Verifying Isolation Properties in the Presence of Middleboxes

Aurojit Panda‡, Ori Lahav‡, Katerina Argyraki‡, Mooly Sagiv○, Scott Shenker‡♠
‡ UC Berkeley, ♦MPI SWS, † EPFL, ○ TAU, ♠ICSI

Abstract
Great progress has been made recently in verifying the correctness of router forwarding tables [17][19][20][26]. However, these approaches do not work for networks containing middleboxes such as caches and firewalls whose forwarding behavior depends on previously observed traffic. We explore how to verify isolation properties in networks that include such “dynamic datapath” elements using model checking. Our work leverages recent advances in SMT solvers, and the main challenge lies in scaling the approach to handle large and complicated networks. While the straightforward application of model checking to this problem can only handle very small networks (if at all), our approach can verify simple realistic invariants on networks containing 30,000 middleboxes in a few minutes.

1 Introduction
Perhaps lulled into a sense of complacency because of the Internet’s best-effort delivery model, which makes no explicit promises about network behavior, networking has long relied on ad hoc configuration and a “we’ll fix it when it breaks” operational attitude. However, as networking matures as a field, and institutions increasingly rely on networks to provide isolation and other behavioral guarantees, there is growing interest in developing rigorous verification tools that can ensure the correctness of network configurations. The first works along this line – Anteater [26], Veriflow [20], and HSA [17][19] – provide highly efficient (in fact, near real-time) checking of connectivity (and, conversely isolation) properties and detect anomalies such as loops and black holes. This represents a massive and invaluable step forward for networking.

These verification tools assume that the forwarding behavior is set by the control plane, and not altered by the traffic, so verification needs to be invoked only when the control plane alters routing entries. This approach is entirely sufficient for networks of routers, which is obviously an important use case. However, modern networks contain more than routers.

Most networks contain switches whose learning behavior renders their forwarding behavior dependent on the traffic they have seen. More generally, most networks also contain middleboxes, and middleboxes often have forwarding behavior that depends on the observed traffic. For instance, content caches forward differently based on whether the desired content is found locally, and firewalls often rely on outbound “hole-punching” to allow flows to enter an enterprise network. We will refer to network elements whose forwarding behavior can be affected by datapath activity as having a “dynamic datapath”, and additional examples of such elements include WAN optimizers, deep-packet-inspection boxes, and load balancers.

While classical networking often treats middleboxes as an unfortunate and rare occurrence in networks, the reality is that middleboxes have become the most viable way of incrementally deploying new network functionality. Operators have turned to middleboxes to such a great extent that a recent study [33] of over one hundred enterprise networks revealed that these networks are roughly equally divided between routers, switches and middleboxes. Thus, roughly two-thirds of the forwarding boxes in enterprise networks have dynamic datapaths, and do not abide by the models used in the recently developed network verification tools. Moreover, the rise of Network Function Virtualization (NFV) [9], in which physical middleboxes are replaced by their virtual counterparts, makes it easier to deploy additional middleboxes without changes in the physical infrastructure. Thus, we must reconcile ourselves to the fact that many networks will have substantial numbers of elements with dynamic datapaths.

Not only are middleboxes prevalent, but they are often responsible for network problems. In December 2012 a misconfiguration in Google’s load balancers resulted in a several minute outage for GMail and other Google Services [3]. A recent two year study [29] of a provider found that middleboxes played a role in 43% of their failure incidents, and between 4 and 15% of these failures were the result of middlebox misconfiguration. Thus, middleboxes are a significant cause of network problems, and we have no verification tools that can help.

The goal of this paper is to extend the notion of verification to networks containing dynamic datapaths, so that we can check if invariants such as connectivity or isolation hold. Our basic approach is simple: we treat each dynamic datapath element as a “subroutine” and the network as a whole as a program. The routers and switches provide the glue that connects these procedures. The specified invariants imply constraints on dataflow within this program. We use symbolic model checking to determine if the specified invariants hold. As described so far, this is a straightforward application of standard programming lan-
guage techniques to networks. However, naïvely applied, this approach would fail to scale: middlebox code is extremely complicated, and checking even simple invariants in modest-sized networks would be intractable. Thus, the bulk of this paper, and the focus of our contribution, is about how to scale this approach to large networks.

Our efforts to scale to large networks involves four different aspects:

1. **Limited invariants:** Rather than deal with an arbitrary set of invariants, we focus on two specific categories. The first category of invariants involve the packet processing requirements (as defined by the operator) for various classes of packets; these requirements are specified as a set of middleboxes (more generally as a DAG of middleboxes) packets should flow through; we call these pipeline invariants. The second category of invariants concern the overall behavior of the network, and here we only consider invariants that address reachability and isolation between hosts (at the packet and content level). These pipeline and isolation invariants play distinct roles in the network, and their verification is done quite differently.

2. **Simple high-level middlebox models:** A standard approach to model checking middleboxes would use their full implementation. This is infeasible for two reasons: (i) we do not have access to middlebox code, and (ii) model checking even one such box for even the simplest invariants would be difficult. Instead, we consider simplified models (as we discuss later, these reduced models capture only the dependence on the packet header). These models can typically be derived from a general description of the middlebox’s behavior and can be represented as a state machine that can be easily analyzed.

3. **Modularized network models:** Networks contain elements with static datapaths and dynamic datapaths. Rather than consider them all within the model checking framework, which would overburden a system already having trouble scaling, we treat the two separately: it is the job of the static datapath elements to satisfy the pipeline invariants (that is, to carry packets through the appropriate set of middleboxes), which we can analyze using existing verification tools; and it is the job of the processing pipeline to enforce isolation invariants, and that is where we focus our attention. Thus, our resulting system is a hybrid of current static-datapath verification tools and our newly-proposed tool for dynamic datapaths.

4. **Special class of network enforcement:** Certain network designs allow invariants to be verified by checking only a portion of the network. We show that these designs carry no additional overhead but allow operators to quickly verify their policies; the use of such designs is key to scaling out verification to large networks.

These four steps lead us to a system that can verify realistic invariants on very large networks; as an example, we can verify a set of isolation invariants on a network containing 30,000 middleboxes in 2 to 5 minutes.

In the next section, we discuss all four of these steps more formally, and then in §5 we provide an overview of the system we built that incorporates these ideas. We provide a theoretical analysis of the tractability of our approach in §2 and provide performance numbers from our operational system in §5. We conclude in §6 with a discussion of related work and a brief summary.

## 2 Background

We begin by defining the verification problem addressed by this paper and describing the simplification steps that allow our system (described in §5) to scale. First we present the specific invariants we analyze (§2.1). Next we show that by focusing on these specific invariants and using some natural restriction on middlebox behaviors (§2.2) we can greatly simplify automated reasoning and verification for these middleboxes. Next in §2.3 we show how multiple middlebox models can be combined so we can reason about a network and finally (§2.4) we find some additional conditions that allow us to verify network wide properties by operating on individual pipelines instead of the entire network.

Note we do not attempt to verify that middlebox implementations are correct (i.e., obey the given model). However, we do discuss how one can enforce that middleboxes obey the abstract model by simulating the state-machine that models its intended behavior. But this enforcement is merely a small aspect of our work: our main focus in this paper is on verifying, using an SMT solver [7], that the combination of several middleboxes enforces (i.e., implements) a given invariant.

### 2.1 Desired Network Properties

We focus on three classes of invariants that address some of the core correctness issues plaguing networks:

**Packet-level reachability and isolation between end-hosts.** This is the most straightforward network invariant: can two hosts exchange packets? In most cases we want to ensure that two hosts can exchange packets, but there are scenarios where isolation is crucial and here we want to ensure that the hosts cannot exchange packets.

**Packet-level reachability and isolation between end-hosts, with learning.** This is a variation of the above invariant, where there is an asymmetry in that, for example, we might want to allow host $a$ to initiate contact with host $b$, but not allow host $b$ to initiate contact with host $a$. But once contact is properly initiated, we want two-way reachability.

**Content-level reachability and isolation between end-hosts.** One of the most interesting consequences of middleboxes is that the content of a host can be
leaked to another host (as through a cache) even when these hosts cannot exchange packets. Therefore we also consider prevention of content exchange between two hosts (and this condition need not be symmetric; content from host $a$ might be allowed to reach host $b$, but not vice versa).

This is a very restricted class of invariants, but they can be used to address slightly more general questions, such as Traversal (do packets going between source $A$ and destination $B$ always go through a particular element or link?), and Preconditions (are packet bodies modified before being processed by a particular middlebox?). However, our current approach is not able to address invariants that address issues such as quantity (how many packets can be sent between hosts?), performance (do packets travel over uncongested links?), or content (are packets containing a certain string delivered?), since these would require detailed consideration of each packet in the network, not just understanding broad classes of network behavior. Further, our choice of invariants imply conditions on particular source-destination pairs (requiring for instance that source $a$ not communicate with source $b$) rather than applying to more general network-wide properties (e.g., all source-destination pairs in a network use disjoint paths).

### 2.2 Middlebox Behavior

Because we are concerned with a limited class of invariants, we need not consider fully detailed models of middleboxes. In fact, our invariants can be checked using relatively simple models that summarize what possible set of behaviors the middlebox might take for packets with a given header. We make this more precise below.

We start with a few basic definitions: $P$ is the space of packets, $P^*$ is the space of all packet sequences, $H$ is the space of packet headers, and $MB$ is the set of middleboxes (including learning switches). In this paper we assume middleboxes have a single output port. Further middleboxes can depend on the entire packet (including the payload) and on the history of packet arrivals. A middlebox $m$ can be more formally represented as:

$$m : P \times P^* \rightarrow A \subseteq P$$

where $A$ represents the middlebox’s action on the packet: given a packet and a packet history, a middlebox can produce zero or more packets (which is why the range of the $m$ is not a single packet but a set of packets).

However, given our limited set of invariants, as we show later (through the success of our approach), we can make do with a reduced model that does not require detailed knowledge of the middleboxes decision process. This reduced model considers only how the behavior depends on the headers using a function $rm$:

$$rm : H \times H^* \rightarrow A \subseteq H$$

The reduced model does not prevent us from considering middleboxes whose behavior depends on the packet body, it merely takes the union of all such body-dependent behaviors and does not try to model which packet bodies elicit which behavior. This means that in order to model a middlebox we need not understand the details of its implementation, but only the broad outlines of the kinds of behaviors it supports.footnote

Note that these reduced models of middleboxes are often quite simple. Firewalls either forward a packet unchanged (if allowed), or block (if disallowed by an ACL), or forward conditionally if a hole has been punched by a packet in the opposite direction. Similarly, a cache either forwards a request or returns a response depending on whether it has a previously cached copy of the requested content. Thus, we assume that these reduced forms can be specified in a limited grammar (described in [3,1] and are equivalent to finite state machines.

Even with these reduced models, verification requires analyzing the entire network. However, as we argue later, this can be avoided due to the fact that many existing middleboxes (including firewalls) only depend on state pertaining to a particular flow. We would like to focus on middleboxes where $rm(h, S_{flow(h)}) = rm(h, S)$ where $rm$ is a reduced middlebox function, $h \in H$, $S \in H^*$ and $S_{flow(h)}$ is the sequence of headers which belong to the same flow as $h$ (the definition of flow can be arbitrary, as long as membership in a flow is a deterministic function of the header). This would effectively allow us to treat a given middlebox as several parallel middleboxes, one per flow. However, it turns out that this simple definition is overly restrictive, and we need a more general definition as follows. We say a middlebox is Flow-Parallel (FP) if and only if:

$$\forall h, S \exists S', S \supseteq S'_{flow(h)} \Rightarrow rm(h, S')_{flow(h)} = rm(h, S)$$

What this awkward definition means is that for every packet history, there is a possible flow history that can reproduce the same behavior. In short, the middlebox can never behave in a way that is inconsistent with a possible history of just that flow; all possible behaviors on a flow can be exhibited just by looking at the single flow. The pertinent example here is that an FP cache never returns content that was in the cache due to some other flow’s previous request if it wouldn’t have returned content if it had been requested by that flow.

---

1. A multiport middlebox can be modeled as a single-port middlebox followed by a multiport router.
2. Middlebox state is derived from this history.
2.3 Pipeline Invariants

Along with the isolation invariants described in §2.1, we also need to check pipeline invariants. A pipeline invariant takes the form: all incoming packets with headers belong to some \( I \subseteq H \) must have passed through the sequence of middleboxes \( mb_1, mb_2, mb_3, \ldots \) before being delivered by the network. Note that these invariants could refer to physical instances of middleboxes (e.g., packets must traverse this particular middlebox) or a class of middleboxes (e.g., packets must traverse a firewall). We assume that all packet headers belonging to the same flow are processed by the same pipeline (which can be enforced, as discussed below).

As we discuss in Section §3, we can check these invariants using slight extensions to current verification tools. This is possible by breaking the network into the static-datapath components and the dynamic-datapath components. A packet entering a static-datapath portion of the network (either from a middlebox, or from an ingress port) emerges at an output port in \( O \) or at a middlebox in \( MB \) with perhaps a modified header. This behavior is described by a “transfer function” \( T \) which can be efficiently computed by current verification tools, and when iterated will produce the pipeline that results from a given input packet. As we discuss later, this is sufficient to scalably verify the pipeline invariants in large networks.

2.4 Stronger Enforcement Conditions

An operator’s goal is to design their network — including the network topology, where middleboxes are placed in the network, and how they are configured — that can enforce their desired invariants. In this paper we examine how we can efficiently verify that a particular network achieves this result. It is simple to efficiently verify pipeline invariants. However, these techniques do not apply to our other invariants, whose enforcement depends in more detail on the behavior of middleboxes. For these invariants we rely on particular forms of pipelines to help scale verification.

Consider a simple pipeline \( \Pi \): a sequence of middleboxes \( \Pi = \{ m_1, m_2, m_3, \ldots, m_k \} \) (for simplicity, we ignore the possibility that intervening routers rewrite any packets and assume they merely forward them).\(^4\) This pipeline is similar to a middlebox: it maps an incoming packet and a history into one or more outgoing packets. However unlike a middlebox, the computation for a pipeline depends not just on the history of packets that traverse the pipeline, but also on packets that are received and processed by any of the constituent middleboxes. More formally:

\[
\Pi : H \times S_n \rightarrow A \subseteq H
\]

where \( S_n \) is the sequence of all packets sent in the network. Analyzing such pipelines is expensive since it requires considering the behavior and possible histories of all the constituents of a network.

Similar to flow-parallel middleboxes, we say that a pipeline is “rest-of-network-oblivious” (RONO) if and only if:

\[
\forall h, S_n \exists S' \supseteq S_n |_{flow(h)} \text{ s.t. } \Pi(h, S' |_{flow(h)}) = \Pi(h, S_n)
\]

where \( S' |_{flow(h)} \) is the history restricted to the flow defined by header \( h \) as above, and whose packet flow through the entire pipeline. As noted previously, the isolation invariants we focus on are all stated in terms of pairs of endhosts (and can thus be naturally extended to a set of flows). It is now simple to see that given an invariant \( I \) involving hosts \( a \) and \( b \), and a set of RONO pipelines \( \Pi_1 \ldots \Pi_n \) connecting \( a \) and \( b \) (each applying to a different set of headers), \( I \) holds for the entire network if and only if \( I \) holds for all pipelines \( \Pi_1 \) to \( \Pi_n \).

Thus if one can enforce the desired invariants in a network using RONO pipelines, then one need only verify invariants on the pipeline in isolation. While RONO and flow-parallelism are closely related, somewhat surprisingly not all compositions of FP middleboxes are RONO. In §4.2 we derive conditions under which compositions of FP middleboxes are guaranteed to be RONO. Isolation invariants can thus be verified quickly by checking that (a) all middleboxes in the pipelines connecting the endhosts are flow-parallel, (b) the pipelines themselves are RONO and (c) the invariant holds on each pipeline. The first two steps are relatively simple static checks that depend only on the middlebox model specification and the last verification step generally scales with the length of the pipeline and policy size (i.e., the number of invariants) rather than the size of the network. We evaluate scalability empirically in §5.

3 System Design

Based on the previous theory, we have implemented a system to verify invariants of the type described in §2.1 in a given network. Our system uses Z3 [7], a state-of-the-art SMT solver, as an oracle that can prove theorems of the appropriate form. For verification we require two inputs: (a) middlebox models written in a restricted language based on Python (§3.3); (b) network topology information including routing tables, middlebox configurations and end host metadata. The system converts these inputs to a suitable form, adds additional assertions (§3.3) describing the physical behavior of the network, as well as the invariant being checked, and produces the input supplied to Z3. Given this input, Z3 returns \texttt{unsat} (indicating that the input assertions can never be satisfied), \texttt{sat} (indicating that a satisfiable assignment was found) or undefined (in-


```python
def LearningFw(node, acl, flows):
    p = recv(node)
    if acl[p.src(), p.dest()]:
        send(node, p)
        flows.set((p.src(), p.dest(),
                   p.src_port(), p.dest_port(),
                   True))
    elif flows[(p.dest(), p.src(),
               p.dest_port(), p.src_port())]:
        send(node, p)
    acl = ConfigMap((Address, Address), Bool)
    flows = Map((Address, Address, Int, Int), Bool)
    LearningFw(f, acl, flows)
```

Listing 1: Model for a learning firewall

indicating that the check timed out). The system interprets this return value to determine if the invariant holds. We describe each of these steps below.

### 3.1 Modeling Middleboxes

SMT solvers cannot always prove (or disprove) fully general theorems and we must limit the complexity of our input to Z3. We therefore require that middlebox models used by our system are restricted so that:

1. Models are expressed so they are loop-free and all received packets are processed in a fixed number of steps.
2. Models only access local state which is naturally true for existing middleboxes, which are physically distinct and must use the network to share state.
3. Models are expressed using a limited set of actions: they can receive packets, check conditions, send packets and update state.
4. Models must be deterministic for a given packet and history. For flow-parallel (FP) middleboxes we require that the modeled behavior be identical for a given packet and all histories with the same flow-restricted history, i.e., \( m(h, S) = m(h, S') \) for all \( S, S' \in H^f \) with \( S|_{flow(h)} = S'|_{flow(h)} \) and FP middlebox \( m \). While existing implementations can be non-deterministic (e.g., NATs that assign ports in order of flow initiation), these have equivalent, semantically correct, deterministic versions (for instance a NAT that uses flow hashing to assign ports) for which our invariants hold if and only if they also hold in the non-deterministic case.

Users specify middlebox models (which are general and can be reused for different networks) using a subset of Python that allows users to:

- Read and set values from instances of `Map` objects, which behave like dictionaries or hash maps.
- Read values from `ConfigMap` objects, which hold configuration information.
- Call uninterpreted functions with finite codomains

\[\text{Listing 2: Model for a DPI middlebox}\]

(i.e., returns one of a finite set of values)\(^5\)
- Use conditionals `if, else, elif`.
- Construct a `packet` and set packet field values.
- Call the `recv` function to receive a packet.
- Call the `send` function to send a packet.

As an example, consider the model for a learning firewall shown in Listing 1. The firewall can forward received packets either because this is explicitly allowed by the firewall policy (line 3, where we check to see if the packet is allowed by the list of ACLs) or because a previously allowed packet established flow state (in line 5, we modify the `Map` flows to record what packets have been seen, which we then check in line 6). The model itself is specified by the function definition (lines 1–7). Lines 9–11 show how this model can be initialized for a node \( f \).

Listing 2 shows an example where an uninterpreted function `dpi` (defined on line 1) is used. `dpi` accepts a packet body and returns a boolean and hence has a finite codomain of size 2 (i.e., returns true or false).

The system translates these models into an equivalent set of formulas in temporal logic that we supply to Z3. Figure 1 shows the formulas for an instance \((f)\) of the learning firewall.

The translation works by performing a depth first traversal of the abstract syntax tree (AST) to find all calls to `send` or `set` (henceforth referred to as “actions”) and the path leading to these calls. The path is converted to an appropriate path constraint and we output assertions of the form `action \implies path constraint`, essentially requiring that if an action (e.g., a packet is forwarded) is executed then all conditions leading up to it must hold.

Our model description could produce several equivalent sets of formulas. Later in \[\text{4}\] we use one such equivalent formulas to prove that our formulation is decidable. We chose this particular form of formulas based on how long it took Z3 to produce a proof for these formulas.

### 3.2 Network Transfer Functions

We also place a few restrictions on the topology and forwarding state for networks we verify, in particular we require that the networks under consideration be:

1. Forwarding loop free, this is required to ensure that the formulas supplied to the SMT solver are decidable.
2. Have no black holes (i.e., routers and switch forwarding tables be setup such that packets are always forwarded to their destination).

---

\(^5\)First-order logic is undecidable in general and we must restrict ourselves to formulas in a decidable fragment. See \[\text{4}\] for details.

\(^6\)In our current implementation we do not check that uninterpreted functions have finite codomains.
We translate these pipelines into composition constraintsed onto our model to determine the possible pipelines traversed by a packet sent from one end host to another.

Finally, we add to the network model one or more variables (encoding conditions that should or should not hold)

\[
\begin{align*}
\text{send}(f, n, p, t) & \Rightarrow \exists n', t'. \text{recv}(n', f, p, t') \land t' < t \\
& \land (\text{acl}(s(p), d(p))) \\
& \lor (\text{acl}(s(p), d(p))) \\
& \lor \text{flows}(d(p), s(p), d(p), sp(p), t')) \\
\end{align*}
\]

\[\text{flows}(s, d, sp, dp, t) \Rightarrow \]

\[\exists n', p. \text{recv}(n', f, p, t') \land t' < t \\
\land \text{acl}(s(p), d(p)) \land s = s(p) \\
\land d = d(p) \land sp = sp(p) \]

Figure 1: Formulas generated from Listing 1. The left side of the first two formulas have a universal quantifier, i.e., each is preceded by \(\forall n, p, t\) etc. \(f\) is a symbolic node representing the firewall.

We leverage VeriFlow [20] to both check that the input topology and forwarding tables meet the previous requirements and to produce a forwarding graph that we can then convert to a set of “composition assertions” that we add to the theorem provided to Z3. Along with the topology and forwarding table we also accept configuration for middleboxes and endhosts. This configuration specifies the type of the node (which we use to create a new instance from the appropriate model) and any configuration that the model might depend on. For end hosts this configuration also specifies the set of addresses assigned to a specific host (we assume that hosts are honest, i.e., they do not send packet with spoofed addresses, this can be easily enforced at the first hop).

Once these models are instantiated we query the forwarding graph to determine the possible pipeline(s) traversed by a packet sent from one end host to another. We translate these pipelines into composition constraints of the form \(\text{send}(a, n, p, t) \land (d(p) = ip_a) \land (s(p) = ip_b) \implies (n = f)\). The previous composition constraint indicates that packets leaving host \(a\) with destination address \(ip_a\) and source address \(ip_b\) are sent to the firewall next. We refer to this collection of instantiated middlebox models and composition constraints as the network model.

### 3.3 Other Assertions

Next we add to the network model some basic axioms describing the universe in which the network operates. These axioms (Figure 2) state that:

1. The network has no local loops (i.e., no packets with the same source and destination address).
2. Any packet received at a node \(n'\) from node \(n\) at time \(t\) was sent by \(n\) at an earlier time \(t'\).
3. Time is represented by positive numbers.

### 3.4 Verification

Finally, we add to the network model one or more variables (representing packets) and assertions on these variables (encoding conditions that should or should not hold

\[
\begin{align*}
\text{send}(n', p, t) & \implies n \neq n' \\
\text{send}(n', p, t) & \implies s(p) \neq d(p) \\
\text{recv}(n', p, t) & \implies \exists f'. \text{send}(n', f', p, t') \land t' < t \\
\text{send}(n', p, t) & \implies t > 0 \\
\text{recv}(n', p, t) & \implies t > 0 \\
\end{align*}
\]

Figure 2: Basic network axioms.

if the invariant holds) to generate an input for Z3. Given this input Z3 either returns a valid assignment for the variables such that the supplied assertions hold under the network model (\texttt{sat}) or that no such assignment exists (\texttt{unsat}). The variables and assertions added for each invariant are:

- **Node Isolation**: To check node isolation between nodes \(a\) and \(b\), we add a variable representing a packet \(p\) and assertions requiring that the packet was sent by \(a\) and later received by \(b\). If the solver returns \texttt{unsat} no such packet can exist and the nodes are isolated. Note, that node reachability is the negation of this and is true whenever the solver returns \texttt{sat}.

- **Flow Isolation**: Flow isolation is verified by adding an additional assertion to Node Isolation: the additional assertion states that \(b\) has never before sent a packet to \(a\).

- **Data Isolation**: We rely on a pseudo-field on our packet to indicate what machine data originated on. Models for caching firewalls are expected to preserve this field. Given this pseudo-field we can check that \(a\) never accesses data from \(b\) by proving that there does not exist a packet \(p\) such that \(p\) is received at \(a\) and has origin \(b\).

- **Node Traversal**: The node traversal invariant requires that all traffic from host \(a\) to host \(b\) pass through some middlebox \(m\). This can be proved by showing that if \(m\) neither nor receives any packets then \(a\) and \(b\) are (node) isolated from each other.

We can also use Node Isolation with an additional constraint to measure the number of host pairs which can potentially use a link (path in our case), we call this link traversal. To measure link traversal we run \(2 \times \binom{n}{2}\) node isolation checks with the added constraint that the packet must have gone over the link. We then return the number of cases in which Z3 returns \texttt{sat} for this check.

### 3.5 Enforcement

Our models are not automatically derived from implementations and hence it is possible (due to bugs) that the implementation for a middlebox deviates from its model. We address this by providing a runtime mechanism for detecting instances where the implementation’s behavior deviates from what is allowed by the model, we call this process enforcement.

Since our models are specified as simple Python programs (§3.1) they can be executed as long as we generate
\[ \text{snd}(e) \land s(e) = f \implies t(cause(e)) < t(e) \]
\[ \land \text{rcv}(cause(e)) \]
\[ \land d(cause(e)) = f \]
\[ \land p(cause(e)) = p(e) \]
\[ \land \text{acl}(s(1), d(1)) \]
\[ \text{rcv}(e) \lor s(e) \neq f \implies cause(e) = e \]

Figure 3: EPR-F formulas equivalent to
\[ \text{send}(f, n, p, t) \Rightarrow (\exists t'. \text{rcv}(n', f, p, t')) \land t' < t \land \text{acl}(s(p), d(p)) \].

an implementation for uninterpreted functions. Since, uninterpreted function have finite codomains, we provide a simple implementation where we execute the model once for each value in an uninterpreted function’s codomain. Given this implementation our enforcement strategy is simple: when a packet is received at a middlebox, we send a copy of the packet to the enforcement code, which generates all possible outputs that are allowed by the model. We then compare the middlebox’s output to these possibilities and report a deviation when no match is found.

4 Theoretical Analysis and Decidability

Next we try and answer two questions about our formulation: are our formulas decidable (§3.1) i.e., can Z3 solver find a satisfiable assignment (or prove that the formulas are unsatisfiable) in a finite number of steps, and secondly conditions under which combination of middleboxes will result in RONO pipelines (§3.2).

4.1 Decidability

In general, first order logic is undecidable. However we show that when we restrict our models (§3.1) and topology (§3.2) as described previously, we get formulas that lie in a decidable fragment of first-order logic. This fragment is a simple extension of “effectively propositional logic” (EPR). EPR is one of the fundamental decidable fragments of first-order logic [28]. An EPR formula is a set of function symbol free \( \exists \forall \exists^+ \) premises (assertions) and a function symbol free \( \forall \exists^+ \) formula as a consequence (whose negation is added as an assertion before checking for satisfiability). Z3 and other SMT solvers use algorithms that are guaranteed to terminate for this fragment.

In our case, the formulas obtained from modeling middleboxes (e.g., Figure 1) as well as one of the network axioms (assertion 3 in Figure 2) are \( \forall \exists^+ \) premises and hence not in EPR. Such premises may result in the formula being undecidable (which would cause the SMT solver to timeout). Therefore our case requires a more expressive logic that we call EPR-F. EPR-F extends EPR to allow restricted unary functions. To ensure that the formulas are decidable, EPR-F requires that unary functions have certain closure properties such that some finite composition (including composition of the function with itself) must result in an idempotent function, e.g., for a EPR-F formula with a single function \( f \) there must exist \( k \) such that \( f^k(x) = f^{k-1}(x) \). Note that “non-cyclic” function symbols that go from one type to another\(^7\) can be employed freely [22]. A fragment similar to EPR-F was employed in [15], which also shows that EPR-F formulas can be reduced to pure EPR.

One can translate our models (e.g., Figure 1 to EPR-F using the following steps and then use the fact that our topology and middleboxes are loop-free to prove that all introduced functions are idempotent:

1. Reformulate our assertions with “event variables” and functions that assign properties like time, source and destination to an event. We use predicate function to mark events as either being sends or receives.

2. Replace \( \forall \exists \) formulas with equivalent formulas that contain Skolem functions instead of symbols. For example the formula \( \forall n, n', p, t. \text{rcv}(n, n', p, t) \Rightarrow \exists t'. \text{send}(n, n', p, t') \land t' < t \) is translated to the equivalent formula \( \forall e. \text{rcv}(e) \Rightarrow \exists t'. \text{send}(e, e, p, t') \land t' < t \) and we add a second assertion \( \forall e. \text{snd}(e) \Rightarrow \text{cause}(e) = e \) to ensure that \( \text{cause}(e) \) is idempotent for send events. Figure 3 shows another example of this reformulation for a simple ACL firewall (with no learning action).

Finally, we need to show that these newly introduced formulas have the desired closure properties. Note first that each cause functions is idempotent: they are the identity function for either send or receive events and when not the identity function \( \text{cause} \) links a send to a receive. Furthermore, when applied to an event \( e \) the value of this function (when not idempotent) is always another event such that \( \text{cause}(e) = s(e) \). Since we assume that we have loop free forwarding this must terminate in a finite number of steps (when we would have reached the edge of the network) and therefore the newly introduced functions meets the requirement for being in EPR-F.

4.2 RONO Pipelines

In this section we look at conditions under which network pipelines are RONO. To start with define composition of two middleboxes (or pipelines, which have the same form) \( \{m_1, m_2\} \) to be the single middlebox representing the case where all outputs from \( m_1 \) are sent to \( m_2 \). More formally we define the composition function \( C \) as:

\[ C(m_1, m_2)(p, S) = \{m_2(p', S) | p' \in m_1(p, S)\} \]

where \( S \) is the combination of both \( m_1 \) and \( m_2 \)’s histories.

\(^7\)Assuming no function go back.

| Hosts |
|-------|
| 500   |
| 1000  |
| 1500  |

| Firewall |
|---------|
| 43.87s  |
| 384.15s |
| 1736.70s|

| Content Cache |
|--------------|
| 282.94s      |
| 4264.16s     |
| 19819.72s    |

Table 1: Time to verify invariants in larger networks.
As described previously, we say that a pipeline $\Pi$ is RONO if and only if
\[
\forall S_n \exists S' \supseteq S_n|_{\text{flow}(h)} \text{ s.t. } \Pi(h, S') = \Pi(h, S_n)
\]
\[\text{i.e., a single middlebox equivalent to the composition of all the middleboxes (and routers and switches) in a pipeline is flow parallel.}\]

In this section we analyze cases when pipelines are RONO and cases where pipelines are not. Note, our analysis here is conservative, i.e., we find sufficient conditions for pipelines to be RONO and it is possible that other pipelines are also RONO.

For the statements below we focus on the composition of flow-parallel middleboxes. While one can in theory construct RONO pipelines out of middleboxes that are not flow parallel, the precise conditions for doing so depend on the behavior of the middleboxes in question and are hard to generalize.

**All pipelines containing a single flow-parallel middlebox are RONO:** this follows trivially from the definition of flow-parallel middleboxes and RONO pipelines.

Based on the previous result we state all subsequent results in terms of RONO pipelines.

**Not all compositions of RONO pipelines are RONO:** we show this by presenting a simple counter-example. Consider the middlebox $m(p, S) = \{c\}$ where $c \in P$ is a packet with source $ip_m$, destination $ip_m$, source port $s_m$ and destination port $d_m$, i.e., $m$ outputs a constant packet $c$ for any input.

Since the definition of $m$ is independent of $S$ we have $\forall S \in H^*. m(h, S)|_{\text{flow}(h)} = m(p, S)$ and $m$ is trivially flow parallel. Now consider the composition of $m$ with a RONO pipeline $\Pi$. Since $m$ outputs the same packet $c$ regardless of the received packet, for $\Pi$ flows are indistinguishable and hence it cannot partition the history of received packet. $C(m, \Pi)$ is therefore not RONO despite both $m$ and $\Pi$ being RONO.

We say a RONO pipeline $\Pi$ is flow preserving if and only if:
\[
\forall S, h, h' : \text{flow}(\Pi(h, S)) = \text{flow}(\Pi(h', S)) \iff \text{flow}(h) = \text{flow}(h')
\]
that is $\Pi$’s action maps headers for packets belonging to the same flow to packets in the same flow (i.e., a flow-preserving pipeline $\Pi$ is injective with respect to flows).

**The composition of two flow preserving pipelines is also flow preserving** – the composition of two injective functions is injective.

The composition of flow-preserving RONO pipelines is RONO: This is obvious to see: the previous problem is a result of flows being merged, this is impossible given flow-preserving middleboxes.

We thus see that any pipeline of flow-preserving, flow-parallel middleboxes is RONO.

## 5 Evaluation

We now evaluate our system’s performance and demonstrate gains when compared to an alternative that model-checks the entire network (dubbed “naïve approach”). First, we show that our system can be used to verify existing networks in reasonable time. We evaluate our system’s performance and scalability both on enterprise and departmental networks that use stateful firewalls (§5.1) and on provider networks that use content caches (§5.2). Next, we evaluate our system’s performance on more general functionality and show that it can be used to check invariants in the presence of other kinds of middleboxes (§5.3). Next, we evaluate the benefits of RONO (§5.4). We close by evaluating our system’s performance when verifying node-traversal invariants (§5.5).

### 5.1 Stateful Firewalls

**Single FP Middlebox.** We first consider the topology in Figure 4(a) where one stateful firewall protects a network that consists of three groups of hosts: Quarantined, Normal, and External-facing. This arrangement is typical of small and medium-size enterprises and several departmental networks (including the network at UC Berkeley). We use our system to verify the following node-isolation and flow-isolation invariants:

1. Quarantined hosts are node-isolated: No quarantined host can send or receive packets from the external network.
2. External-facing servers can both access and be accessed from the external network.

They may be several redundant physical firewall devices for fault tolerance, but a single firewall processes all traffic between the internal and external network.
3. Normal hosts are flow-isolated: Any normal host is allowed to establish connections and communicate with nodes in the external network, but the external network cannot establish a connection with the host.

To implement these invariants, we configure the firewall with two rules denying access (in either direction) for each quarantined host, plus one rule denying inbound connections for each normal host. For our evaluation, we consider a network with equal numbers of hosts of each group (i.e., a third of the hosts are quarantined and a third are externally accessible). We configure our firewall correctly, and our results are for the case where all invariants hold. Note, however, that misconfiguring the firewall so that an invariant does not hold for a particular host merely places that host in a different group. For instance, suppose the firewall is misconfigured causing a particular quarantined host not to be isolated; from the point of view of verification complexity, this is similar to the situation where the host is a normal or external-facing host, and the firewall is correctly configured. Thus, our evaluation provides relevant timing information for both the case where an invariant holds and the cases where it is violated.

To start with, we measure the time taken to verify that the correct invariants hold for all the hosts in the network. Learning firewalls are flow-parallel, hence each pipeline that involves a host and the firewall is RONO and can be checked in isolation. Figure 4(b) shows pipeline verification time when we check each pipeline in isolation (“Ours”) and when we check the entire network (“Naïve”): for a moderately sized network with 100 hosts, our system takes about 1 sec per pipeline; hence, in the worst-case scenario where it runs on a single core, it checks all 100 pipelines in about 100 seconds, which is two orders of magnitude faster than the naïve approach. Table 1 shows pipeline verification time for our system, when we have larger numbers of hosts: for a network of 1500 hosts, it is close to half an hour; the naïve approach does not terminate in useful time.

Even for our system, pipeline verification time increases with the number of hosts in the network, because the number of hosts affects the number of rules that are installed in the firewall, and these rules must be checked for each pipeline. Figure 4(c) breaks down pipeline verification time per invariant, and we see that verifying node-isolation takes less time than verifying flow-isolation or reachability. This is because node-isolation is expressed as unsatisfiability, and Z3 has a known pathology where, for many problems, proving unsatisfiability is faster than finding a satisfiable assignment. While this is a known problem, the exact reason is not understood. Other solvers, for instance CVCLite, do not suffer from this limitation but have other idiosyncrasies.

**Increasing Pipeline Length.** In reality, hosts in Figure 4(a) connect to the firewall through a series of switches, routers, and perhaps other middleboxes that we have elided. Next, we look at two questions: (i) what is the cost of explicitly modeling switches/routers and (ii) how does verification scale with increasing pipeline length. We analyze both of these by changing the pipeline in Figure 4(a) by introducing a series of routers (or middleboxes that forward all received packets) as shown in Figure 5(a).

Figure 6 shows pipeline verification time for pipelines of increasing length. We find that it grows rapidly with increasing pipeline length, both for our system and the naïve approach. Even so, our system is two orders of magnitude faster. Fortunately, due to the additional latency added by each middlebox, we expect the number of middleboxes in a pipeline to be relatively small. Further, given that we require the forwarding configuration of switches and routers in the network under analysis to contain no black holes, we can usually elide switches and routers while verifying our supported set of invariants.

**Scalability microbenchmark.** Our system outperforms the alternative because it models the network as a set of parallel pipelines and checks each pipeline in isolation; but even for a single pipeline, policy size grows as a function of the network size: for instance in our enterprise network example, the firewall has $O(n)$ ACL rules where $n$ is the number of hosts. Next, we consider the impact of policy size on verification time. We study this in the topology in Figure 5(b), where two hosts, each with $n$ IP addresses, are separated by a firewall with $O(n^2)$ ACLs. We consider two invariants:

1. Isolation: the two hosts can never communicate. For this, we install $n^2$ DENY exact-match rules in the firewall,
We measure how verification time increases with the number of firewall rules. Figure 7 shows the results. Note that while verification time does not grow rapidly, larger policy sizes result in larger models that need more memory to be expressed. At large enough sizes verification does not succeed because of memory pressure.

5.2 Content Caches

Next, we consider the topology in Figure 8(a) where a set of hosts access a set of content servers placed behind a content cache and a firewall. This setup is representative of content-provider networks, which employ content caches to improve the performance of user-facing services and reduce server load, but may place restrictions on what hosts can access a particular class of content.

A content cache typically implements an ACL to prevent clients from using caching to bypass policy: Consider Figure 8(a) and suppose that each of the two content servers provides content to different clients. For example, when client a requests “http://xxx.com/latest-news,” the request is served by s1; when client b requests the same name, the request is served by s2. So, a request for the same name results in different content, depending on who is asking (a typical practice by content providers). To implement this policy, the provider configures the firewall to prevent communication between a and s2. However, every time b accesses content from s2, that content gets cached in the content cache and becomes available to a—unless the cache implements an ACL specifying that a must not access content that originated in s2.

We use our system to verify two data-isolation invariants:

1. Hosts h2 . . . h_n can never access any content that originated in server s2.
2. Hosts h1 . . . h_n can always access any content that originated in server s1.

To implement these invariants, we configure the content cache with an exact-match rule denying access (to data that originated from server s2) for each of hosts h2 . . . h_n.

To make verification more challenging, we also configure the firewall with two exact-match rules denying access (to and from s2) for each of these hosts. As above, we configure the middleboxes correctly, and our results are for the case where all invariants hold, but verification complexity is similar when the invariants are violated.

We measure the time taken to verify that the correct invariants hold for all hosts in the network. The content cache is flow-parallel: any content requests made by host h_i do not affect the behavior of the content cache toward requests made by host h_j. The firewall is also flow-parallel and flow-preserving. Hence, a pipeline consisting of the content cache, the firewall and two end-hosts is RONO and can be checked in isolation. Figures 8(b) and 8(c) show pipeline verification time when we verify each pipeline in isolation (“Ours”) and when we verify the entire network (“Naive”): in the case of 100 accessing hosts, our system takes about 12 minutes per pipeline, more than an order of magnitude faster than the naive approach. Table 1 shows pipeline verification time for our system, when we have larger numbers of hosts: for a network of 1000 hosts, it is a little above an hour; the naive approach does not terminate in useful time. Consistently with the results presented in Section 5.1, verifying that certain hosts cannot access certain content (Figure 8(c)) is significantly faster than verifying that certain hosts can access certain content (Figure 8(b)).

5.3 Generic Middleboxes

So far we have considered two existing, commonly used middleboxes; now we provide evidence that our system can handle other types of stateful middleboxes.

We consider the topology in Figure 9(a) where source a is connected to destination b through a “permutation middlebox” p_0 and a firewall f that drops traffic between specific address pairs. Each of hosts a and b has multiple addresses; we denote a’s addresses by a’ and b’s addresses by b’. When a sends a packet to b, p_0 replaces the packet’s source and destination addresses with a different pair; how it chooses the new address pair depends on previously observed traffic from a to b. For example, the box’s configuration may dictate that: after observing a packet
with source address $a^1$ and destination address $b^1$, in any future packet, source address $a^1$ will be replaced with $a^2$, and destination address $b^1$ will be replaced with $b^2$. Similarly, each permutation box $p_i$ permutes the source and destination addresses of packets from host $e_i$ to host $b$ based on previously observed traffic between these hosts.

This setup is contrived, but it captures the complexity of any situation where a middlebox changes incoming packets based on traffic history. In our particular setup, the middlebox permutes the source and destination addresses of incoming packets, and its choice of new addresses determines whether the firewall will drop each resulting packet or not (potentially causing the violation of an isolation invariant). In a different setup, the middlebox might change some other part of incoming packets, and its choice would determine whether a downstream intrusion detection system would drop each resulting packet or not.

We use our system to verify the following node-isolation invariant: host $a$ can never send traffic to host $b$. To implement this, we configure the firewall $f$ to drop all packets with $alb$ address pairs. The permutation box $p_0$ is flow-parallel: its behavior with respect to $alb$ traffic depends only on previously observed $alb$ traffic, which allows us to check the $a-p_0-f-b$ pipeline in isolation. Figure 9(b) shows pipeline verification time as a function of the number of hosts in the network.

The three curves in Figure 9(b) together show the benefit of RONO: when middleboxes are flow-parallel, our system leverages RONO and completes verification in tens of milliseconds (“Ours”); with the naïve approach (“Naïve”), verification takes as long as if the middleboxes were not flow-parallel (“Non-FP”).

5.4 The Benefit of RONO

So far we have considered only flow-parallel middleboxes, which our system was designed to handle efficiently; we now describe a scenario with non-flow-parallel middleboxes, where our system is as good as the naïve approach.

We consider again the topology in Figure 9(a) and the same node-isolation invariant as above (host $a$ can never send traffic to host $b$), and we configure the firewall to drop all packets with $alb$ address pairs. However, we make the permutation middleboxes non-flow-parallel: $p_0$ determines how to permute $alb$ addresses based on previously observed traffic, not only from $a$ to $b$, but from any host $e_i$ to $b$. As a result, $p_0$’s behavior with respect to $alb$ traffic depends on traffic previously carried by other pipelines, and we cannot check the $a-p_0-f-b$ pipeline in isolation.

The “Non-FP” curve in Figure 9(b) shows pipeline verification time as a function of the number hosts in the network.

The three curves in Figure 9(b) together show the benefit of RONO: when middleboxes are flow-parallel, our system leverages RONO and completes verification in tens of milliseconds (“Ours”); with the naïve approach (“Naïve”), verification takes as long as if the middleboxes were not flow-parallel (“Non-FP”).

5.5 Node Traversal

We consider the topology in Figure 10(a) where $a$, $b$, $c$ and $d$ are middleboxes that always forward all received packets: the forwarding tables are set up such that all traffic sent by any host ($e_0$…$e_n$) is first sent to $a$, which depending on the destination forwards this traffic to either $c$ or $d$; finally, both $c$ and $d$ forward packets to $b$, which delivers them to the intended host. Note that checking node traversal requires considering the entire network (so RONO does not help us here). We use our system to verify a node-traversal invariant: that traffic from host $e_i$ to host $e_j$ always traverses middlebox $c$, for a given set of $\{i, j\}$ pairs. We configure the middleboxes such that the invariant holds for half the host pairs and not for the other half. Figure 10(b) shows the average time to check this property for each host pair (averaged across the set of all considered host pairs).
6 Related Work

The earliest use of formal verification in networking focused on proving correctness and checking security properties for protocols [6,30]. The first application of these techniques to control and data plane verification looked at verifying BGP configuration [11,12] in WANs.

Verifying Forwarding Rules Recent efforts in network verification [14,19,20,25,32,34] have focused on verifying the network dataplane by analyzing forwarding tables. Some of these tools including HSA [18], Libra [38] and VeriFlow [20] have also developed algorithms to perform near real-time verification of simple properties such as loop-freedom and the absence of blackholes. While well suited for checking networks with static data planes they are insufficient for dynamic datapaths.

Verifying Network Updates Another line of network verification research has focused on verification during configuration updates. This line of work can be used to verify the consistency of routing tables generated by SDN controllers [16,36]. Recent efforts [24] have generalized these mechanisms and can be used to determine what parts of the configuration are affected by an update, and verify invariants on this subset of the configuration. This line of work has been restricted to analyzing policy updates performed by the control plane and does not address dynamic data plane elements where state updates are more frequent and span a wider range.

Verifying Network Applications Other work has looked at verifying the correctness of control and data plane applications. NICE [4] proposed using static analysis to verify the correctness of controller programs. Later extensions including [23] have looked at improving the accuracy of NICE using concolic testing [51] at the cost of completeness. More recently, Vericon [22] has looked at sound verification of a restricted class of controllers.

Recent work [8] has also looked at using symbolic execution to prove properties for programmable datapaths (middleboxes). This work in particular looked at verifying bounded execution, crash freedom and that certain packets are filtered for stateless or simple stateful middleboxes written as pipelines and meeting certain criterion. The verification technique does not scale to middleboxes like content caches which maintain arbitrary state.

Finite State Model Checking Finite state model checking has been applied to check the correctness of many hardware and software based systems [5]. Here the behavior of a system is specified as a transition relation between finite state and a checker can verify that all reachable states from a starting configuration are safe (i.e., do not cause any invariant violation). Tools such as NICE [4], HSA [19] and others [55] rely on this technique. However these techniques scale exponentially with the number of states and for even moderately large problems one must chose between being able to verify in reasonable time and completeness. Our use of SMT solvers allows us to reason about potentially infinite state and our choice of formulas are expressible in a way that guarantees termination of the SMT solver (§4.1).

Language Abstractions Several recent works in software-defined networking [13,14,21,27,37] have proposed the use of verification friendly languages for controllers. One could similarly extend this concept to provide a verification friendly data plane language however our approach is orthogonal to such a development: we aim at proving network wide properties rather than properties for individual middleboxes.

Finally, parallel to our work Fayaz, et al. [10] use middlebox models to generate test packets to check network invariants. Similar to us, they model middleboxes as state machines, however our models are expressed differently and they aim to test rather than verify networks.

7 Conclusion

On the one hand our work can be seen as merely a technical demonstration that one can verify isolation environments in large networks. However, our ambition is larger than merely providing operators with another verification tool. We hope that armed with the ability to verify both the static-datapath and dynamic-datapath aspects of a network, operators demand abstract middlebox models from their vendors. This would (a) allow operators to enforce these aspects of the middleboxes, so that middlebox violations of prescribed behavior can be detected and (b) allow operators to verify that their overall network meets their desired invariants. This would move networking from its current ad hoc practice to a more desirable state where invariants are explicitly expressed and rigorously enforced.
References

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In POPL, 2014.

[2] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In PLDI, 2014.

[3] J. Brodkin. Why Gmail went down. ArsTechnica Dec 11, 2012 [http://goo.gl/2WNqmr](http://goo.gl/2WNqmr) (Retrieved 09/21/2014), 2012.

[4] M. Canini, D. Venzano, P. Peres, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In NSDI, 2012.

[5] E. M. Clarke, O. Grumberg, and D. Peled. Model checking. MIT Press, 2001.

[6] E. M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In PROCOMET. 1998.

[7] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008.

[8] M. Dobrescu and K. Argyraki. Software Dataplane Verification. In NSDI, 2014.

[9] ETSI. Network Functions Virtualisation. Retrieved 07/30/2014 [http://portal.etsi.org/NFV/NFV_White_Paper.pdf](http://portal.etsi.org/NFV/NFV_White_Paper.pdf).

[10] S. K. Fayaz, Y. Tobioka, S. Chaki, and V. Sekar. BUZZ: Testing Contextual Policies in Stateful Data Planes. Technical Report CMU-CyLab-14-013, CMU CyLab, 2014.

[11] N. Feamster. Practical verification techniques for wide-area routing. In HotNets, 2004.

[12] N. Feamster and H. Balakrishnan. Detecting bgp configuration faults with static analysis. In NSDI, 2005.

[13] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for software-defined networks. IEEE Communications Magazine, 51(2):128–134, 2013.

[14] A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. In PLDI, pages 483–494, 2013.

[15] S. Itzhaky, A. Banerjee, N. Immerman, O. Lahav, A. Nanevski, and M. Sagiv. Modular reasoning about heap paths via effectively propositional formulas. In POPL, 2014.

[16] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In NSDI, 2013.

[17] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In NSDI, 2013.

[18] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking using Header Space Analysis. NSDI, 2013.

[19] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In NSDI, 2012.

[20] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In NSDI, 2013.

[21] T. Koponen, K. Amidon, P. Ballard, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stibring, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. NSDI, 2014.

[22] K. Korovin. Non-cyclic sorts for first-order satisfiability. In P. Fontaine, C. Ringeissen, and R. Schmidt, editors, Frontiers of Combining Systems, volume 8152 of Lecture Notes in Computer Science, pages 214–228. Springer Berlin Heidelberg, 2013.

[23] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT Way for OpenFlow Switch Interoperability Testing. In CoNEXT, 2012.

[24] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. DNA pairing: Using differential network analysis to find reachability bugs. Technical report, Microsoft Research, 2014. research.microsoft.com/pubs/215431/paper.pdf.

[25] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In SIGCOMM, 2011.

[26] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with Anteater. In SIGCOMM, 2011.
[27] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. A balance of power: Expressive, analyzable controller programming. NSDI, 2014.

[28] R. Piskac, L. M. de Moura, and N. Bjørner. Deciding Effectively Propositional Logic using DPLL and substitution sets. J. Autom. Reasoning, 44(4), 2010.

[29] R. Potharaju and N. Jain. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In IMC, 2013.

[30] R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In Security and Privacy, 2000.

[31] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In CAV, 2006.

[32] D. Sethi, S. Narayana, and S. Malik. Abstractions for model checking sdn controllers. In FMCAD, 2013.

[33] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else’s problem: Network processing as a cloud service. In SIGCOMM, 2012.

[34] R. Skowyra, A. Lapets, A. Bestavros, and A. Kfoury. A verification platform for sdn-enabled applications. In HiCoNS, 2013.

[35] A. Sosnovich, O. Grumberg, and G. Nakibly. Finding security vulnerabilities in a network protocol using parameterized systems. In Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, pages 724–739, 2013.

[36] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford. HotSwap: Correct and Efficient Controller Upgrades for Software-Defined Networks. In HOTSDN, 2013.

[37] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: simplifying sdn programming using algorithmic policies. SIGCOMM, 2013.

[38] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In NSDI, 2014.