arrives at a transformer, it may be filtered. If not filtered, it may exit unchanged, exit as another packet (deterministic transformation), or exit as any packet in a specified set of packets (non-deterministic transformation). To handle most packet transformations in real networks, the problem is substantially more challenging than the one we solved previously for filters only [26], [28]. We needed a new definition of packet equivalence *together with* a new algorithm for computing the coarsest equivalence classes (i.e., atomic predicates). The major contributions in this paper are summarized as follows:

1) A General Theory of Packet Equivalence: Every packet injected into the network may possibly be transformed into other packets *by any sequence of transformers* in the network. Therefore, we need a new packet equivalence relation that formalizes the following intuition: namely, two packets are equivalent if and only if they are treated identically by every filter and by every possible sequence of one or more transformers in the network. After a number of attempts, we conceived a formal definition of the intuition based upon the following insight: For every sequence of transformations, consider the sequence of inverse mappings instead of the sequence of forward mappings. (See Definition 3 in Section IV.)

Subsequently, we solved two additional hard problems: (i) formulating a *new definition* of atomic predicates for transformers and filters with a *proof* that they specify the coarsest equivalence classes of packets, and (ii) designing *a new algorithm* for computing atomic predicates for transformers and filters with a *proof* that the algorithm terminates and, upon termination, it returns the set of atomic predicates. The definition, algorithm, and theorems are presented in Sections IV and V. Proofs of the theorems are presented in the Appendix.

2) Formulas for Transformed Predicates: For a set of packets specified by a predicate, P, we derived formulas for computing the transformed predicate, $T(P)$, where T is one of the following five different transformations: packet rewriting; encapsulation and de-encapsulation of a new header; encapsulation and de-encapsulation of a new instance of the outermost header. These formulas are novel and necessary for computing transformations of atomic predicates in implementation.

3) APT Implementation and Its Performance: The new algorithm and formulas are implemented in a new verification tool, APT, in which sets of equivalent packets are represented by integer identifiers of atomic predicates. By working with atomic predicates instead of individual packets, APT achieves performance gains in computation time and space by orders of magnitude.

The performance of APT was evaluated using the Stanford and Internet2 datasets, as well as two large provider cone datasets. (The provider cone of a network consists of all of its direct and indirect Internet service providers including tier-1 ISPs.) Various numbers of NATs, IP-in-IP tunnels, and MPLS tunnels were added into each dataset for performance evaluation of APT.

For each of the datasets with transformers, we measured the times to compute reachability trees from source ports which can be used to verify safety and progress properties specified by a temporal logic (e.g., CTL [10]). For examples, the reachability tree from a source port to all other ports in the same network can be used to detect forwarding loops for all packets injected into the source port, and for verifying that all injected packets traverse a specified sequence of waypoints

in the network. Performance results for the two large provider cone datasets demonstrate that APT is scalable to very large networks, such as, ISP networks and large-scale datacenter infrastructures.

Since Hassel supports packet rewriting,[1] we ran the Hassel in C code on our computer for the Stanford dataset with 0 to 14 NATs added and the Internet2 dataset with 0 to 9 NATs added and compare its results with those of APT. We found that APT is faster than Hassel in C by 2 orders of magnitude for the Stanford dataset and faster by 3 orders of magnitude for the Internet2 dataset. *For the Stanford dataset, APT also found the same 12 infinite forwarding loops found by Hassel in C.* This direct comparison provides validation for APT.

Lastly we experimented and found that APT recovers quickly from dynamic network changes including link/box status change, addition/removal of a NAT or tunnel, and insertion/deletion of rules.

The balance of this paper is as follows: Our network model is presented in Section II. Formulas for transformed predicates are presented in Section III. Our theory on packet equivalence and atomic predicates is presented in Section IV. In Section V, our theory is applied to the design of an algorithm for computing atomic predicates. Performance evaluation of APT is presented in Section VI. We conclude in Section VII. Proofs of theorems are in the Appendix.

II. Network Model

We use 'box' to refer to any network device that forwards packets, including routers, switches, as well as middle boxes such as firewalls, NATs, etc. A packet network is modeled as a directed graph of boxes. We use predicates to represent packet sets. A predicate is a Boolean formula where each variable represents one packet header bit.[2] A predicate represents a set of packets for which the predicate evaluates to true. Predicate *false* specifies the empty set, and predicate $true$ specifies the set of all packets.

ACLs in our model make filtering decisions for each packet in isolation. In particular, our model and datasets do not include *stateful* NATs. However, APT can be used to analyze the dynamic behavior of a stateful NAT by modeling such behavior as rule insertion and deletion. To illustrate, consider a stateful NAT which tracks TCP connections between a private network and the outside world. When a host in the private network initiates a TCP connection to a server outside, a new rule is added to the ACL guarding the NAT's input port to admit packets of the TCP connection from outside. When the TCP connection is closed, the rule is deleted from the ACL. In Section VI-F, we show that APT adapts quickly to dynamic network changes, including rule insertion and deletion.

Research on the verification of stateful NATs and other middle boxes is at an early stage. A comprehensive study of the topic is beyond the scope of this paper.

[1]Hassel has no implementation for header encapsulation and de-encapsulation because HSA's data structure does not support these transformations.

[2]In theory, a variable can also represent a bit in the payload of a packet. In practice, we consider header bits only.

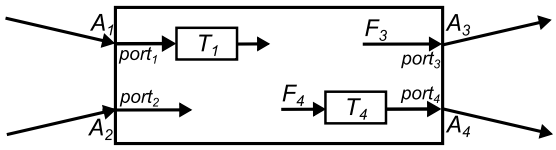


Fig. 1. An example of a box with packet transformers T_1 and T_4. $A_1, \ldots, A_4$ are predicates for ACLs, F_3 and F_4 are predicates for forwarding.

A. Box Model

Each box has at least one input port and one output port. (We use 'port' to refer to either an input or output port. A *duplex port* is one endpoint of a full-duplex link consisting of an input port and an output port.) A forwarding table is used to forward each input packet to one or more output ports (multicast is allowed). A special port for intentional drops is allowed. The forwarding table may be obtained from multiple protocols, such as IPv4, IPv6, MPLS, etc.

In the model, the forwarding table is converted to a set of predicates such that each output port is guarded by a predicate for forwarding. Each port, input or output, is also guarded by a predicate specified by an ACL. (If a port is not guarded by an ACL, the predicate of the port is *true*.)

Our model of a box is illustrated in Figure 1. A box may have one or more packet transformers, or none.

B. Packet Transformer Model

In general, packet transformers are modeled as functions that map an input packet set to an output packet set. For a packet transformer T and a predicate P specifying its input packet set, $T(P)$ denotes the transformed predicate specifying the output packet set. This model includes both transformers that are: (i) *deterministic* (one packet is mapped to another packet), and (ii) *nondeterministic* (one packet is mapped to any packet in a set of packets).

Three widely used types of packet transformations are implemented in APT: (i) *packet translation* which rewrites the bits of an existing header; (ii) *packet encapsulation* which prepends a header to the existing packet header; (iii) *packet de-encapsulation* which removes the outermost header from a packet. For examples, packet translation is performed in NATs; packet encapsulation and de-encapsulation are performed for IP-in-IP tunnels; all three transformations are performed for MPLS tunnels. Any transformer T, not in any of these three types, can also be implemented by deriving a new formula for $T(P)$ as illustrated in Section III.

Consider the box example in Figure 1. T_1 and T_4 denote two packet transformers. Note that a packet may be transformed after it enters an input port and before forwarding (e.g., encapsulation, de-encapsulation, and MPLS label switching). After forwarding, the packet may be transformed before it is sent to an output port (e.g., NAT).

Lastly, each transformer in our model may actually be multiple transformers in sequence, e.g., one encapsulation immediately following another. For example, if T_1 is actually T_a followed by T_b, then for input packet set P, the output packet set of T_1 is computed from $T_1(P) = T_b(T_a(P))$.

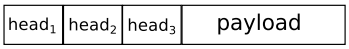


Fig. 2. An example of a packet with a stack of headers. head_1 is at top of the stack, head_3 is at bottom of the stack.

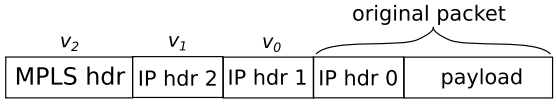


Fig. 3. Auxiliary variables v_0, v_1, v_2, are used in predicates to indicate presence of protocol headers in a header stack.

C. Packet Header Model

The header of a packet is modeled by a *stack of protocol headers* (see example in Figure 2), also referred to as *header stack* or just *stack*. Encapsulation pushes a new protocol header into the stack. De-encapsulation removes a protocol header from the top of the stack. Packet translation rewrites some bits within the packet header.

There are multiple fields in each protocol header within a header stack. Not all of them are used by packet filters and transformers in a given network. APT uses only header fields that are *relevant* for verifying the network's reachability properties, such as, IP addresses, port numbers, MPLS label, etc. In the balance of this paper, all fields mentioned are relevant fields.

Relevant fields in different protocol headers within a header stack are represented by different bit variables. For example, if IPv4 packets are encapsulated with a new IPv4 header, the inside destination IPv4 address and the outside destination IPv4 address are different fields.

Each possible header stack corresponds to a unique sequence of relevant header fields, which is represented by a sequence of bit variables in our model. The set of all possible values of such bit strings represents the set of all packets in our model, i.e., the *packet space*. In practice, packets have a maximum size. Therefore, the set of all possible header stacks is finite and the *set of all packets is finite*.

III. Predicate Transformations

Consider a predicate P specifying the input packet set of a packet transformer T, we present in this section five formulas for computing $T(P)$ for the three basic types of packet transformations. *These formulas are used by APT for computing transformations of atomic predicates*.[3]

Auxiliary Variables: Extending the model of a packet header to a header stack requires the use of auxiliary variables in each predicate to identify the protocol headers in a stack. *Auxiliary variables do not represent any real header bits; they are not implemented and are used for verification only.*

Consider the example in Figure 3 which shows three binary variables, v_0, v_1, v_2, which are auxiliary variables. For examples, $v_2 = 1$ indicates a MPLS encapsulation. $v_1v_0 = 01$ indicates one IP encapsulation, and $v_1v_0 = 11$ indicates two IP encapsulations. For a packet without any encapsulation, these variables are set to $v_0 = 0, v_1 = 0$, and $v_2 = 0$.

Logical Operations: Logical operations used for computing the output packet set of a packet transformer include conjunction ("$\wedge$"), disjunction ("$\vee$"), negation ("$\neg$"), substitution ("$|$"), and existential quantification ("$\exists$"). The first three operations are common logical operations. In the following, we briefly introduce substitution and existential quantification.

We use notation $P|_{x=b}$ to denote the predicate computed by the substitution operation that replaces variable x in predicate P by expression b. In general, b can be a constant (either $true$ or $false$) or a predicate. When b is a constant (either $true$ or $false$), predicate $P|_{x=b}$ is called a restriction of P. The existential quantification of variable x is defined using substitution:

$$\exists x.P = P|_{x=true} \vee P|_{x=false}, \tag{1}$$

which removes each occurrence of x (either x or $\neg x$) in predicate P.

A. Packet Rewrite

Consider packets with variables $x_1, x_2, \ldots, x_k$ representing bits to be translated in their headers. Suppose the header bits are changed to new values specified by predicate Q that has variables $x_1, x_2, \ldots, x_k$. We use existential quantification and conjunction to change the values of variables $x_1, x_2, \ldots, x_k$. In predicate notation, incoming packets specified by P are translated to output packets specified by

$$T(P) = (\exists x_k. \ldots \exists x_2. \exists x_1. P) \wedge Q, \tag{2}$$

For example, let x_i, x_j be the boolean variables representing two header bits of packets in P. These two bits are set to $x_i = 0, x_j = 1$ by a NAT. The set of packets exiting the NAT is specified by the predicate, $(\exists x_j. \exists x_i. P) \wedge (\neg x_i \wedge x_j)$.

Note that packet translation does not change the value of any auxiliary variable because the header stack is not changed.

B. Encapsulation and De-Encapsulation

There are two cases of encapsulation and de-encapsulation: (i) encapsulation of a protocol header different from the protocol header on top of the header stack, and its subsequent de-encapsulation, e.g., IPv4 in MPLS; (ii) encapsulation of a new instance of the protocol header on top of the header stack, and its subsequent de-encapsulation, e.g., IPv4 in IPv4.

(i) A Different Protocol Header: For encapsulation, let predicate P specify a set of packets entering the transformer. Let H be a predicate that specifies the new header encapsulating the packets and v be the auxiliary variable for the new header. H has variables $y_1, y_2, \ldots, y_k$ representing bits in the new header. $y_1, y_2, \ldots, y_k$ do not exist in P.

We use conjunction ("$\wedge$") and existential quantification ("$\exists$") to add the new header and set the auxiliary variable v to 1. The set of encapsulated packets leaving the transformer is specified by predicate

$$T(P) = (\exists v.(H \wedge P)) \wedge v. \tag{3}$$

[3]These formulas are novel and necessary for implementation. However, for readers not interested in predicate logic, they can skip ahead to Section IV by considering $T(P)$ as the result returned by a function call, and still fully understand the other key ideas and contributions of this paper.

At a transformer that de-encapsulates packets specified by $T(P)$, the following predicate is computed:

$$(\exists v.\exists y_k.\ldots.\exists y_2.\exists y_1.T(P)) \wedge (\neg v) \tag{4}$$

which specifies the set of de-encapsulated packets leaving the transformer. Existential quantifications on $y_1, y_2, \ldots, y_k$ remove the encapsulated header, and auxiliary variable v is set to 0 by the conjunction.

(ii) A New Instance of the Protocol Header on Top of Stack: We use IPv4 in IPv4 as an example. The formulas are the same for other protocols. For encapsulation, let P be a predicate specifying a packet set entering the transformer. Let H be a predicate that specifies the IPv4 header encapsulating the packets. Prior to encapsulation, the outermost header of packets specified by P is IPv4. Both H and P have variables $x_1, \ldots, x_k$ because they are for the same protocol (IPv4).

There are two steps to perform this encapsulation: (a) rename variables of the existing outermost IPv4 header; (b) set the auxiliary variable v for the new IPv4 header to 1 and push the new IPv4 header specified by H into the header stack.

To perform step (a), we use substitution operations and compute predicate $P|_{x_1=y_1,x_2=y_2\ldots,x_k=y_k}$, where each occurrence of $x_1, x_2, \ldots, x_k$ in P is replaced by $y_1, y_2, \ldots, y_k$, respectively. To perform step (b), we use existential quantification ("$\exists$") and conjunction ("$\wedge$") to set the auxiliary variable and use conjunction to add the encapsulated header.

Therefore, the encapsulated packet set $T(P)$ is computed by the following formula:

$$T(P) = (\exists v.(P|_{x_1=y_1,x_2=y_2,\ldots,x_k=y_k})) \wedge v \wedge H. \tag{5}$$

There are two steps to perform de-encapsulation: (a) remove the outermost IPv4 header represented by variables $x_1, x_2, \ldots, x_k$, and set auxiliary variable v to 0; (b) rename variables $y_1, y_2, \ldots, y_k$ of the IPv4 header that becomes the outermost IPv4 header after de-encapsulation. Thus, the de-encapsulated packet set is computed by the following formula

$$((\exists v.\exists x_k.\ldots.\exists x_1.T(P)) \wedge (\neg v))|_{y_1=x_1,y_2=x_2,\ldots,y_k=x_k}, \tag{6}$$

where predicate $\exists v.\exists x_k.\ldots.\exists x_1.T(P)$ does not have any occurrence of $v, x_k, \ldots,$ and x_1.

IV. Theory

Let U denote the set of all elements.[4] Without qualification, an element x is always in set U, and a set of elements is always a subset of U. A predicate specifies a set of elements in U. Predicate $true$ specifies U. Predicate $false$ specifies the empty set.

The indicator function for a set D of elements and an element x is defined as follows:

$$I_D(x) = \begin{cases} 1 & x \in D, \\ 0 & x \notin D. \end{cases}$$

[4]In the context of a packet network, U is the packet space. Definition 4 in this section is the foundation of Algorithm 1 for computing atomic predicates in Section V.

A. Atomic and Representative Predicates for a Set of Filters

We first consider a network with a set, $\mathcal{P}$, of predicates (representing filters) only. Two elements, x_1 and x_2 are *equivalent* w.r.t. $\mathcal{P}$ if and only if $I_P(x_1) = I_P(x_2), \forall P \in \mathcal{P}$, where P is interpreted as a set of elements.

Definition 1 (Atomic Predicates for $\mathcal{P}$): Given a set $\mathcal{P}$ of predicates, its set of atomic predicates $\{b_1, \ldots, b_n\}$ satisfies these five properties:

1) $b_i \neq false, \forall i \in \{1, \ldots, n\}$.
2) $\vee_{i=1}^{n} b_i = true$.
3) $b_i \wedge b_j = false$, *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:*

$$P = \bigvee_{i \in S(P)} b_i, \quad \textit{where } S(P) \subseteq \{1, \ldots, n\}. \tag{7}$$

5) *n is the* minimum *number such that the set $\{b_1, \ldots, b_n\}$ satisfies the above four properties.*

Definition 1 is from [26] and [28]. By satisfying property 5 in the above definition, atomic predicates specify the *coarsest* equivalence classes w.r.t $\mathcal{P}$ (i.e., smallest number of equivalence classes). They can be used to provide the best computation time and space performance in network verification.

For a given set $\mathcal{P}$ of predicates, we will use $\{C_1, \ldots, C_n\}$ to denote the set of equivalence classes specified by the atomic predicates for $\mathcal{P}$, where n is the number of atomic predicates.

We will refer to a set of predicates that satisfies the first four properties of the above definition as *representative*. Representative predicates also specify equivalence classes w.r.t. $\mathcal{P}$ and may be used in lieu of atomic predicates in network verification, albeit less efficiently than atomic predicates.

Definition 2 (Representative Predicates for $\mathcal{P}$): A set, $\mathcal{B} = \{b_1, b_2, \ldots, b_l\}$, of predicates is representative of $\mathcal{P}$ if and only if $\mathcal{B}$ satisfies the first four properties of Definition 1.

B. Transformers

Let $\mathcal{T}$ denote a set of transformers. A transformer $T \in \mathcal{T}$ maps an element from its *domain* to a set of elements in its *range*. Both the domain and the range of T are subsets of U.

For a transformer T, and an element x in the domain of T, $T(x)$ denotes the set of elements after transformation. For a set D of elements, we define

$$T(D) = \bigcup_{x \in D} T(x) \tag{8}$$

Assumption: For each transformer T, its inverse T^{-1} is a function from the range of T to the domain of T.

For an element $x \in U$, $T^{-1}(x)$ is *undefined* if x is not in the range of T.

C. Equivalence Relation for Sets of Transformers and Filters

Consider a network with a set, $\mathcal{T}$, of transformers as well as a set, $\mathcal{P}$, of predicates. We define a *new* ***packet equivalence relation*** for a network with both transformers and filters, which formalizes the intuition that two packets are equivalent if and

only if they are treated identically by every filter and by *every possible sequence of one or more transformers* in the network.

Definition 3 (Equivalence w.r.t. $\mathcal{P}$ and $\mathcal{T}$): Given a set $\mathcal{P}$ of predicates and a set $\mathcal{T}$ of transformers. Let $\{C_1, C_2, \ldots, C_n\}$ denote equivalence classes specified by the atomic predicates for $\mathcal{P}$.

Two elements x_1, x_2 in set U are equivalent w.r.t. $\mathcal{P}$ and $\mathcal{T}$ if and only if the following two conditions hold:

1) *$I_{C_i}(x_1) = I_{C_i}(x_2)$ for each $i \in \{1, \ldots, n\}$.*
2) *Either both $T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_1)$ and $T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_2)$ are undefined, or*

$$I_{C_i}(T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_1)) = I_{C_i}(T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_2)) \quad (9)$$

for each $i \in \{1, \ldots, n\}$, any positive integer k and any possible sequence $T_{\alpha_k} \ldots T_{\alpha_1}$ of transformers, $T_{\alpha_j} \in \mathcal{T}, j \in \{1, \ldots, k\}$.

Let $\{B_1, \ldots, B_l\}$ denote equivalence classes defined by the above equivalence relation (Definition 3). Note that elements x_1 and x_2 are equivalent w.r.t. $\mathcal{P}$ and $\mathcal{T}$ if and only if $x_1, x_2 \in B_j$, for some $j \in \{1, \ldots, l\}$.

Note: It is possible to define a less coarse equivalence relation w.r.t. $\mathcal{P}$ and $\mathcal{T}$ by replacing $\{C_1, C_2, \ldots, C_n\}$ in Definition 3 with a set of equivalence classes specified by representative predicates, instead of atomic predicates, for $\mathcal{P}$.

D. Atomic and Representative Predicates for Transformers and Filters

Definition 4 (Representative Predicates for $\mathcal{P}$ and $\mathcal{T}$): Given $\mathcal{P}$ and $\mathcal{T}$, the set, $\mathcal{B} = \{b_1, b_2, \ldots, b_l\}$, of predicates is representative of $\mathcal{P}$ and $\mathcal{T}$ if and only if $\mathcal{B}$ satisfies the following properties:

1) $\mathcal{B}$ is representative of $\mathcal{P}$.
2) For each $T \in \mathcal{T}$ and for each $b_i \in \mathcal{B}$ that is transformed by T, the following holds:

$$T(b_i) = \bigvee_{j \in S(b_i)} b_j, \quad \text{where } S(b_i) \subseteq \{1, \ldots, l\}. \quad (10)$$

Note that Property 2 in Definition 4 requires that, for each transform T in $\mathcal{T}$ and each predicate b_i in $\mathcal{B}$, if packets in b_i can be transformed by T, then the transformed predicate of b_i must be the disjunction of a subset of predicates in $\mathcal{B}$. This means that all packets specified by each predicate in $\mathcal{B}$ are treated identically by each transformer, i.e., *they are equivalent w.r.t. all transformers and can be represented as a single entity.* We can now define ***atomic predicates*** for sets of predicates and transformers.

Definition 5 (Atomic Predicates for $\mathcal{P}$ and $\mathcal{T}$): Given $\mathcal{P}$ and $\mathcal{T}$, the set of atomic predicates for them, $\mathcal{B} = \{b_1, b_2, \ldots, b_l\}$, satisfies the following properties:

1) $\mathcal{B}$ is representative of $\mathcal{P}$ and $\mathcal{T}$.
2) l is the minimum number such that the set $\{b_1, \ldots, b_l\}$ satisfies the above property.

In the Appendix, we proved the following: (i) Definition 3 defines the coarsest equivalence relation w.r.t. $\mathcal{P}$ and $\mathcal{T}$. (ii) The equivalence classes in Definition 3 are specified by atomic predicates in Definition 5. These results constitute Theorem 1.

Theorem 1: Given a set $\mathcal{P}$ of predicates and a set $\mathcal{T}$ of transformers, the atomic predicates for $\mathcal{P}$ and $\mathcal{T}$ (defined in Definition 5) specify the coarsest equivalence classes in the set U w.r.t. $\mathcal{P}$ and $\mathcal{T}$ (defined in Definition 3).

Advantages of Atomic Predicates: For real networks, each atomic predicate represents a very large number of *equivalent packets in many disjoint fragments of the packet space*. A predicate is equal to the disjunction of a subset of atomic predicates and is represented in APT by the integer identifiers of the atomic predicates; a packet transformer is represented by a set of mappings, each of which maps an integer to a set of integers (that is, from one atomic predicate to a subset of atomic predicates).

APT computes reachability trees for atomic predicates, each of which represents a very large number of packets with equivalent behavior, rather than for individual packets. Thus the use of atomic predicates reduces the time and space required for computing and storing these trees, as well as for verifying reachability properties, by orders of magnitude.

V. Algorithm Design

Consider a network with a set, $\mathcal{T}$, of transformers and a set, $\mathcal{P}$, of predicates (representing filters). Each transformer $T \in \mathcal{T}$ is modeled as a function that maps a predicate specifying its input packet set to another predicate specifying its output packet set. The input packet set is the *domain*, D, of T and the output packet set is the *range*, $T(D)$, of T in Equation (8). *The predicate specifying the input packet set of T is added to $\mathcal{P}$ if it is not already included.*

To design an algorithm for computing the network's atomic predicates, we make use of the following observations:

(i) For $T \in \mathcal{T}$, its input packet set, specified by $P \in \mathcal{P}$, $P \neq false$, is equal to the disjunction of a subset of representative predicates for $\mathcal{T}$ by Equation (7):

$$P = \bigvee_{i \in S(P)} b_i, \quad \text{where } S(P) \subseteq \{1, \ldots, l\}. \quad (11)$$

Thus the transformed predicate $T(P)$ is the following:

$$T(P) = \bigvee_{i \in S(P)} T(b_i), \quad \text{where } S(P) \subseteq \{1, \ldots, l\}. \quad (12)$$

(ii) We can use the formulas in Section III to compute the transformed predicate $T(b_i)$ where T is one of five different packet transformations.

(iii) For $T \in \mathcal{T}$, its input packet set consists of all packets that can pass through the transformer. However, not all packets in the input packet set are transformed by T. For examples: the entrance of an IP-in-IP tunnel only encapsulates packets with certain destination IP addresses; a MPLS router only switches labels for packets with MPLS headers. Therefore, for each transformer, the algorithm only needs to compute $T(b_i)$ for those representative predicates specifying packets that are changed by the packet transformer.

Notation: Given any set, $\mathcal{Q}$, of predicates, we use $\mathcal{A}(\mathcal{Q})$ to denote the set of atomic predicates for $\mathcal{Q}$, which can be computed using one of the algorithms in [26] and [28].

Algorithm 1 below computes the set of atomic predicates for $\mathcal{P}$ and $\mathcal{T}$.

Algorithm 1 for Computing Atomic Predicates After Adding Transformers

Input: a set $\mathcal{P}$ of predicates, a set $\mathcal{T}$ of transformers
Output: a set $\mathcal{B} = \{b_1, b_2, \ldots, b_l\}$ of predicates
1: $\mathcal{P}' \leftarrow \mathcal{P}, \mathcal{B} \leftarrow \mathcal{A}(\mathcal{P}')$
2: Compute the following set:

$$\mathcal{R} = \{T(b_i) | \text{ for each } T \in \mathcal{T}, \text{ and for each } b_i \in \mathcal{B} \text{ that is transformed by } T\}$$

3: **if** $\mathcal{B} = \mathcal{A}(\mathcal{P}' \cup \mathcal{R})$ **then**
4: **return** $\mathcal{B}$.
5: **else**
6: $\mathcal{P}' \leftarrow \mathcal{P}' \cup \mathcal{R}$, $\mathcal{B} \leftarrow \mathcal{A}(\mathcal{P}')$
7: **goto** line 2
8: **end if**

In line 1 of Algorithm 1, $\mathcal{B}$ is set to $\mathcal{A}(\mathcal{P})$. It is representative of $\mathcal{P}$ but not representative of $\mathcal{P}$ and $\mathcal{T}$ unless $\mathcal{T}$ is empty.

Line 2 computes the transformed predicate of each predicate in $\mathcal{B}$ if it can be transformed by T, for each T in $\mathcal{T}$.

In line 3, if the condition $\mathcal{B} = \mathcal{A}(\mathcal{P}' \cup \mathcal{R})$ is satisfied, it means that each transformed predicate in $\mathcal{R}$ is equal to the disjunction of a subset of predicates in $\mathcal{B}$, which is Property 2 in Definition 4. Property 1 of Definition 4 also holds because $\mathcal{B}$ is the set of atomic predicates for $\mathcal{P}'$ and $\mathcal{P}' \supseteq \mathcal{P}$ is an invariant of Algorithm 1. Therefore, the set $\mathcal{B}$ returned by Algorithm 1 is representative of $\mathcal{P}$ and $\mathcal{T}$ by Definition 4.

Additionally, we proved that the predicates returned by Algorithm 1 specify equivalence classes defined in Definition 3. Thus they are the atomic predicates for $\mathcal{P}$ and $\mathcal{T}$. These results are stated in Theorem 2 with proof in the Appendix.

Theorem 2: If the set U of all elements is finite, then Algorithm 1 will terminate and return the set of atomic predicates for the set $\mathcal{P}$ of predicates and set $\mathcal{T}$ of transformers.

Observations: (i) Transformers in the set $\mathcal{T}$ in Algorithm 1 can be of different types. (ii) Changing the ordering of transformers in $\mathcal{T}$ does not change the set of atomic predicates returned by Algorithm 1 but the computation time varies. (iii) Algorithm 1 can be used for adding more transformers to a network with existing predicates and transformers; in this case, the input set $\mathcal{P}$ must be the set of atomic predicates for the existing predicates and transformers.

Number of Iterations: Algorithm 1 requires different numbers of iterations for different types of packet transformers. A simple case such as "NATs only" requires *two iterations*. In the first iteration, the set $\mathcal{R}$ is computed. Each predicate in $\mathcal{R}$ is found to be atomic in line 3 and thus not transformed by any transformer in $\mathcal{T}$ in the second iteration. As a result, the set $\mathcal{R}$ computed in the second iteration is the same as that in the first iteration. Thus the termination condition $\mathcal{B} = \mathcal{A}(\mathcal{P}' \cup \mathcal{R})$ in line 3 is satisfied.

IP-in-IP tunnels (all variations) belong to the general case. However, Algorithm 1 only needs *three iterations* to compute the new set of atomic predicates after adding a set of IP-in-IP tunnels. The first iteration computes atomic predicates specifying packets allowed to enter each tunnel and the corresponding encapsulated packets. The second iteration computes atomic predicate(s) specifying packets not allowed to enter each tunnel. In the third iteration, the termination condition is satisfied and Algorithm 1 terminates.

A more general case, such as MPLS, may require more than 3 iterations (see experimental results in Section VI).

Sets of Predicates: For networks with two very different types of packet filters (e.g. derived from ACLs and forwarding tables), it is more efficient to represent them by two different sets of predicates [26], [28]. In this case, Algorithm 1 is run twice (once for each set of predicates with the set of transformers) to obtain two sets of atomic predicates.

TABLE I
STATISTICS OF THE THREE DATASETS

	Stanford	Internet2	Provider cone 1
No. of routers	16	9	51
No. of duplex ports	58	55	1,048

	Stanford		Internet2		Provider cone 1
No. of rules	IPv4	ACL	IPv4	MPLS	IPv4
	757,170	1,584	126,017	318	6,958,862

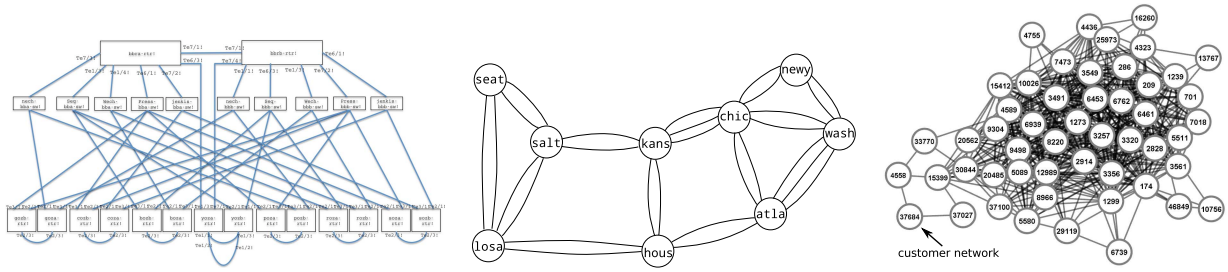
VI. PERFORMANCE EVALUATION

We implemented Algorithm 1 in APT for the three types of widely used packet transformations with the formulas presented in Section III. In APT, predicates are represented by binary decision diagrams (BDDs) which are rooted, directed acyclic graphs, with logical operations for packet transformations implemented by BDD operations [24]. For each packet transformer, the BDD specifying the set of transformed packets can be computed from the BDD specifying the set of input packets using highly efficient, optimized graph algorithms [7]. All results in this paper were computed using one core of a six-core Xeon E5-1650 processor with 12 MB of L3 cache and 16 GB of DRAM.

We evaluated the performance and scalability of APT using four datasets, namely: the Stanford dataset [3] and Internet2 dataset [4] from real networks, and two very large provider cone datasets we constructed from Internet open sources. Transformers used by NATs, IP-in-IP tunnels, and MPLS tunnels were added to these datasets. We performed two sets of experiments. The Stanford, Internet2, and the first provider cone datasets were used in the first set of experiments presented in subsections VI-A to VI-D. The second provider cone data set, presented in subsection VI-E, was used for a scalability analysis.

A. Datasets Used in the First Set of Experiments

Statistics of the three datasets used in the first set of experiments are shown in Table I. Topologies of the three datasets are shown in Figure 4. We downloaded the Stanford dataset [3] and the Internet2 dataset [4].



(a) (b) (c)

Fig. 4. Network topologies of three datasets. (The Stanford topology is from [13]. The middle row consists of switches that are not modeled [3]). (a) Stanford dataset. (b) Internet2 dataset. (c) Provider cone 1 dataset.

The Stanford dataset has 16 routers (in Figure 4(a), 2 backbone routers at the top connected to 14 zone routers at the bottom via 10 switches that are not modeled [3]). The 16 routers have a total of 58 duplex ports, 757,170 IPv4 forwarding rules, 1,584 ACL rules. Each router has an average of 3.6 duplex ports (neighbors).

The Internet2 dataset has 9 routers with a total of 55 duplex ports, 126,017 IPv4 forwarding rules, and 318 MPLS rules. Each router has an average of 6.1 duplex ports (neighbors).

To evaluate scalability, we constructed a large network dataset as follows: From the Internet topology for October 2013 created by CAIDA [1], we isolated the *provider cone* of a monitored tier-4 AS (37684) consisting of the direct and indirect Internet service providers of the tier-4 AS. In the dataset, each AS is represented by a router; two routers are connected by a link if the ASes they represent have a provider-customer or peer relationship. Routing tables were computed using control plane data for October 2013 from several open sources [2], [17], [19], [23]. Its topology in Figure 4(c) has 51 routers, representing 51 providers *including all 15 tier-1 ISPs*. The dataset has a total of 1,048 duplex ports and 6,958,862 IPv4 forwarding rules.[5] Each router has an average of 20.5 duplex ports (neighbors), i.e., the topology is very dense.

B. Transformers Added to the Datasets

NATs: In the experiments, NATs are added to each dataset connecting edge routers of the network to private subnets. Since we will compute reachability trees rooted at ports with public addresses only, we added to the datasets only NATs for translating public addresses to private subnet addresses. For each NAT added for a subnet, we use a different public IP address for the newly created port of the edge router and a different private prefix for the subnet.

IP-in-IP Tunnels: An IP-in-IP tunnel consists of an entry router and an exit router. At the entry router, packets allowed to enter the tunnel are encapsulated with a new IP header; at the exit router, the new IP header is de-encapsulated and the original packets are recovered. The exit router also filters out packets not allowed to enter the tunnel. *The entry and exit routers of an IP-in-IP tunnel have packet transformers*. (Thus two packet transformers are added for each tunnel added.) To create a tunnel for the Internet2 dataset, we randomly select three different routers. The first two routers are the entry and exit routers. The IP prefix of the third router is the destination address of packets encapsulated and de-encapsulated by the entry and exit routers, respectively. For the provider cone 1 data set, the entry router of each tunnel is the router representing the customer network (AS 37684). Each exit router is a router representing a non tier-1 AS. Each tunnel carries packets destined to a router representing a tier-1 AS.

MPLS Tunnels: When a packet travels along a MPLS tunnel, it is encapsulated by a MPLS header at the entry router, its MPLS label is changed at each transit router, and the MPLS header is de-encapsulated at the exit router. Thus *every router along the tunnel has a packet transformer*. The Internet2 dataset includes incomplete MPLS tunnel configurations. Each tunnel is missing the entry router and the exit router, which we added. The set of packets allowed to enter the tunnel is specified by the prefix of a router not in the MPLS path. We found a total of 28 distinct MPLS tunnels which use 109 transformers. The longest tunnel has five hops. On average, a MPLS tunnel in the Internet2 dataset has 2.9 hops.

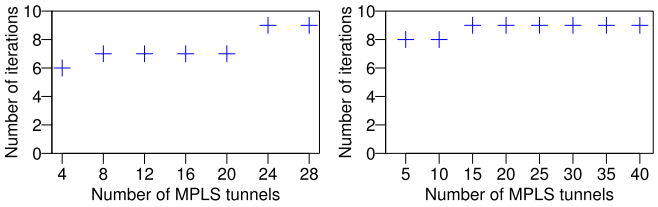
To create MPLS tunnels for the provider cone 1 dataset, we use the same entry and exit routers as the ones created for the IP-in-IP tunnels. For each pair of entry and exit routers, we compute a random path for the tunnel, and specify the set of packets allowed to enter the tunnel by the prefix of a router not in the MPLS path. We created 40 MPLS tunnels which use 176 transformers. The longest tunnel has five hops. On average, a MPLS tunnel in the dataset has 3.4 hops. (We used parameter values similar to Internet measurements [20]).

Header Stack and Auxiliary Variables: For all experiments presented in this paper, it was sufficient to use a header stack consisting of a MPLS header, followed by an IPv4 header, followed by the IPv4 header of the original packet. Each IP header has five relevant fields: source and destination IP addresses, source and destination port numbers, and protocol number. In the MPLS header, the MPLS label is the only relevant field. A predicate specifying a packet set is represented by a BDD with a total of 230 bit variables, including two auxiliary variables, one for the MPLS header, and the other for the outer IP header.

C. Computing Atomic Predicates

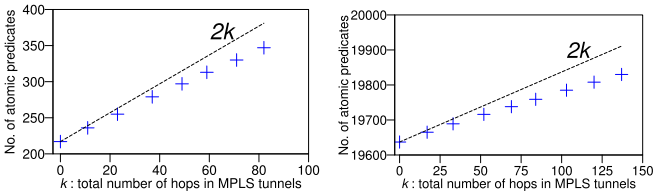
Number of Algorithm 1 Iterations: We have proved that after adding a set of NATs, Algorithm 1 requires two

[5]For each router, duplex ports connecting to routers outside of the graph are merged as one additional duplex port.



(a) (b)

Fig. 5. Number of Algorithm 1 iterations for networks with MPLS tunnels. (a) Internet2 dataset. (b) Provider cone 1 dataset.



(a) (b)

Fig. 6. Number of atomic predicates for networks with MPLS tunnels. (a) Internet2 dataset. (b) Provider cone 1 dataset.

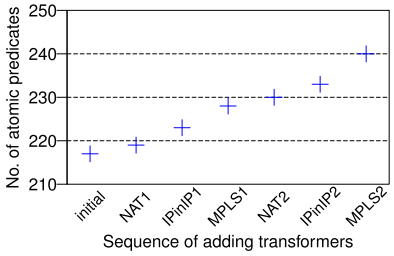


Fig. 7. Number of atomic predicates for a network with multiple transformer types (MPLS1 has two hops, MPLS2 has four hops).

iterations. Also, after adding a set of IP-in-IP tunnels, Algorithm 1 requires three iterations. We do not have an analytic bound for adding MPLS tunnels.

Figures 5(a) and (b) show the number of iterations used by Algorithm 1 versus the number of MPLS tunnels, for the Internet2 and provider cone 1 datasets, respectively. For each dataset, Algorithm 1 requires up to nine iterations in the experiments.

Number of Atomic Predicates: To illustrate, consider an IPv4 network. Allowing packets in the network to be encapsulated by just one IPv4 header increases the packet space size from 2^n to 2^{2n}, where $n = 32$ if only the destination IP address is used for filtering, or $n = 104$ if bits in the 5-tuple are used for filtering. Instead of individual packets, APT works with atomic predicates each of which represents a large equivalence class of packets. In this subsection, we studied how the number of atomic predicates increases when packet transformers are added to existing networks with filters only.

We make two empirical observations from the atomic predicates computed for our datasets: (i) After adding a set of transformers to an existing set of filters, the increase in the number of atomic predicates is dependent mainly on the number of packet transformers added, and not on the number of existing filters. (ii) For the three types of transformers added (described in subsection VI-B), the increase in the number of atomic predicates is ≤ 2 per transformer on the average.

To illustrate, consider the Internet2 dataset with 126,017 forwarding rules. The number of atomic predicates for filters only is 217. Consider the provider cone 1 dataset with 6,958,862 forwarding rules (including forwarding rules obtained from all 15 tier-1 ISPs), the number of atomic predicates for filters only is 19,636. Each MPLS tunnel of h hops has $h + 1$ transformers. Figures 6(a) and (b) show the number of atomic predicates for both filters and transformers versus the total number, k, of hops in MPLS tunnels, for the Internet2 and provider cone 1 datasets, respectively. In each figure, the reference line has a slope of 2. The number of atomic predicates grows at a rate of less than 2 per hop (or per transformer).

As another example, we computed atomic predicates for the Internet2 dataset with 2 NATs, 2 IPv4-in-IPv4 tunnels, and 2 MPLS tunnels. In particular, the dataset has an IP-in-IP tunnel nested inside a MPLS tunnel. Figure 7 shows the number of atomic predicates versus the sequence of transformers added. The increase in the number of atomic predicates is 2 for each NAT (1 transformer each), 3 for IP-in-IP1 and 4 for IP-in-IP2 (two transformers per tunnel), 5 for MPLS1 (3 transformers), and 7 for MPLS2 (5 transformers).

Time to Compute Atomic Predicates: We show the time used by Algorithm 1 to compute atomic predicates for all packet filters and transformers in each of the three datasets. The computation time results are presented in Figure 8 for NAT, Figure 9 for IPv4-in-IPv4, and Figure 10 for MPLS. The black portion of each bar in these figures represents the computation time for the initial set of atomic predicates for packet filters only. (These times are slightly higher than the corresponding times in [26] and [28] due to new packet filters added to guard the input ports of transformers.) The grey portion of each bar represents the additional computation time used by Algorithm 1 to process packet transformers to obtain the atomic predicates for both transformers and filters.

We found that Algorithm 1 is very fast. In Figures 8-10, observe that the vertical axis is in milliseconds for the Stanford and Internet2 datasets, and in seconds for the large ISP-scale provider cone 1 dataset. In almost all cases, the time for computing atomic predicates for packet filters only (line 1 of Algorithm 1) is much greater than the time for computing atomic predicates for the set of packet transformers. For a given network, once the atomic predicates have been computed for its initial topology, filtering rules, and transformers, the set of atomic predicates can be updated very quickly after dynamic network changes (see subsection VI-F).

Baseline Preprocessing Time: The network verification problem has a baseline preprocessing overhead that is necessary, irrespective of which verification tool is used, namely: the overhead to parse and convert forwarding tables and ACLs in network devices into data structures that represent them in the verification tool. Such baseline preprocessing overheads for other verification tools are rarely shown in prior papers on network verification.

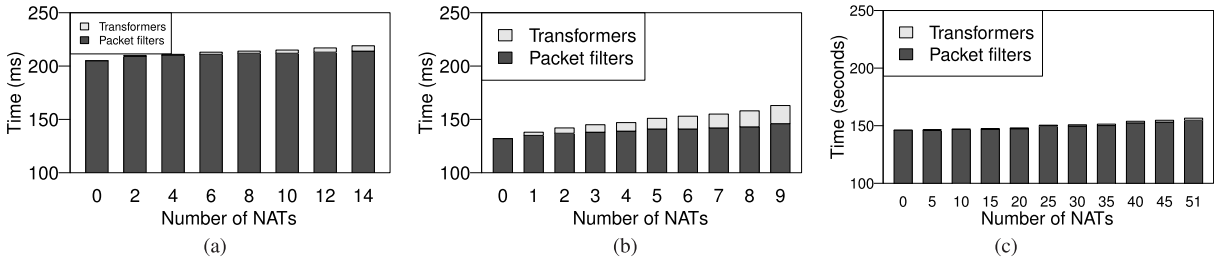


Fig. 8. Time to compute atomic predicates for networks with NATs. (a) Stanford dataset. (b) Internet2 dataset. (c) Provider cone 1 dataset.

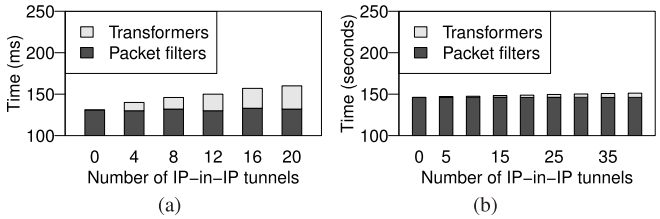


Fig. 9. Time to compute atomic predicates for networks with IP-in-IP tunnels. (a) Internet2 dataset. (b) Provider cone 1 dataset.

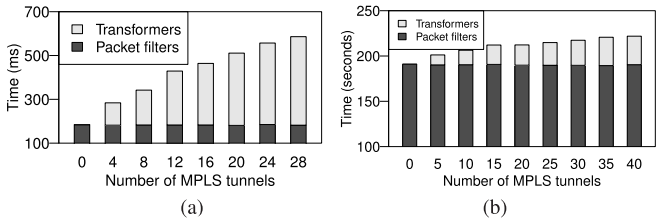


Fig. 10. Time to compute atomic predicates for networks with MPLS tunnels. (a) Internet2 dataset. (b) Provider cone 1 dataset.

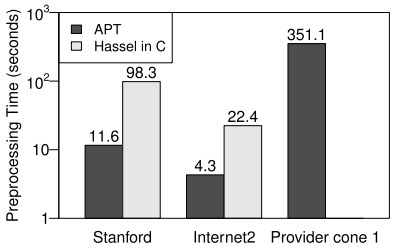


Fig. 11. Preprocessing time.

The preprocessing times of Hassel in C and APT are shown in Figure 11. Note that the vertical axis is in log scale. APT computes faster than Hassel by 8 and 5 times for the Stanford and Internet2 datasets, respectively. The forwarding tables in the provider cone 1 dataset are too large for the Hassel program. Therefore, only the preprocessing time of APT is shown for this dataset.

Observe that these preprocessing times are much larger than the times used by APT to compute atomic predicates in the previous subsection for the same networks. The comparison demonstrates that the computation of atomic predicates incurs negligible preprocessing overhead.

D. Verifying Network Properties

APT computes reachability trees for verifying network properties. Such a tree is rooted at a port in the network. Computing the reachability tree from a source port detects *forwarding loops* for all packets injected into the source port, if any. More generally, reachability trees can be used to verify safety and progress properties specified in a temporal logic (such as, CTL [10]): for example, verifying that all packets injected into the source port traverse a specified sequence of required waypoints in the network. Some optimization techniques for verifying properties are presented in [26] and [28].

How to Compute Reachability Trees: Consider a network of boxes interconnected by links. Each box with input ports and output ports is modeled as shown in Figure 1. A reachability tree is rooted at a port which is the source of packets to the tree.[6] The reachability tree consists of every path along which a set of packets can travel from the source port to another port in the network. Each node in the tree stores a port identifier and the set of packets that can reach the port from the source. The set of packets is represented by integer identifiers of atomic predicates. The same port may appear in multiple paths of the tree.

APT computes reachability trees in a graph whose nodes are ports. Within each box, all input ports are connected to all output ports. An output port of a box sends packets to the input port of another box connected by a data link. (It is possible that an input port receives packets from a box outside of the network and an output port sends packets to a box outside of the network.) Every packet set (also packet filter) is represented by a set of integers that identify atomic predicates. A packet transformer maps an atomic predicate to a subset of atomic predicates, i.e., an integer to a set of integers. As a result, reachability tree computation by APT is very fast.

Consider input port, $port_1$, of the box in Figure 1. Suppose a packet set P is injected into the port. The set of packets that can reach $port_1$ is $P \wedge A_1$. The sets of packets that can reach output ports, $port_3$ and $port_4$, are sets $T_1(P \wedge A_1) \wedge F_3$ and $T_4(T_1(P \wedge A_1) \wedge F_4)$, respectively. The sets of packets that are transmitted out of $port_3$ and $port_4$ are sets, $T_1(P \wedge A_1) \wedge F_3 \wedge A_3$ and $T_4(T_1(P \wedge A_1) \wedge F_4) \wedge A_4$, respectively.

The reachability tree from a port s to all other ports in the network is computed by performing a depth-first search which

[6]A port is an input or output port. We consider only forward reachability trees in this paper. The same model and data can be used to compute reverse reachability trees rooted at a destination port, which have other useful network management applications [27].

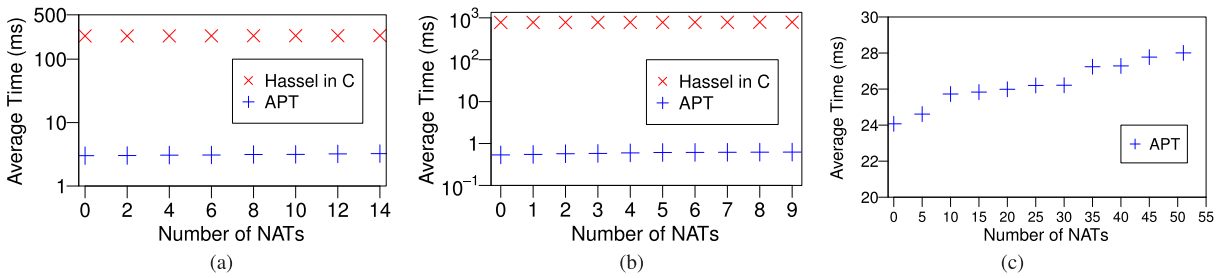


Fig. 12. Ave. time to compute reachability tree from one port (also for loop detection) for datasets with NATs. (a) Stanford dataset. (b) Internet2 dataset. (c) Provider cone 1 dataset.

begins by visiting port s. A search branch is terminated after visiting a port (say x) if one of the following conditions holds: (i) the set of packets that can reach x is empty; (ii) port x is an output port that is not connected to an input port of a box in the network; (iii) port x is an input port of a box with no output port (unexpected); (iv) port x has been visited before in the search (*loop detected*). In each of these cases, the search backtracks and depth-first search continues until no more port in the network can be reached. When depth-first search terminates, a reachability tree from port s to all other ports in the network is created.

We use the time to compute the reachability tree from a source port, averaged over all source ports, as the *benchmark* for performance evaluation and comparison. For the Stanford and Internet2 datasets, all ports are source ports. For the provider cone 1 dataset, only ports used by the customer AS to reach its direct providers are source ports.

NATs: Figures 12(a)-(c) show average tree computation times of Hassel in C and APT for the three datasets versus the number of NAT transformers. Forwarding tables of the provider cone 1 dataset are too big for Hassel in C. Therefore, the average tree computation time is shown only for APT in Figure 12(c).

Note that the vertical axes in Figures 12(a)-(b) are in log scale. Average time is measured in milliseconds for all 3 datasets (including provider cone 1) showing that network verification by APT is extremely fast. Also the verification time is not very sensitive to the number of NAT transformers. APT is faster than Hassel by 2 orders of magnitude for the Stanford dataset and by 3 orders of magnitude for the Internet2 dataset. *For the Stanford dataset, APT found the same 12 infinite forwarding loops found by Hassel in C.* This direct comparison provides validation for APT.

Hassel performs better for the Stanford dataset because after compression [9], its 757,170 forwarding rules are reduced to 3,840 forwarding rules. In comparison, the Internet2 dataset still has 77,451 forwarding rules after compression. We note that, for APT, the number of atomic predicates is the same, irrespective of whether forwarding tables are compressed or not.

Tunnels: Figures 13(a) and (b) show the average tree computation times of APT for the Internet2 and provider cone 1 datasets versus the number of tunnels for IP-in-IP and MPLS, respectively.

The vertical axes in Figures 13(a) and (b) are in log scale. Also average time is measured in milliseconds for both

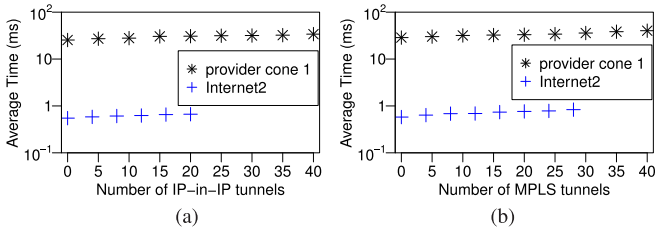


Fig. 13. Ave. time used by APT to compute reachability tree from one port (also for loop detection) for datasets with IP tunnels and MPLS tunnels. (a) IP tunnels. (b) MPLS tunnels.

datasets (including provider cone 1) showing that network verification by APT is extremely fast. Also the verification time is not very sensitive to the number of tunnels.

Memory Space Required: The memory requirement of APT is very low. For the Internet2 dataset with 28 MPLS tunnels, APT uses 11.44 MB to store predicates for filters, atomic predicates, and transformer mappings, and uses 10.94 MB to store all 55 trees. For the provider cone 1 dataset with 40 MPLS tunnels, APT uses 355.57 MB to store predicates for filters, atomic predicates, and transformer mappings, and uses 4.14 MB to store two trees.

E. A Scalability Analysis

To evaluate the performance of APT as the network size increases, we constructed a larger dataset for the provider cone of another customer AS (52941) from the same Internet topology and data collected during October 2013 used previously for AS 37684. For the two provider cones, we present their statistics in Table II, number of atomic predicates in Table III, time to compute atomic predicates in Table IV, average time to compute the reachability tree from one port in Table V, and memory space usage in Table VI. In the last column of each table, we show the ratio of the number for provider cone 2 to the number for provider cone 1 for each row.

Table II shows statistics of the two provider cones. The average number of neighbors per router is the number of duplex ports divided by the number of routers, which is 20.5 and 20.9 for provider cones 1 and 2, respectively. Thus the topology of each network is a dense mesh, which requires much more time and space for reachability analysis than sparse topologies.

Table III shows that, for packet filters only, the ratio of the numbers of atomic predicates is 1.42 indicating that the

TABLE II
STATISTICS OF THE TWO PROVIDER CONE DATASETS

	provider cone 1	provider cone 2	ratio
customer net	AS 37684	AS 52941	
# routers	51	92	1.80
# duplex ports	1,048	1,920	1.83
ave. # of neighbors per router	20.5	20.9	1.02
# rules	6,958,862	11,691,232	1.68
# tier-1 ISPs	15	15	

TABLE III
NUMBER OF ATOMIC PREDICATES

	provider cone 1	provider cone 2	ratio
for packet filters only	19,636	27,911	1.42
additional for 51 NATs	102	102	1.00
additional for 40 IP-in-IP	115	120	1.04
additional for 40 MPLS	194	191	0.98

TABLE IV
TIME TO COMPUTE ATOMIC PREDICATES (SECONDS)

	provider cone 1	provider cone 2	ratio
packet filters and 51 NATs	157	335	2.14
packet filters and 40 IP-in-IP	159	339	2.13
packet filters and 40 MPLS	222	477	2.15

TABLE V
AVE. TIME TO COMPUTE REACHABILITY TREE FROM ONE PORT (SECONDS)

	provider cone 1	provider cone 2	ratio
with 51 NATs	0.02800	0.07800	2.79
with 40 IP-in-IP	0.03420	0.06221	1.82
with 40 MPLS	0.04021	0.06542	1.63

TABLE VI
MEMORY SPACE USAGE (MBYTES)

	provider cone 1	provider cone 2	ratio
predicates, atomic predicates and 40 MPLS	355.57	739.42	2.08
one reachability tree (ave.)	2.07	4.45	2.15

percentage increase in the number of atomic predicates from provider cone 1 to provider cone 2 is less than the percentage increase of the number of ports (also predicates).

Tables IV and V, however, show that some of the ratios of the times to compute atomic predicates and the times to compute reachability trees exceed the ratio of the numbers of ports. It is because *these computation times depend on the number of rules as well as the number of ports*. From Table II, the ratio of the numbers of rules is 1.68.

Table VI shows that provider cone 2 used 739.42 MB to store all its predicates, atomic predicates, and 40 MPLS. Each reachability tree occupied 4.45 MB on the average. The ratios are 2.08 and 2.15, respectively. Space requirements depend on both the number of ports and number of rules.

Thus, *the number of rules and number of ports are the important factors that determine the scalability of network verification tools*. Suppose the impacts of ports and rules are multiplicative. The product of the two ratios in Table II is $1.83 \times 1.68 = 3.07$, which is substantially larger than all of the other ratios in Table III - VI (the worst ratio is 2.79 in Table V which is still well below 3.07). This observation indicates that *the computation time and space requirements of APT increase much more slowly than the increase in network size* (i.e., product of the two ratios for ports and rules).

The largest network in these datasets is provider cone 2 with 40 MPLS tunnels (170 transformers), 1920 duplex ports, and 11,691,232 rules. For this network dataset, the average time used to compute a reachability tree is 65.42 ms *using only one core* of a 6-core Xeon processor. Thus we can expect APT to be scalable to networks much larger than provider cone 2.

Furthermore, the computation of atomic predicates and reachability trees can be easily parallelized to make use of multiple cores. This is because each reachability tree is computed independently. As for the computation of atomic predicates, our current implementation uses an algorithm that adds predicates one at a time; however, atomic predicates can be computed in parallel by placing the initial predicates at leaf nodes of a binary tree and then computing tree nodes from their children in parallel as suggested in [26] and [28].

TABLE VII
AVERAGE TIME USED BY APT TO UPDATE A REACHABILITY TREE FOR INTERNET2

	Network with NATs	Network with IP tunnels	Network with MPLS tunnels
Link up	0.048 ms	0.055 ms	0.42 ms
Link down	0.0091 ms	0.0092 ms	0.022 ms
Box up	0.39 ms	0.46 ms	1.98 ms
Box down	0.059 ms	0.060 ms	0.11 ms

F. Handling Dynamic Changes

We next describe how APT handles events that change the network state.

Link/Box Status Change: The set of predicates and the set of atomic predicates are not changed by link up/down status change. However, reachability trees from source ports may be affected by a link status change. APT checks each reachability tree and updates it if needed. A box up event can be handled by processing link up events for all links connected to the box, and a box down event can be handled by processing link down events for all links connected to the box.

For evaluation, we used the 3 Internet2 datasets with one type of added transformers (9 NATs, 20 IP-in-IP tunnels, or 28 MPLS tunnels). Table VII shows the average time used by APT to update a reachability tree for a link up/down event and a box up/down event. For link up/down events, the average time to *update* a reachability tree is very small; it is smaller than the average time to compute a reachability tree (see Figures 12-13) by 1-2 orders of magnitude.

Addition/Deletion of Transformers: When a NAT or a tunnel is removed from the network, the existing set $\mathcal{B}$ of predicates is still a *representative* set and can be used for reachability verification (the number of predicates in the set may not be smallest). APT recomputes all reachability trees but not the atomic predicates.

When a NAT or a tunnel is added to the network, APT first updates the set of atomic predicates and then recomputes all reachability trees.

TABLE VIII
AVERGE TIME USED BY APT TO UPDATE ATOMIC PREDICATES AFTER ADDING A NAT OR A TUNNEL

	Network with NATs	Network with IP tunnels	Network with MPLS tunnels
Internet2	2.0 ms	6.1 ms	37.8 ms
Provider cone 1	0.065 sec	0.29 sec	2.7 sec

TABLE IX
AVERGE TIME USED BY APT TO UPDATE ATOMIC PREDICATES AFTER A PREDICATE CHANGE

	Network with NATs	Network with IP tunnels	Network with MPLS tunnels
Internet2	2.11 ms	2.32 ms	3.89 ms
Provider cone 1	0.273 sec	0.278 sec	0.280 sec

For evaluation, we used the 3 Internet2 datasets described above and also 3 provider cone 1 datasets (with 51 NATs, 40 IP-in-IP tunnels, or 40 MPLS tunnels). Algorithm 1 is first used to compute atomic predicates for all NATs (or all tunnels) except one. The last one is then added and we measured the average time for Algorithm 1 to update the atomic predicates after adding the last NAT or tunnel. Table VIII shows the average time to update the set of atomic predicates for the Internet2 and provider cone 1 datasets.

Rule Updates: When a rule is inserted into, or deleted from, a forwarding table (or ACL), it may change one or more predicates for some filters. We did some experiments using the Internet2 dataset and found that 44% of rule insertions and deletions do not change any predicate. However, if a predicate is changed by rule updates, APT first updates the set of atomic predicates and then recomputes all reachability trees.

We evaluated the time to update the set of atomic predicates after a predicate change for the 6 Internet2 and provider cone 1 datasets. A predicate change is represented by deleting an existing predicate and adding a new predicate. The average time to update the set of atomic predicates after one predicate change is shown in Table IX for the six datasets.

Observation: Updating the set of atomic predicates after an incremental change (adding a NAT/tunnel or a predicate change) is very fast, much faster than computing the atomic predicates from scratch (see Figures 8-10 for comparison).

VII. CONCLUSIONS

Towards scalable verification of packet networks with transformers and filters, we conceived and formally defined a novel packet equivalence relation. Our transformer model is general, including most transformers used in real networks. We also define atomic predicates which specify the coarsest equivalence classes of packets in the packet space. We built a verification tool, APT, based upon a new algorithm for computing atomic predicates for networks with both packet transformers and filters.

For real networks, an atomic predicate typically represents equivalent packets in a large number of disjoint fragments of the packet space. In APT, each packet filter is represented by a set of integers (identifiers of atomic predicates). Each transformer is represented by a set of mappings, each of which maps an integer to a set of integers. By representing a very large set of equivalent packets by a single integer, the use of atomic predicates reduces the computation time and space required for network verification by orders of magnitude.

Our experimental results for the two provide cone datasets with very large numbers of routers, ports, rules, and tunnels demonstrate that APT is scalable to large networks, such as, ISP networks and large-scale datacenter infrastructures. We also experimented and found that APT recovers quickly from network changes including link/box status change, addition/removal of a NAT or tunnel, and rule updates.

We make two more observations: (1) The performance evaluation results were computed using only one core of a 6-core processor. The computation of atomic predicates and reachability trees can be easily parallelized to make use of multiple cores. (2) We consider only forward reachability trees in this paper. The same model and data can be used to compute *reverse reachability trees* rooted at destination ports, which have other useful network management applications [27].

APPENDIX
PROOFS OF THEOREMS

We prove two theorems. Given a set U of elements, a set $\mathcal{P}$ of predicates, and a set $\mathcal{T}$ of transformers, Theorem 1 states that the atomic predicates for $\mathcal{P}$ and $\mathcal{T}$ (defined in Definition 5) specify the coarsest equivalence classes in the set of all elements w.r.t. $\mathcal{P}$ and $\mathcal{T}$ (defined in Definition 3).

Theorem 2 states that, if the set U of all elements is finite, then Algorithm 1 will terminate and return the set of atomic predicates for $\mathcal{P}$ and $\mathcal{T}$.

A. Lemmas

The following three lemmas are useful in proofs to follow.

Lemma 1: Consider a transformer $T \in \mathcal{T}$, an element $x \in U$, and a set D of elements. If $T^{-1}(x)$ is defined, we have

$$I_{T(D)}(x) = I_D(T^{-1}(x)) \tag{13}$$

Proof: Let $y = T^{-1}(x)$.

If $y \in D$, then $x \in T(y) \subseteq T(D)$. As a result, $I_{T(D)}(x) = I_D(T^{-1}(x)) = 1$.

If $y \notin D$, then $x \in T(y)$ and $T(y) \cap T(D) = \emptyset$. As a result, $I_{T(D)}(x) = I_D(T^{-1}(x)) = 0$. □

Lemma 2: Consider any set of predicates $\{q_1, \ldots, q_m\}$ that is representative of the set $\mathcal{P}$ of predicates. Let $\{D_1, \ldots, D_m\}$ be the collection of sets such that q_i specifies D_i, $i \in \{1, \ldots, m\}$. Let x_1 and x_2 be two elements in U, and T be any transformer in $\mathcal{T}$.

Then the following two properties are equivalent:

P1. *For each $i \in \{1, \ldots, m, \}$, $I_{T(D_i)}(x_1) = I_{T(D_i)}(x_2)$.*

P2. *For each $i \in \{1, \ldots, m, \}$, either both $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are undefined, or $I_{D_i}(T^{-1}(x_1)) = I_{D_i}(T^{-1}(x_2))$.*

Proof (P2 implies P1): We first show it is not possible that only one of $T^{-1}(x_1)$ and $T^{-1}(x_2)$ is defined. Without loss of generality, assume that $T^{-1}(x_1) = y_1$ and $T^{-1}(x_2)$ is not defined. There exists $j \in \{1, \ldots, m\}$ such that $y_1 \in D_j$. Therefore, $x_1 \in T(y_1) \subseteq T(D_j)$ and $x_2 \notin T(D_j)$. So $I_{T(D_j)}(x_1) = 1$ and $I_{T(D_j)}(x_2) = 0$, which is a contradiction.

If both $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are undefined, then property *P2* holds.

If both $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are defined, $I_{D_i}(T^{-1}(x_1)) = I_{D_i}(T^{-1}(x_2))$ by Lemma 1 and property *P2* also holds.

Proof (P2 implies P1):

If both $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are undefined, $I_{T(D_i)}(x_1) = I_{T(D_i)}(x_2) = 0$.

If both $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are defined, $I_{T(D_i)}(x_1) = I_{T(D_i)}(x_2)$ by Lemma 1. □

Lemma 3: If x_1 and x_2 are equivalent w.r.t. $\mathcal{P}$ and $\mathcal{T}$, then for any $T \in \mathcal{T}$, either both $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are undefined, or $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are also equivalent w.r.t. $\mathcal{P}$ and $\mathcal{T}$.

Lemma 3 follows directly from Definition 3.

B. Atomic Predicates Specify the Coarsest Equivalence Classes With Respect to $\mathcal{P}$ and $\mathcal{T}$

Theorem 1: Given a set $\mathcal{P}$ of predicates and a set $\mathcal{T}$ of transformers, the atomic predicates for $\mathcal{P}$ and $\mathcal{T}$ (defined in Definition 5) specify the coarsest equivalence classes in the set U w.r.t. $\mathcal{P}$ and $\mathcal{T}$ (defined in Definition 3).

Proof Outline: Theorem 1 follows from Lemmas 4 and 5. Lemma 4 proves that the predicates that specify the equivalence classes in Definition 3 are representative predicates for $\mathcal{P}$ and $\mathcal{T}$ in Definition 4. Lemma 5 proves that the equivalence classes in Definition 3 are the coarsest equivalence classes. Therefore, they are specified by the smallest set of representative predicates for $\mathcal{P}$ and $\mathcal{T}$ which, by Definition 5, is the set of atomic predicates for $\mathcal{P}$ and $\mathcal{T}$.

Lemma 4: Given a set $\mathcal{P}$ of predicates and a set $\mathcal{T}$ of transformers, the predicates that specify equivalence classes $\{B_1, \ldots, B_l\}$ w.r.t. $\mathcal{P}$ and $\mathcal{T}$ in Definition 3 satisfy the two properties in Definition 4 for representative predicates.

Proof: Condition 1 in Definition 3 guarantees that the set of predicates specifying equivalence classes $\{B_1, \ldots, B_l\}$ is representative of $\mathcal{P}$. Hence the first property in Definition 4 is satisfied.

To prove the second property in Definition 4, consider two elements, x_1 and x_2 that are equivalent w.r.t. $\mathcal{P}$ and $\mathcal{T}$ (that is, $x_1, x_2 \in B_j$ for some $j \in \{1, \ldots, l\}$). For any $i \in \{1, \ldots, l\}$ and any transformer $T \in \mathcal{T}$, if both $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are not defined, then $I_{T(B_i)}(x_1) = I_{T(B_i)}(x_2) = 0$.

If both $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are defined, then by Lemma 1, we have

$$\begin{aligned} I_{T(B_i)}(x_1) &= I_{B_i}(T^{-1}(x_1)) \\ I_{T(B_i)}(x_2) &= I_{B_i}(T^{-1}(x_2)) \end{aligned} \tag{14}$$

In this case, equivalence of x_1 and x_2 implies equivalence of $T^{-1}(x_1)$ and $T^{-1}(x_2)$ by Lemma 3. Hence, we have $I_{B_i}(T^{-1}(x_1)) = I_{B_i}(T^{-1}(x_2))$ which together with equation (14) prove that $I_{T(B_i)}(x_1) = I_{T(B_i)}(x_2)$. Therefore, for any i, $j \in \{1, \ldots, l\}$, and for any $T \in \mathcal{T}$, we have, for the indicator function, $I_{T(B_i)}(B_j) = I_{T(B_i)}(x), x \in B_j$. As a result, we have, for any $i \in \{1, \ldots, l\}$ and for any $T \in \mathcal{T}$

$$T(B_i) = \bigcup_{\substack{I_{T(B_i)}(B_j)=1 \\ j \in \{1,\ldots,l\}}} B_j \tag{15}$$

Thus the second property in Definition 4 is also satisfied by predicates that specify $\{B_1, \ldots, B_l\}$. □

The following lemma proves that the equivalence classes w.r.t. $\mathcal{P}$ and $\mathcal{T}$ in Definition 3 are the coarsest equivalence classes.

Lemma 5: For a set $\mathcal{P}$ of predicates and a set $\mathcal{T}$ of transformers, let $\{B_1, \ldots, B_l\}$ denote equivalence classes w.r.t. $\mathcal{P}$ and $\mathcal{T}$, and $\{C_1, \ldots, C_n\}$ denote equivalence classes w.r.t. $\mathcal{P}$ in Definition 3. Consider any set of predicates $\{q_1, \ldots, q_m\}$ that satisfy the properties in Definition 4 for representative predicates. Let $\{D_1, \ldots, D_m\}$ be the collection of sets such that q_i specifies D_i, $i \in \{1, \ldots, m\}$. Then for all $i \in \{1, \ldots, m\}$, there exists a unique $j \in \{1, \ldots, l\}$ such that $B_j \supseteq D_i$. This implies that $m \geq l$ which is minimum.

Proof: To prove this lemma, it is sufficient to prove the following statement:

S1: For any $x_1, x_2 \in U$, if $x_1, x_2 \in D_i$ for some $i \in \{1, \ldots, m\}$, then x_1 and x_2 are equivalent w.r.t. $\mathcal{P}$ and $\mathcal{T}$.

If **S1** is true, then there exists a unique $j \in \{1, \ldots, l\}$ such that $x_1, x_2 \in B_j$. As a result, $B_j \supseteq D_i$ implying that $m \geq l$ which is minimum. We proceed to prove that **S1** holds by showing that for any $x_1, x_2 \in D_i$, for some $i \in \{1, \ldots, m\}$, x_1 and x_2 satisfy both conditions of Definition 3. Thus x_1 and x_2 are equivalent w.r.t. $\mathcal{P}$ and $\mathcal{T}$ by definition.

To prove condition 1 in Definition 3, we use [26, Lemma 2] or [28, Lemma 2], which shows that for each $g \in \{1, \ldots, m\}$, there exists a unique $j \in \{1, \ldots, n\}$, such that $D_g \subseteq C_j$. We also have, $I_{D_g}(x_1) = I_{D_g}(x_2)$ for $x_1, x_2 \in D_g$, for each $g \in \{1, \ldots, m\}$ because x_1 and x_2 are equivalent w.r.t. $\mathcal{P}$. This property implies that $I_{C_h}(x_1) = I_{C_h}(x_2)$ for each $h \in \{1, \ldots, n\}$. Thus condition 1 in Definition 3 is satisfied.

To prove that condition 2 in Definition 3 is satisfied, it is sufficient to prove the following statement:

S2: Either both $T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_1)$ and $T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_2)$ are undefined, or

$$I_{D_g}(T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_1)) = I_{D_g}(T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_2)) \tag{16}$$

for each $g \in \{1, \ldots, m\}$, any integer k and any possible sequence $T_{\alpha_k} \ldots T_{\alpha_1}$ of transformers.

Observe that if both $T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_1)$ and $T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_2)$ are undefined, then condition 2 in Definition 3 is satisfied. If equation (16) holds, then both $T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_1)$ and $T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_2)$ are in D_r, for some $r \in \{1, \ldots, m\}$. Since $D_r \subseteq C_j$ for some $j \in \{1, \ldots, n\}$, both $T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_1)$ and $T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_2)$ are in C_j. Therefore, we have

$$I_{C_h}(T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_1)) = I_{C_h}(T_{\alpha_k}^{-1} \cdots T_{\alpha_1}^{-1}(x_2)) \tag{17}$$

for each $h \in \{1, \ldots, n\}$, any integer k and any possible sequence $T_{\alpha_k} \ldots T_{\alpha_1}$ of transformers. Thus condition 2 in Definition 3 is satisfied.

We next prove* S2 *by induction on k, the length of the sequence of transformers. Note that we assume $x_1, x_2 \in D_i$, for some $i \in \{1, \ldots, m\}$.

Base Case ($k = 1$): For each $D_g \in \{D_1, \ldots, D_m\}$ and $T \in \mathcal{T}$, $T(D_g)$ can be represented by the union of a subset of $\{D_1, \ldots, D_m\}$. There are two possibilities in the union representation of $T(D_g)$. First, D_i containing x_1 and x_2 appears in the subset representing $T(D_g)$, in which case,

$I_{T(D_g)}(x_1) = I_{T(D_g)}(x_2) = 1$. Second, D_i does not appear in the subset representing $T(D_g)$, in which case, $I_{T(D_g)}(x_1) = I_{T(D_g)}(x_2) = 0$.

Therefore, $I_{T(D_g)}(x_1) = I_{T(D_g)}(x_2)$, for each $g \in \{1, \ldots, m\}$ and each $T \in \mathcal{T}$. The base case follows from Lemma 2.

Induction Case: Assume that for $k = h$, either both $T^{-1}_{\alpha_h} \cdots T^{-1}_{\alpha_1}(x_1)$ and $T^{-1}_{\alpha_h} \cdots T^{-1}_{\alpha_1}(x_2)$ are undefined, or

$$I_{D_g}(T^{-1}_{\alpha_h} \cdots T^{-1}_{\alpha_1}(x_1)) = I_{D_g}(T^{-1}_{\alpha_h} \cdots T^{-1}_{\alpha_1}(x_2)) \quad (18)$$

for each $g \in \{1, \ldots, m\}$, and any possible sequence $T_{\alpha_h} \ldots T_{\alpha_1}$ of transformers.

For $k = h + 1$, consider any sequence of length $h + 1$: $T^{-1}_{\alpha_{h+1}}, \ldots, T^{-1}_{\alpha_1}$. From the induction assumption, we know that either (i) both $T^{-1}_{\alpha_h} \cdots T^{-1}_{\alpha_1}(x_1)$ and $T^{-1}_{\alpha_h} \cdots T^{-1}_{\alpha_1}(x_2)$ are undefined, or (ii) $y_1 = T^{-1}_{\alpha_h} \cdots T^{-1}_{\alpha_1}(x_1)$ and $y_2 = T^{-1}_{\alpha_h} \cdots T^{-1}_{\alpha_1}(x_2)$.

In case (i), both $T^{-1}_{\alpha_{h+1}} T^{-1}_{\alpha_h} \cdots T^{-1}_{\alpha_1}(x_1)$ and $T^{-1}_{\alpha_{h+1}} T^{-1}_{\alpha_h} \cdots T^{-1}_{\alpha_1}(x_2)$ are undefined. Thus this case holds for $k = h + 1$.

In case (ii), we have $I_{D_g}(y_1) = I_{D_g}(y_2)$ for each $g \in \{1, \ldots, m\}$. $T_{\alpha_{h+1}}(D_g)$ can be represented by the union of a subset of $\{D_1, \ldots, D_m\}$. Using the same argument as the one for the base case, we have

$$I_{T_{\alpha_{h+1}}(D_g)}(y_1) = I_{T_{\alpha_{h+1}}(D_g)}(y_2),$$

for each $g \in \{1, \ldots, m\}$. From Lemma 2, we have either $T^{-1}_{\alpha_{h+1}}(y_1)$ and $T^{-1}_{\alpha_{h+1}}(y_2)$ are undefined, or $I_{D_g}(T^{-1}_{\alpha_{h+1}}(y_1)) = I_{D_g}(T^{-1}_{\alpha_{h+1}}(y_2))$. Thus this case holds for $k = h + 1$.

We have proved by induction on k that **S2** is true. Therefore, for any $x_1, x_2 \in D_i$, for some $i \in \{1, \ldots, m\}$, condition 2 of Definition 3 is satisfied. Thus we have proved that **S1** is true and the lemma is proved. □

C. Correctness and Termination of Algorithm 1

Proof Outline: Consider a network where h is the maximum number of transformations that can be applied to any packet injected into the network. By ***induction*** on h, we prove Lemma 6, which states that the set of predicates computed by the *h*th iteration of Algorithm 1 satisfies the equivalence relation, R_h, stated in Lemma 6, for $h \geq 1$. Lemma 7 states that Algorithm 1 returns the set of atomic predicates for $\mathcal{P}$ and $\mathcal{T}$. Termination follows from the assumption that set U is finite. Theorem 2 is thus proved.

Lemma 6: Given a set $\mathcal{P}$ of predicates and a set $\mathcal{T}$ of transformers. Let $\{C_1, C_2, \ldots, C_n\}$ denote equivalence classes specified by the atomic predicates for $\mathcal{P}$. At the hth iteration, Algorithm 1 computes a set of predicates that satisfies the following equivalence relation, R_h:

Two elements x_1 and x_2 are equivalent w.r.t. R_h if and only if the following two conditions hold

1) *$I_{C_i}(x_1) = I_{C_i}(x_2)$ for each $i \in \{1, \ldots, n\}$.*
2) *Either both $T^{-1}_{\alpha_g} \cdots T^{-1}_{\alpha_1}(x_1)$ and $T^{-1}_{\alpha_g} \cdots T^{-1}_{\alpha_1}(x_2)$ are undefined, or*

$$I_{C_i}(T^{-1}_{\alpha_g} \cdots T^{-1}_{\alpha_1}(x_1)) = I_{C_i}(T^{-1}_{\alpha_g} \cdots T^{-1}_{\alpha_1}(x_2)) \quad (19)$$

for each $i \in \{1, \ldots, n\}$, any integer $g \leq h$ and any possible sequence $T_{\alpha_g} \ldots T_{\alpha_1}$ of transformers, $T_{\alpha_j} \in \mathcal{T}$, $j \in \{1, \ldots, g\}$.

(Note: The above definition is the same as Definition 3 but with the number of transformations bounded by h.)

Proof: We prove this lemma ***by induction on*** h.

Base Case ($h = 1$): According to the algorithm, the set of predicates computed specifies the following equivalence relation.

Elements x_1 and x_2 are equivalent if and only if

1) $I_{C_i}(x_1) = I_{C_i}(x_2)$ for each $i \in \{1, \ldots, n\}$.
2) $I_{T(C_i)}(x_1) = I_{T(C_i)}(x_2)$ for each $i \in \{1, \ldots, n\}$, and each $T \in \mathcal{T}$.

From Lemma 2, the second condition is equivalent to: for each $i \in \{1, \ldots, n\}$ and each $T \in \mathcal{T}$, either both $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are undefined, or $I_{C_i}(T^{-1}(x_1)) = I_{C_i}(T^{-1}(x_2))$. Thus the above two conditions satisfy R_1 and the base case is proved.

Induction Case: Assume that the lemma is true for $h = f$. The fth iteration computes predicates that satisfy the equivalence relation, R_f, and specify equivalence classes, $E_1, \ldots, E_r$. The following properties are equivalent.

- Elements x_1 and x_2 are equivalent w.r.t. R_f.
- $I_{E_i}(x_1) = I_{E_i}(x_2)$ for each $i \in \{1, \ldots, r\}$.

Consider the $(f+1)$st iteration of Algorithm 1. In line 2 of the algorithm, set $\mathcal{R}$ is computed to be

$$\mathcal{R} = \{T(E_i) | \text{ for each } T \in \mathcal{T}, \text{ and for each } i \in \{1, \ldots, r\} \text{ that is transformed by } T\}.$$

$\{E_1, E_2, \ldots, E_r\}$ is the set of atomic predicates for P' in line 3. Therefore, the set of atomic predicates computed in line 3 is $\mathcal{A}(\mathcal{P}' \cup \mathcal{R}) = \mathcal{A}(\mathcal{A}(\mathcal{P}') \cup \mathcal{R}) = \mathcal{A}(\{E_1, \ldots, E_r\} \cup \mathcal{R})$.

By [28, Th. 2] (or [26, Th. 1]), $\mathcal{A}(\{E_1, \ldots, E_r\} \cup \mathcal{R})$ satisfies the following equivalence relation: two elements x_1 and x_2 are equivalent if and only if

1) $I_{E_i}(x_1) = I_{E_i}(x_2)$ for each $i \in \{1, \ldots, r\}$.
2) $I_{T(E_i)}(x_1) = I_{T(E_i)}(x_2)$ for each $i \in \{1, \ldots, r\}$, and each $T \in \mathcal{T}$.

From Lemma 2, the above equivalence relation is the same as: two elements x_1 and x_2 are equivalent if and only if

1) $I_{E_i}(x_1) = I_{E_i}(x_2)$ for each $i \in \{1, \ldots, r\}$.
2) Either both $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are undefined, or $I_{E_i}(T^{-1}(x_1)) = I_{E_i}(T^{-1}(x_2))$, for each $i \in \{1, \ldots, r\}$ and each $T \in \mathcal{T}$.

The first condition means that x_1 and x_2 are equivalent w.r.t. R_f; the second condition means that either both $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are undefined, or $T^{-1}(x_1)$ and $T^{-1}(x_2)$ are equivalent w.r.t. R_f. Since the second condition holds for $h = f+1$, x_1 and x_2 are equivalent w.r.t. R_{f+1}, for $g \leq f+1$ in Equation (19).

The case for $h = f+1$ is proved. Thus the lemma is proved by induction. □

Lemma 7: If Algorithm 1 terminates, it returns the set of predicates that specify the equivalence classes of Definition 3. Therefore, Algorithm 1 computes the set of atomic predicates for set $\mathcal{P}$ of predicates and set $\mathcal{T}$ of transformers.

Proof: If Algorithm 1 terminates at the first iteration, transformers in $\mathcal{T}$ do not transform any elements. The set of predicates returned by the algorithm is the set of atomic predicates for $\mathcal{P}$.

Assume that Algorithm 1 terminates at the hth iteration, $h > 1$. The set of predicates computed in the $(h-1)$th iteration is the same as the set computed in the hth iteration, which is the set of predicates returned. Therefore, the equivalence relation R_{h-1} of the $(h-1)$th iteration is the same as the equivalence relation R_h of the hth iteration. Suppose we keep running the algorithm, we have $R_{h-1} = R_k$, where $k \geq h$ is an arbitrary integer. Therefore, by Lemma 6, the set of predicates returned specifies the equivalence classes of Definition 3. □

Theorem 2: If the set U of all elements is finite, then Algorithm 1 will terminate and return the set of atomic predicates for the set $\mathcal{P}$ of predicates and set $\mathcal{T}$ of transformers.

Proof: In each iteration, the size of the set $\mathcal{B}$ of predicates computed strictly increases (except for the last iteration). The largest possible set for $\mathcal{B}$ is the set of all single elements. Therefore, Algorithm 1 will terminate and the theorem follows from Lemma 7. □

REFERENCES

[1] (Oct. 2013). *The CAIDA UCSD AS Relationships Dataset*. [Online]. Available: http://www.caida.org/data/as-relationships/

[2] (Oct. 2013). *Internet AS-level Topology Archive*. http://irl.cs.ucla.edu/topology/

[3] (Mar. 2013). *Header Space Library and NetPlumber*. [Online]. Available: https://bitbucket.org/peymank/hassel-public/

[4] (Oct. 2012). *The Internet2 Observatory Data Collections*. [Online]. Available: http://www.internet2.edu/observatory/archive/data-collections.html

[5] 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, Oct. 2009, pp. 123–132.

[6] 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.

[7] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986.

[8] R. Bush, O. Maennel, M. Roughan, and S. Uhlig, "Internet optometry: Assessing the broken glasses in Internet reachability," in *Proc. ACM IMC*, Chicago, IL, USA, 2009, pp. 242–253.

[9] R. P. Draves, C. King, V. Srinivasan, and B. D. Zill, "Constructing optimal IP routing tables," in *Proc. IEEE INFOCOM*, New York, NY, USA, Mar. 1999, pp. 88–97.

[10] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Cambridge, MA, USA: MIT Press, 1990.

[11] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," in *Proc. ACM SIGCOMM*, Chicago, IL, USA, 2014, pp. 27–38.

[12] P. Kazemian *et al.*, "Real time network policy checking using header space analysis," in *Proc. USENIX NSDI*, Lombard, IL, USA, 2013, pp. 99–111.

[13] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks," in *Proc. USENIX NSDI*, San Jose, CA, USA, 2012, pp. 113–126.

[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. 467–472.

[15] P. Nuno Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proc. USENIX NSDI*, Oakland, CA, USA, 2015, pp. 499–512.

[16] H. Mai *et al.*, "Debugging the data plane with anteater," in *Proc. ACM SIGCOMM*, Toronto, ON, Canada, 2011, pp. 290–301.

[17] (Oct. 2013). *Packet Clearing House*. [Online]. Available: https://www.pch.net/resources/data.php

[18] G. Plotkin, N. Bjørner, N. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *Proc. POPL*, Petersburg, FL, USA, 2016, pp. 69–83.

[19] RIPE NCC. (Oct. 2013). *RIS Raw Data*. [Online]. Available: http://www.ripe.net/data-tools/stats/ris/ris-raw-data

[20] J. Sommers, P. Barford, and B. Eriksson, "On the prevalence and characteristics of MPLS deployments in the open Internet," in *Proc. ACM IMC*, Berlin, Germany, 2011, pp. 445–462.

[21] S. Spinoso *et al.*, "Formal verification of virtual network function graphs in an sp-devops context," in *Proc. Eur. Conf. Service-Oriented Cloud Comput.*, 2015, pp. 253–262.

[22] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 314–327.

[23] U. O. Oregen. (Oct. 2013). *RouteView Project*. [Online]. Available: http://www.routeviews.org

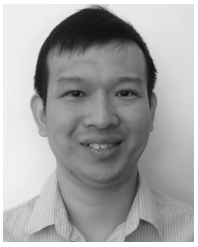
[24] A. Vahidi. (2004). *JDD, a Pure Java BDD and Z-BDD Library*. [Online]. Available: http://javaddlib.sourceforge.net/jdd/

[25] G. G. Xie *et al.*, "On static reachability analysis of IP networks," in *Proc. IEEE INFOCOM*, Miami, FL, USA, Mar. 2005, pp. 2170–2183.

[26] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," in *Proc. IEEE ICNP*, Göttingen, Germany, Oct. 2013, pp. 1–11.

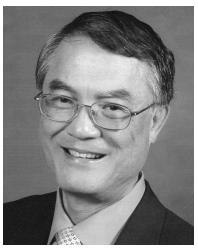
[27] H. Yang and S. S. Lam, "Collaborative verification of forward and reverse reachability in the Internet data plane," in *Proc. IEEE ICNP*, Raleigh, NC, USA, Oct. 2014, pp. 320–331.

[28] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 887–900, Feb. 2016.



Hongkun Yang received the B.S.E. (Hons.) and M.S.E. degrees from Tsinghua University in 2007 and 2010, respectively, and the Ph.D. degree from the Department of Computer Science, University of Texas at Austin, in 2015, where he was a recipient of the Microelectronics and Computer Development Fellowship.

He has published research papers in a number of conferences and journals, including the IEEE ICNP, the IEEE INFOCOM, the IEEE/ACM TRANSACTIONS ON NETWORKING, and the IEEE TRANSACTIONS ON MOBILE COMPUTING. His research interests include computer networks, protocol verification, network security, and formal methods.



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 (UCLA), in 1970 and 1974, respectively. From 1971 to 1974, he was a Postgraduate Research Engineer with the ARPA Network Measurement Center, UCLA, where he was involved in 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 currently a Professor and Regents Chair in computer science, and served as the Department Chair from 1992 to 1994.

He served as the 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, the IEEE TRANSACTIONS ON COMMUNICATIONS, the PROCEEDINGS OF THE IEEE, *Computer Networks*, and *Performance Evaluation*. He co-founded the ACM SIGCOMM Conference in 1983 and the IEEE International Conference on Network Protocols in 1993.

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