New Directions for Network Verification

Aurojit Panda[1], Katerina Argyraki[2], Mooly Sagiv[3], Michael Schapira[4], and Scott Shenker[5]

1 UC Berkeley, US
2 EPFL, CH
3 Tel Aviv University, IL
4 Hebrew University of Jerusalem, IL
5 UC Berkeley and ICSI, US

Abstract

Network verification has recently gained popularity in the programming languages and verification community. Much of the recent work in this area has focused on verifying the behavior of simple networks, whose actions are dictated by static, immutable rules configured ahead of time. However, in reality, modern networks contain a variety of middleboxes, whose behavior is affected both by their configuration and by mutable state updated in response to packets received by them. In this position paper we critically review recent progress on network verification, propose some next steps towards a more complete form of network verification, dispel some myths about networks, provide a more formal description of our approach, and end with a discussion of the formal questions posed to this community by the network verification agenda.

1998 ACM Subject Classification C.2.6 Internetworking, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Middleboxes, Network Verification, Mutable Dataplane

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.209

1 Introduction

Verification – by which we mean the general practice of checking the correctness of a computer-based system before it is put into use – was first developed to check the correctness of hardware, and is now increasingly used in the software development process. While networks have been around for many decades and are now an essential piece of our computational infrastructure, only recently has verification been applied to ensure their correctness.[1] As a result, there is now a growing literature on systems that can verify that the current or proposed network configuration (as represented by router forwarding tables) obey various important invariants (such as no routing loops or dead-ends). These systems – which allow network operators to verify that their networks will operate correctly, in terms of some well-defined invariants – represent a valuable, and long overdue, step forward for networking, which for too long was satisfied with not only *best-effort* service but also *best-guess* configuration. In this position paper we critically review this recent progress, propose some next steps towards a more complete form of network verification, dispel some myths about networks, provide a more formal description of our approach, and end with a discussion of the formal questions posed to this community by the network verification agenda.

[1] There has been much work on verifying network protocols and their implementations, but until recently almost none on verifying a given network configuration.

1st Summit on Advances in Programming Languages (SNAPL'15).
Eds.: Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett; pp. 209–220
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In the rest of this section, we provide some necessary background on networks and the current verification techniques.

The application of verification to the forwarding behavior of networks coincided with the rise of Software-Defined Networking (SDN)[2]. While not essential for the use of verification in networks, SDN provides a useful platform on which to deploy these tools so we discuss verification within the SDN context. Networks are comprised of two planes: the *data plane* which decides how packets are handled locally by each router (based on the local forwarding state and other information, such as state generated by previous packets), and the *control plane* which is a global process that computes and updates the local forwarding state in each router. In legacy networks, both planes are implemented in routers (with the data plane being the forwarding code or datapath, and the control plane being the global routing algorithm), but in SDN there is a clean separation between the two planes. The SDN control plane is logically centralized, and implemented in a few servers (called controllers) that compute and then install the necessary forwarding state. SDN-controlled routers only implement the data plane, executing a very simple datapath (OpenFlow [14]) in which the routing state is a set of $\langle \mathit{match}, \mathit{action} \rangle$ flow entries: all packets with headers matching the *match* entry are subject to the specified *action* which is often either to forward out a specific port (perhaps with a slightly modified header) or to drop the packet. Each router in the network is configured with a table containing flow entries (henceforth referred to as the router's *configuration*), and the network configuration is the set of flow tables for all switches and routers in the network.

The first wave of verification tools [12, 11, 10, 9] analyzed the global behavior of a network made up of switches obeying this simple forwarding model. As a packet travels through the network, its next-hop is dictated by the routing state in the current router; thus, this network-wide behavior can be thought of as the composition of the routing state in each router. These early verification tools would take a snapshot of network state (either that which is already in the network, or that which the control is poised to insert into the network) and then verify whether some basic invariants held. These invariants (which are specified by the network operator) are typically quite simple and few in number: reachability (*e.g.,* packets from host A can reach host B), isolation (*e.g.,* packets from host A cannot reach host B), loop-freedom (no packet enters into an infinite loop), and no dead-ends (no packet arrives at a router which cannot forward it to another router or to the end-destination). Subsequent network verification tools (*e.g.,* [5, 2, 17, 18]) make the same assumptions about the datapath, but generalize along various other dimensions.

All of these tools leverage the fact that the simple forwarding model renders the datapath *immutable*; by this we mean that the forwarding behavior does not change until the control plane explicitly alters the routing state (which happens on relatively slow time scales). Thus, one can verify the invariants before each control-plane-initiated change and know that the network will always enforce the operator-specified invariants.

While the notion of an immutable datapath supported by an assemblage of routers makes verification tractable, it does not reflect reality. Modern enterprise networks are comprised of roughly 2/3 routers[3] and 1/3 *middleboxes* [23]. Middleboxes – such as firewalls, WAN optimizers, transcoders, proxies, load balancers, intrusion detection systems (IDS) and the like – are the most common way to insert new functionality in the network datapath, and

[2] Previous work including FANG [13], SPAN [6], Margrave [19], etc. has looked at verifying firewall policies for enterprise networks, but made no attempt to verify the actual forwarding behavior.

[3] In this paper we do not distinguish between routers and switches, since they obey similar forwarding models.

are commonly used to improve network performance and security.[4]

Just as the configuration of a router is the state it uses to make forwarding decisions, the configuration of a middlebox is its set of policies (*e.g.,* drop all Skype packets). Different middleboxes are capable of implementing widely different policies (*e.g.,* an application firewall can drop Skype packets, a cache on the other hand cannot) and hence middlebox configuration cannot easily be fit into a flow-table like abstraction shared by all middleboxes. The configuration of a network is the configurations of all of its routers, all of its middleboxes, and the topology of the physical network connecting these elements. The goal of verification is to ensure that a given network configuration supports a given invariant.

While useful, middleboxes are a common source of errors in the network [21], with middleboxes being responsible for over 40% of all major incidents in networks. Thus, one cannot ignore middleboxes when verifying network configurations.

However, middleboxes do not adhere to the simple forwarding model in routers. Many middleboxes have a *mutable* datapath, in which the handling of a packet depends not just on immutable forwarding state, but also on the sequence of previously encountered packets (*e.g.,* a firewall allows packets from a flow into a network only if it has previously seen an outgoing packet from that flow). This dependence on previously seen packets renders the datapath quite mutable (with state changing at packet timescales). This prevents the use of current verification techniques, because the control over packet behaviors is no longer centralized in the control plane, but can depend on the history of packets seen by each middlebox.

Thus, we must find a way to verify network behavior in the presence of middleboxes. The more complex forwarding model supported by middleboxes renders the network Turing-complete. Furthermore, since the forwarding behavior can depend arbitrarily on packet history, the verification technique must also cope with reasoning about a potentially unbounded state space. Thus, verification techniques that cope with middleboxes look much more like general program verification than the current generation of specialized network verification techniques. The next generation of network verification tools must address two main technical challenges:

- How do we model these mutable datapaths so that verification is tractable?
- How can we feasibly analyze a network made up of these mutable datapaths?

We address these two challenges in the following sections, and then discuss how to formalize this approach and end by describing a set of open questions.

2 How to Model Middleboxes?

The natural approach for verifying mutable datapaths would be to apply standard program verification techniques to the code in each middlebox (and then extend this to the network as a whole, which is the problem we address in the next section). The practical problem with this approach is that middlebox code is typically proprietary, and any approach that relies on middlebox vendors releasing their code is doomed to fail. Moreover, there is a deeper conceptual problem with this approach. The invariants specified by network operators often use abstractions, such as user identity, host identity, application-type (of the traffic), and whether or not the traffic is "suspicious" (*e.g.,* after deep packet inspection). In fact, recent efforts to build policy languages are built around a similar set of abstractions [20].

[4] We should note that the network function virtualization (NFV) movement is moving middleboxes out of separate physical machines and into VMs that can be hosted on a cluster of servers; however, nothing in the move from physical to virtual middleboxes changes our story.

The correctness of these abstractions often *cannot* be fundamentally verified (*e.g.,* a middlebox in the middle of the network cannot always know for sure which host the packet came from given the various forms of spoofing or relaying available) or even precisely defined (*e.g.,* what is "suspicious" traffic?). Yet these abstractions are quite useful (and already widely used) in practice, and operators are willing to live with their approximations (*e.g.,* various techniques can be used to limit spoofing so that in some contexts host identification can rely on IP or MAC addresses without great risk).

To allow reasoning in terms of high-level abstractions without worrying about the various approximations that go into their definition, we model middleboxes in two parts: a reasonably simple abstract model that captures the action of a middlebox in terms of high-level primitives and an *oracle* that is described by the set of abstractions it supports. The oracle maps packets to one or more abstract classes (*e.g.,* this packet is from a Skype flow from host A and user X to host B and user Y), while the abstract model describes how the middlebox forwards packets belonging to different abstract classes (*e.g.,* a middlebox might be configured to drop all suspicious packets, or only allow packets from host A to reach host B but no other hosts). For instance, for an IDS that identifies suspicious packets and forwards them to a scrubbing box, the oracle part of the model determines which packets are suspicious and the abstract model is what dictates that such packets are forwarded to a scrubbing box.

The oracles in different middleboxes may use very different techniques to implement these abstractions. While operators care about the quality of this mapping, the goal of our network verification approach is to check that a network configuration correctly enforces invariants *assuming that the oracles are correct.* Also, different oracles may support different sets of abstractions (*e.g.,* some firewalls may be able to identify Skype traffic, and others not), and this would be described as part of the middlebox model.

In contrast, the abstract models are fairly generic in the sense that the abstract model of a firewall applies to most firewalls. The degree of detail in these abstract models depends on the invariants one wants to check. The basic network invariants of reachability and isolation only require that the abstract model describe the forwarding behavior (*e.g.,* if and where each packet is forwarded). Our initial target is verifying these basic invariants, as these properties are by far the most important safety property provided by networks. If one wants to support performance-oriented invariants, then the abstract model must include timing information (*e.g.,* what packet delays might occur), and other extensions are needed to consider invariants that address simultaneity (two properties always hold at the same time). For simplicity, we do not consider such extension here.

Separating middlebox models into an abstract model and an oracle has several advantages.

- It captures the fact that there are a limited number of middlebox "types", with many implementations of each. The abstract model applies to all of these implementations (and is fairly simple in nature), and implementations mainly differ in the abstractions and features offered by the oracle. Thus, our verification approach – which asks whether invariants are enforced assuming the oracles are right – can be applied independent of the implementations.
- It differentiates between improvements in the oracle (*e.g.,* adding new abstractions to recognize application types), which is what consumes the bulk of the development effort, and verifying correctness of the network configuration.
- It could change the vendor ecosystem by allowing (or requiring) vendors to provide the abstract model (and a description of which abstractions their oracle supports) along with their middlebox. Network operators could then perform verification, while vendors could keep their implementations private.

- While it would be useful for vendors to *verify* that their code obeys the abstract model (using standard code verification methods), what makes this approach particularly appealing is that vendors and operators alike can *enforce* that the middleboxes obey the abstract model. The abstract models (and we have built several of them) are so simple that they execute much faster than the actual middlebox implementation, so one can run these abstract models in parallel and ensure that the middleboxes take no action that does not obey the abstract model.

3 Network-Wide Verification

Modeling individual middleboxes is only the first step; our ultimate goal is to verify network-wide invariants in a network containing middleboxes and routers. There are numerous technical challenges (such as how to deal with loops), but in this short position paper we focus on the most challenging one: how can we feasibly perform verification in very large networks (containing on the order of thousands of middleboxes and routers)? To see why this is hard, consider the case of an isolation invariant (packets from host A cannot reach host B) in a network with many middleboxes. Since middlebox behavior depends not just on their configuration but on the packet history they've seen (since their datapath is mutable), verifying that this invariant holds, even if we ignore the possibility of packet loops, involves checking that there is no sequence of packets – involving packets sent from anywhere in the network at any time – that includes a packet from host A reaching host B.

Without additional assumptions, this is not feasible as the forwarding of packets from host A to host B can arbitrarily depend on other hosts (*e.g.,* on whether some other host C previously sent packets to another host D). However, the most common classes of middleboxes have an important property: the handling of a packet from host A to host B depends only on the sequence of packets (seen by the middlebox) between hosts A and B. That is, many mutable datapaths exhibit a useful kind of locality. For instance, in IP firewalls, the middlebox tracks established connections, and allows packets to pass if they are either explicitly permitted by policy or belong to an established connection. A connection between host A and B can only be established by host A and B. We therefore do not need to consider the actions of any other hosts in the network. We call middleboxes whose behavior for a pair of hosts depends only on the traffic sent between these hosts "RONO (Rest-Of-Network Oblivious) middleboxes". See Section 5 for a formal definition of RONO. RONO is not a rare property; in fact, most middleboxes (including firewalls, WAN optimizers, load balancers, and others) are RONO. Moreover, one can verify whether a middlebox is RONO by statically analyzing its abstract model.

Note that in many practical cases, the composition of RONO middleboxes is also RONO. As a result, in a network containing only RONO middleboxes we can verify reachability properties on a small subset of the network (the path between two hosts) and these properties would equivalently hold in the context of the wider network. We have leveraged this fact to verify correctness of a network containing 30,000 middleboxes in under two minutes.

4 Common Myths about Networks and SDN Verification

We next highlight some common myths about SDN networks and contrast them with the reality of today's networks.

Myth #1: *SDN networks only have controllers and OpenFlow routers, with all complicated (particularly mutable) packet processing done at the controller.* The early SDN literature [4, 16] showed examples where anything expressible using OpenFlow rules was pushed down to routers, while anything more complex was implemented in the controller. Complicated functionality included both complex processing that could not be performed at routers and simple tasks that required mutability (*e.g.,* learning switches and stateful firewalls). However, doing this processing at the controller does not scale and, in reality, middleboxes are used to provide most of the processing functions not implementable on routers, and most routers provide some mutable behavior (e.g., learning switch).

Myth #2: *Centralization, as provided by SDN, is what makes current network verification efforts possible.* Centralization is neither necessary nor sufficient for network verification. Not necessary: Verification in a network with immutable datapaths only requires being able to access router forwarding state, and current commercial network verification efforts can do this in legacy networks by using commonly available commands to read this forwarding state. Not sufficient: Regardless of SDN, current network verification efforts cannot verify networks that have middleboxes with mutable datapaths (which describes almost all real networks).

Myth #3: *Middleboxes are an aberration that will be eliminated by the rise of SDN.* Quite the opposite is true. Not only are middleboxes here to stay, but SDN itself has been evolving to incorporate middleboxes [22]. Furthermore, recently there has been an effort to move middleboxes from dedicated hardware (which is time-consuming to deploy) into virtual machines that can be deployed on quicker timescales, on existing hardware, and at lower cost. This effort is generally described as NFV (network function virtualization), and has gained significant traction commercially (comparable to or exceeding that of SDN), and recent efforts at defining a common configuration language, *e.g.,* Congress‘[20], treat middleboxes (virtualized or not) as first-class network citizens.

Myth #4: *We should write all network code and configuration in declarative languages, because their use makes verification easy.* In general, reasoning about declarative languages is undecidable [7]. It is true that verification is easy for declarative programs that do not use recursive rules (e.g., Congress [20] or NLOG programs), even in the presence of mutable states. But then, verification is equally easy for imperative programs (e.g., Python, Java, or C programs) that honor certain restrictions, e.g., do not use loops. So, in the end, it is unclear that declarative languages can make a practical difference in verification. Some argue that declarative programs are easier to read and debug, once a programmer gets used to them. On the other hand, their readability becomes questionable in the presence of side-effects.

Once one discards these myths, it becomes clear that network-verification efforts must directly confront the presence of mutable datapaths. While the approach described here may not be optimal, it is currently the only one that confronts the reality of today's and tomorrow's networks. It is time to take the next step in network verification.

5 Formalizing the Mutable Data Plane

The previous sections argued why a new approach to network verification is needed and briefly outlined what it might look like. In this section, we sketch a concrete way to formalize and prove interesting properties of networks of middleboxes.

Listing 1 Model for an IDS

```
1 oracle suspicious? (packet: Packet) : Boolean;
2
3 model ids (p: Packet) = {
4   when suspicious?(p) =>
5       forward {}
6   default =>
7       forward {p}
8 }
```

DeMillo et al. [3] previously argued that specifying the desired behavior of a program (or network) is hard. Indeed, the lack of a precise specification is a major problem for program and network verification.

The primary function of networks is to allow hosts to communicate with each other. Reachabality, the property that a certain class of packets sent from host A can reach host B, and its converse, isolation, are fundamental to networks: all useful networks must satisfy some set of reachability properties and their verification is thus universally important. In the rest of this section we limit our discussion to Reachability invariants.

We formally state these invariants using temporal logic where we assume fairness, *i.e.,* we assume any continuously enabled transition will eventually occur. First, we define two relations: $\mathit{Send}(n, p)$ indicating some network entity (node) n sent a packet p (at some time), and $\mathit{Recv}(n, p)$ indicating node n received packet p. Given these relations any reachability property can be expressed in LTL (Linear Temporal Logic) as

$$\forall p \in \text{Packet} : \Box(\mathit{Send}(\mathit{src}, p) \wedge \mathit{Predicate}(p) \implies \Diamond(\mathit{Recv}(\mathit{dest}, p)))$$

This temporal logic statement says that a packet p sent by src which satisfies $\mathit{Predicate}$ is eventually received by dest. Similarly, isolation can be formally expressed by requiring that a packet sent by src, satisfying $\mathit{Predicate}$ is never received by dest:

$$\forall p \in \text{Packet} : \Box(\mathit{Send}(\mathit{src}, p) \wedge \mathit{Predicate}(p) \implies \Box(\neg \mathit{Recv}(\mathit{dest}, p)))$$

Predicate in the definitions above is specified using the same abstractions used to specify network policies, *i.e.,* either in terms of packet header fields (source, destination, etc.) or in terms of the abstraction provided by a middlebox oracle (§2). For example, a property saying no SSH traffic can reach a server d can be expressed as

$$\forall s \in \text{Node}, p \in \text{Packet} : \Box(\mathit{Send}(s, p) \wedge \mathit{ssh}(p) \implies \Box(\neg \mathit{Recv}(d, p))) \tag{1}$$

where $\mathit{ssh}(p)$ returns true if an Oracle classifies the packet as belonging to an SSH connection. We reason about reachability and isolation properties assuming that the Oracles are correct. Verifying the isolation property in Equation (1) therefore requires answering the question: "assuming SSH traffic is correctly identified, can a packet belonging to an SSH connection reach d?"

As stated earlier, we model a middlebox as an oracle and a simple abstract model. The Oracle provides abstractions that are used to specify the properties being checked. We expect models for middleboxes (which include both an oracle and a generic model) to be specified using a constrained programming language. Listing 1 shows an example of such a specification for an intrusion detection system (IDS). The IDS oracle provides one abstraction, `suspicious?`, defined in the first line. The abstract model is defined in lines 4–9 and uses this abstraction. First we check to see if the packet is suspicious (the Oracle's decision here

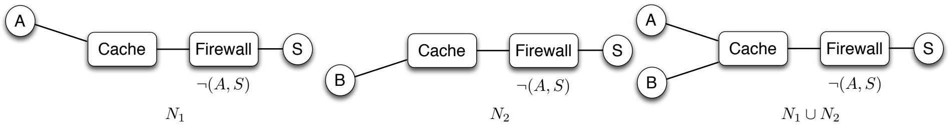

Figure 1 Example where networks are not composable with respect to reachability properties.

might be based on what packets it has seen previously), and drop the packet (line 6) or forward the packet (line 9) depending on the value returned by the Oracle.

Next, we develop abstract semantics for middleboxes. We use these to reason about general properties (such as composition) that apply to all middleboxes. Our semantic model is defined over a potentially infinite set of packets, P. We augment packets to include information about their location (*i.e.,* the middlebox or switch port). We also define the operator $\doteq$, such that $p_1 \doteq p_2$ implies that packets p_1 and p_2 are identical except for their location. Finally, we use P^* to represent the set of all (potentially unbounded) sequence of packets. The abstract model middlebox m is a function $m\colon P \times P^* \to 2^P$ which takes a packet ($p \in P$) and a history ($h \in P^*$) of all the packets that have previously been processed by m and produces a (possibly empty) set of packets $m(p, h)$. Given this model a switch is a simple function for which $m(p, h) \doteq \{p\}$. Similarly a simple firewall, f (whose decision process is represented by *allowed*) can be expressed as

$$f(p, h) = \begin{cases} \{p'\} \;\; p' \doteq p & \text{if } allowed(p, h) \\ \{\} & otherwise\,. \end{cases}$$

Ideally, we would like to be able to reason about the network *compositionally*, *i.e.,* the correctness of the network should follow from the correctness of smaller, simpler components. Compositional reasoning can reduce the cost of verification and enable incremental verification of changes in the network. Compositionality also allows us to potentially verify invariants in much larger networks, both by allowing us to parallelize verification and by reducing the size of the problem that needs to be provided to the SMT solver. Compositionality has been important for making verification tractable in other domains, for instance the use of rely-guarantees [15, 8], was important for enabling verification of concurrent programs.

We start by defining what it means to be able to compositionally verify a network. Let us define the union of two networks N_1 and N_2 in the natural way, *i.e.,* $N_1 \cup N_2$ contains the union of all nodes and links in each of N_1 and N_2. Consider two networks: N_1, where property P_1 holds (represented as $N_1 \models P_1$); and N_2, where property P_2 holds. We can compose the proofs for properties P_1 and P_2 if and only if both P_1 and P_2 hold for $N_1 \cup N_2$. For example, consider verifying the invariant "A and B cannot receive data from S" in network N in Figure 1. If compositional verification is possible for N, then the property holds in N if and only if A cannot receive data from S in N_1, and B cannot receive data from S in N_2. More formally, we can verify properties P_1 and P_2 compositionally for network N if for any $N_1 \subset N$ and $N_2 \subset N$

$$\frac{N_1 \models P_1, N_2 \models P_2}{(N_1 \cup N_2) \models P_1 \wedge P_2}\,.$$

Generally, one cannot perform compositional verification of reachability properties. For instance, consider the example in Figure 1. The cache in this example records all requests to and the corresponding responses from S. On receiving a new request, the cache checks to see if it has previously recorded a response for this request, in which case it returns the saved

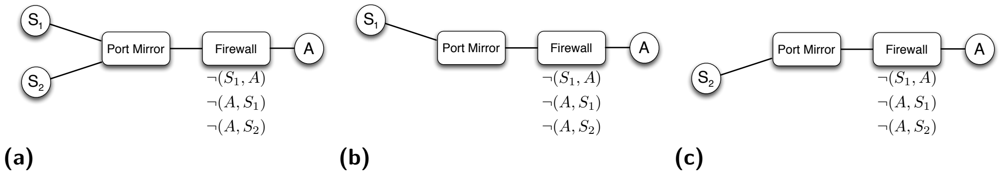

Figure 2 Example where the composition of two RONO middleboxes is not RONO.

response; otherwise the cached forwards the request, unmodified, to the firewall. The firewall drops all requests sent from A to S, but otherwise forwards all other requests and responses unmodified. In network N_1, A can never receive a response from S (thus is isolated). However in the composed network $N_1 \cup N_2$, if B sends a request r and receives response r' from S, then A can also request r and receive r'.

One key insight is that despite being impossible in general, there exists an important subset of networks where compositional reasoning can be used to verify reachability properties. We have found that networks which contain only a special class of middleboxes, Rest-of-Network Oblivious (RONO) middleboxes (§3) are often amenable to compositional verification. A RONO middlebox is one whose forwarding behavior (which is all we care about for reachability and isolation) for a pair of hosts depends only on the traffic sent between these hosts. More formally, define the restriction $h|_{(A,B)}$ of a packet history $h \in P^*$ to be the largest subsequence of h containing only those packets that were sent between host A and B; we then define a middlebox m to be RONO if and only if

$$\begin{aligned} &\forall p : p.src = A \land p.dest = B \quad f(p,h) = f(p, h|_{(A,B)}) \quad \text{and} \\ &\forall p : p.src = B \land p.dest = A \quad f(p,h) = f(p, h|_{(A,B)})\,. \end{aligned} \tag{2}$$

Similarly, we define an entire network as RONO, if its semantics can be described using a function that meets the condition stated in Equation 2.

Surprisingly, we find that not all networks that contain only RONO middleboxes are RONO, *i.e.,* RONO is not closed under composition. For example, consider the network in Figure 2. The firewall in this example is stateful and is configured to allow *no communication* between A and S_1. Furthermore, the firewall configuration allow A and S_2 to communicate, provided A establishes the connection, *i.e.,* sends the first packet. The port mirror is configured to duplicate and send all packets received from the firewall to both S_1 and S_2. Beyond these policies, both the firewall and port mirror forward traffic as expected (*i.e.,* they forward packets towards their intended destination). In this example, the port mirror is stateless, and hence trivially RONO according to Equation 2. The behavior of the stateful firewall is also RONO. However, the composition of these two middleboxes is not RONO. When we consider just S_1 and A in isolation (Figure 2b), *i.e.,* a case where S_2 neither sends nor receives any packets, A and S_1 cannot communicate, since all packets between them are dropped at the firewall. However, when we remove this restriction, *i.e.,* allow S_2 to send or receive packets, we find that A can in fact communicate with S_1: S_2 needs to first send a packet establishing a connection with A, any subsequent responses sent by A are allowed through the firewall, and duplicated so they are received by both S_1 and S_2. Therefore, it is simple to see that the abstract semantics of the network are different depending on whether we consider the entire packet history or a restriction, and the network is therefore not RONO, despite containing only RONO middleboxes. This shows that RONO is not closed under composition, *i.e.,* the composition of two RONO middleboxes might not be RONO.

However, despite RONO not being closed under composition in general, we have found that many existing networks are in fact RONO, and are hence amenable to compositional verification. Further, given that operators frequently add new middleboxes to their network, and RONO networks are precisely those where such addition cannot disrupt unaffected parts of the network, we think that many existing network might, in fact, be RONO by design.

6 Some Open Problems in Network Verification

Finally, we present some open problems that we have encountered while looking at how to verify mutable dataplanes. This list is not exhaustive, but is rather an attempt to list the first set of hurdles that need to be crossed given this new network verification agenda.

Decidability of Verification. When processing a packet, a middlebox might access potentially unbounded state. This prevents the use of finite-state model checking, and other verification techniques are undecidable for general programs in this class. We are currently working on a limited programming language that is rich enough to specify many existing middleboxes and to enable verification of some interesting network properties, including reachability properties. What other network properties can be verified in a decidable manner remains an important open problem.

Specification. While we have provided some tools that allow us to specify and check reachability properties; extending this to other invariants, for example performance-based invariants is challenging. How middleboxes and properties are specified also has a huge impact on verification time and decidability. Therefore, it is crucial to pick specifications that are rich enough to permit operators to express interesting and useful properties, yet narrow enough to permit automated reasoning.

Conditions for Compositional Verification. We have found a set of sufficient conditions that allow compositional verification of networks. However, finding a set of necessary conditions remains an open problem. Necessary conditions allowing compositional verification are useful not just for the formal verification community, but might also provide important insights about how networks should be designed and configured.

Correctness-Preserving Transformations. It might be possible to extend some of our results on compositional reasoning to show that the addition of certain types of middleboxes can never affect some class of invariants. We know this is true for some middleboxes in reality, e.g., the addition of a stateless firewall can never affect an isolation invariant (though it might invalidate some reachability invariants). Developing a theory for when this holds might be useful in developing techniques to help simplify network changes.

Verifying Parametric Topologies. Some network topologies are parametric. For example, one can generate a fat-tree topology [1] for a given datacenter size. It is possible that we can leverage compositional verification techniques to verify properties independent of the parameter. This would both speed-up verification and perhaps provide insights into the kinds of networks that are easily evolvable.

Acknowledgments. We thank Ori Lahav for help with the RONO proof. We also thank Cole Schlesinger, Aditya Akella, Shriram Krishnamurthi and Nate Foster for help debugging parts of the RONO proof. We are grateful to Justine Sherry, Colin Scott and the anonymous reviewers for their comments and suggestion on this paper. This research is supported in part by NSF grants 1420064, 1343947, 1040838; ISF grant 420/12; Israel Ministry of Science Grant 3-9772; a Marie Curie Career Integration Grant; the Israeli Center for Research Excellence in Algorithms; and by funding from Intel and AT&T.

References

1 Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

2 Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *POPL*, 2014.

3 Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. In *POPL*, 1977.

4 Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM CCR*, 38(3):105–110, 2008.

5 Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *PLDI*, 2013.

6 Swati Gupta, Kristen LeFevre, and Atul Prakash. Span: a unified framework and toolkit for querying heterogeneous access policies. In *HotSec*, 2009.

7 Alon Y. Halevy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Static analysis in datalog extensions. *J. ACM*, 48(5):971–1012, September 2001.

8 Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.

9 Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

10 Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

11 Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

12 Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *SIGCOMM*, 2011.

13 Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A firewall analysis engine. In *Security and Privacy*, 2000.

14 Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. In *SIGCOMM CCR*, 2008.

15 Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

16 Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks. In *NSDI*, 2013.

17 Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2014.

18 Tim Nelson, Arjun Guha, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A balance of power: Expressive, analyzable controller programming. In *HotSDN*, 2013.

19 Timothy Nelson, Christopher Barratt, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

20 OpenStack. Congress. `https://wiki.openstack.org/wiki/Congress`; retrieved 01/08/2015.

21 Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In *IMC*, 2013.

22 Scott Shenker. SDN: Looking Back, Moving Forward. Talk at Interent2 Technology Exchange 2014.

23 Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *SIGCOMM*, 2012.