Verifying Reachability in Networks with Mutable Datapaths

Aurojit Panda* Ori Lahav† Katerina Argyraki‡ Mooly Sagiv♢ Scott Shenker*♠
*UC Berkeley †MPI-SWS ‡EPFL ♢TAU ♠ICSI

ABSTRACT

Recent work has made great progress in verifying the forwarding correctness of networks [19–21, 26]. However, these approaches cannot be used to verify networks containing middleboxes, such as caches and firewalls, whose forwarding behavior depends on previously observed traffic. We explore how to verify reachability properties for networks that include such "mutable datapath" elements. We want our verification results to hold not just for the given network, but also in the presence of failures. The main challenge lies in scaling the approach to handle large and complicated networks, We address by developing and leveraging the concept of slices, which allow network-wide verification to only require analyzing small portions of the network. We show that with slices the time required to verify an invariant on many production networks is independent of the size of the network itself.

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, network operators have long relied on best-guess configurations 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 reachability, isolation, and other behavioral constraints, there is growing interest in developing rigorous verification tools that can ensure that these constraints will be enforced by the network configuration. The first generation of such tools – Anteater [26], Veriflow [21], and HSA [19,20] – provide highly efficient (in fact, near real-time) checking of reachability (and, conversely isolation) properties and detect anomalies such as loops and black holes. This technical advance represents an 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, firewalls often rely on outbound "hole-punching" to allow hosts to establish flows to the outside world, and content caches forward differently based on whether they have previously cached the desired content. We refer to network elements whose forwarding behavior can be altered by datapath activity as having a "mutable datapath" (in contrast with static datapaths whose behavior is fixed until the control plane intervenes), and additional examples of such elements include WAN optimizers, deep-packet-inspection boxes, and load balancers. In short, the behavior of a static datapath is only a function of its configuration, while the behavior of a mutable datapath also depends on the entire packet history that it has seen.

While classical networking often treats middleboxes as an unfortunate and rare occurrence in networks, in reality middleboxes are the most viable way to incrementally deploy new network functionality. Operators have turned to middleboxes to such a great extent that a recent study [37] of fifty-seven 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 can have mutable datapaths that would not conform to the models used in the recently developed network verification tools. In addition, the rise of Network Function Virtualization (NFV) [12], 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 mutable datapaths (and hereafter, when referring to such elements we will call them middleboxes). Moreover, not only are middleboxes prevalent, but they are often responsible for network problems. A recent two year study [34] 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.

The goal of this paper is to extend the notion of verification to networks containing mutable datapaths, so that such middlebox misconfiguration problems can be prevented.[1] Further, these techniques should ensure correctness even in the presence of failures, a requirement not addressed by any of the existing network verification tools. Our basic approach, which we call Verification for Middlebox Networks (VMN) is

[1] While we are extending the class of networks – to those including mutable datapaths – we are *not* significantly expanding the class of invariants to be checked; just as in the earlier works, we are focusing on reachability and isolation.

simple: given a topology containing both mutable datapaths (*e.g.,* middleboxes) and static ones (*e.g.,* routers), we derive a logical formula that model the network as a whole. We then add logical formulas derived from the specified invariants so that an invariant holds if and only if there is no satisfying assignment for this set of logical formulas as a whole. As described so far, this is a straightforward application of standard program verification techniques to networks.

However, naïvely applied, this approach would not scale: middlebox code is complex, and checking even simple invariants in modest-sized networks would be intractable. *Thus, our focus is on how to scale this approach to large networks.* VMN uses four techniques to scale verification:

1. Limited invariants: Rather than deal with an arbitrary set of invariants, we focus on two specific categories that are the dominant concerns for network operators. First, we look at invariants describing the set of middleboxes (more generally as a DAG of middleboxes) packets should flow through (*e.g.,* all http traffic should pass through a firewall then a cache); we call these *pipeline* invariants. Second, we also consider invariants that address reachability/isolation between hosts (at the packet or content level), such as packets from host A should not reach host B (and hereafter we will call these reachability invariants). Our contributions relate to verifying reachability invariants, and we rely on established techniques to verify pipeline invariants.

2. Simple high-level middlebox models: One approach to verifying networks with middleboxes would be to use their full implementation to determine their behavior. This is infeasible for two reasons: (i) most commonly deployed middleboxes are proprietary, and we do not have access to their code, and (ii) model checking even one such box for even the simplest invariants would not scale.[2] Therefore, we model middleboxes using a simple abstract forwarding model and a set of abstract packet classes used by this model. We do not model the packet classification algorithm in middleboxes, and instead rely on an oracle to classify packets.[3] The forwarding models can typically be derived from a general description of the middlebox's behavior and can be easily analyzed using standard techniques.

3. Modularized network models: Networks contain elements with static datapaths and elements with mutable datapaths. Rather than consider them all within the one verification 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 reachability 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 mutable datapaths.

4. Symmetries and Common Cases If middleboxes are *flow-parallel* or *origin-agnostic* (both to be defined later, and most middleboxes fall into one or both categories), we can perform network-wide verification by examining only a small portion of the network. This allows us to scale verification of a single invariant to large networks. Furthermore, operational networks exhibit a great deal of symmetry in how they are structured and in the policies they enforce. We exploit this symmetry to reduce the total time taken for verifying all invariants in the network. The combination of these two observations allows us to verify the correctness of large networks in a few seconds.

Prior work, particularly Buzz [13], have described network tools (akin to ATPG [45]) that can be applied to networks with middleboxes; they generate test packets that (when sent) efficiently explore whether or not invariants are violated. In contrast, VMN provides mechanisms for verifying (akin to HSA [19]) reachability invariants in networks with middleboxes. Our contributions also include slicing, which allows verification to scale to arbitrarily large networks (through the use of *slices*), and checking whether invariant hold during failures. We provide a more detailed comparison between VMN and other systems in §6.

In the next section, we discuss all four of these steps more formally, and then in §3 we provide an overview of VMN. We discuss our strategy for performing verification on smaller, fixed size subnetworks for scalability in §4. In §5 we evaluate VMN by verifying invariants for a variety of real-world scenarios. Finally we conclude in §6 and §7 with a discussion of related work and a brief summary.

2 Our Approach

In this section we provide an overview of our approach.

2.1 Invariants

The purpose of verification is to test whether some properties (*invariants*) hold for a given network, where a network is defined by both its *topology* (location of routers/switches and middleboxes) and its *configuration* (routing tables and middlebox settings, including how both routers and middleboxes respond to failures). The verification techniques employed necessarily depend on the nature of the invariants to be checked. Thus, to understand the verification problem we are tackling, we must first specify the class of invariants we consider.

Verifying arbitrary invariants for networks with mutable datapaths is undecidable[4] [43], which is why we focus more narrowly on pipeline and reachability invariants for various

[2]This assertion does not contradict the results in [11], which we discuss later in the paper.

[3]A third reason we do not apply model checking to the full implementation, which we discuss in the next section, is that there is a semantic mismatch between raw middlebox code and operator-specified invariants, which are described in terms of basic abstractions. In fact, this mismatch is what led us to the combination of an Oracle and an abstract model.

[4]Middleboxes render the network Turing complete, so verifying certain invariants is equivalent to solving the halting problem.

classes of packets that are defined in terms of hosts, users, source/destination addresses, ingress/egress ports, flows, content, application, whether or not a packet is "malicious" (as decided, for example, by a DPI box), whether or not a packet belonging to this flow has been seen before, and other concepts. These invariants can refer to the current network configuration, or be predicated on one or more failures in the network (*i.e.,* one can insist that reachability or pipeline invariant not only hold for the current network, but for all single failures in that network).

2.2 High-Level Middlebox Models

Some invariants are defined in terms of packet classes based only on *intrinsic* information — such as physical ports and header fields — that can be precisely defined by network operators. However, operators frequently rely on invariants defined in terms of higher-level abstractions — such as what user a flow belongs to, what application sent a flow, whether the packet or flow is malicious, etc. — that depend on other context (for instance authentication services implemented outside of a particular middlebox), and can often not be precisely defined (whether a packet is malicious depends on the current set of CVEs [28] and various heuristics). Using high-level abstractions enables operators to express their intent (such as to drop all malicious traffic, or drop all traffic from a given user) without having to specify, within the invariant itself, the precise mechanisms used to define those abstractions. For example, an operator may wish to drop all Skype traffic, but does not know (or care) about the precise mechanisms an application-level firewall uses to identify such traffic. Therefore, to reflect how things are currently done when configuring middleboxes or expressing policies (*e.g.,* Congress [31]), we allow invariants to be defined in terms of these higher-level abstractions.

How do we model middleboxes when their code is necessarily in terms of intrinsic information but the invariants can be in terms of higher-level abstractions? We choose to describe middleboxes in two stages: we start with a simple abstract forwarding model coupled with a set of abstract packet classes (is a packet malicious, is it a part of a Skype flow, etc.) and any requirements for classifying packets into this type (*e.g.,*, to determine whether a packet belong to a Skype connection the middlebox implementation needs to see all packets in a flow). VMN then augments this model by adding a *classification oracle* which is responsible for classifying packets into these packet classes. The abstract forwarding model describes how the middlebox operates on a packet (*i.e.,* whether and where it is forwarded, and whether the header is rewritten) given the intrinsic information (*i.e.,* packet header and port information), the entire packet history (since middlebox datapaths can be mutable, their behavior can depend on packet history), and the abstractions it is assigned by the oracle (is the packet part of a Skype flow?). The *type* of middlebox (*e.g.,* firewall or load balancer) describes the basic behaviors supported by the middlebox (for instance, which abstractions it supports, whether it rewrites packet headers, etc.), but the *configuration* of the middlebox dictates to which class of packets these behaviors are applied.

The task of verification that we address in this paper is whether the invariants are obeyed *assuming that middlebox implementation can correctly classify packets into abstract types.* That is, we verify network and middlebox configuration, not the implementation of the abstractions. Clearly, if a middlebox cannot correctly classify traffic (*e.g.,* Skype traffic) an invariant might not hold. However, this error is not caused by network configuration but rather by bugs in the middlebox implementation. While verifying middlebox implementation is an important task, it is beyond the scope of this paper. Here we focus solely on whether the overall network configuration upholds invariants assuming middleboxes are correctly implemented.

2.3 Modularized Network Models

The tools developed for static datapaths [19, 20] not only verify invariants, but also summarize the behavior of such networks as transfer functions. A transfer function for a network of static datapaths maps an incoming packet on a physical port at the edge of the network to one or more outgoing packets (perhaps with rewritten header fields) departing out one or more physical ports at the edge of the network. Thus, we can consider our network as a set of elements with mutable datapaths tied together by transfer functions which represent the behavior of the static datapath portion of the network.[5] When a failure occurs, routing in the static datapaths will change, producing a new transfer function. Rather than model the details of the routing algorithm, we assume we are given a function mapping failure conditions to these new transfer functions (*e.g.,* a list of backup paths taken in response to failures).

The glue provided by these transfer functions is precisely what is responsible for enforcing pipeline invariants. In its simplest incarnation, a pipeline invariant takes the form: *all incoming packets with a certain class of headers must pass through the sequence of middleboxes* mb_1, mb_2, mb_3, *... before being delivered to the intended destination*. More complicated pipeline invariants involve a DAG of middleboxes and specify the appropriate branching at each step (*e.g.,* all `http` packets leaving the firewall go to the load balancer, while all other traffic goes directly to the destination). 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). In what follows, we will assume that all packets belonging to the same flow are processed by the same physical pipeline (this can be easily enforced in existing networks).

Once we have decomposed the network into a set of middleboxes connected by transfer functions, checking these

[5] If the static datapaths lead to a loop for a particular packet, we raise an exception (so the network operator is aware of it) and treat the packet as dropped.

pipeline invariants is straightforward using existing network verification tools. The reachability invariants are more difficult, as we discuss next.

2.4 Scaling Verification

For moderately sized networks, we can use the techniques discussed above to generate logical formulas modeling network behavior, and other formulas corresponding to the invariants that need to be verified. We can then use an SMT solver (*e.g.,* Z3 [10]) to check if the invariants hold for the provided network or not. However as the scale of the network increases, the SMT solver has to account for an exponentially larger state space, slowing down or preventing verification.

Inspired by compositional verification [17, 27] (which allows the results from verifying components of a program to be combined to reason about the program as a whole) we have identified a general class of subnetworks, which we refer to as *slices*, where any invariant that only references middleboxes or endhosts contained in a slice holds for the entire network if and only if it holds for the slice. Therefore, we can scale network verification if given a network and an invariant, we can find a slice whose size is independent of the size of the network and which contains all middleboxes or endhosts referenced by the invariant.

While slices help us rapidly verify that an individual invariant holds in a network, a network might implement several invariants. Prior work [33] has observed that many operational networks have symmetric topologies and policies. Since the proof generated while verifying that a particular invariant holds also applies to all other symmetric invariants, this greatly reduces the time taken to verify the behavior of a network allowing us to scale verification to extremely large operational networks.

3 System Design

We model network behavior in discrete timesteps. During each timestep a previously sent packet can be delivered to a node (middlebox or host), a host can generate a new packet that enters the network, a middlebox can process a previously received packet, a failure can occur, or a previously failed node can recover. We do not attempt to model the *likely* order of these various events, but instead consider all such orders in search of invariant violations. To do so, we invoke a *scheduling oracle* that assigns a single event to each timestep, subject to the constraint that ordering (both for receiving and processing) is maintained for packets sent on the same link.

3.1 Overview

VMN accepts as input a set of middlebox models (§3.4), and a set of transfer functions (§3.5) with a mapping between failure conditions and transfer functions. VMN builds on Z3 [29] a state of the art SMT solver. SMT solvers accept as input a set of boolean formulae expressed in terms of a set of variables, and then either find an assignment for the variables such that the conjunction of these formulae is true (a satisfying assignment) or a proof that no such assignment exists. In this section we present our technique for converting a network and set of invariants to a set of axioms, and producing a logical formulae including these axioms. We can then use Z3 to check satisfiability for these formulae, where a satisfying assignment indicates that an invariant is violated by the network.

In VMN middleboxes and networks are modeled using quantified formula, which are axioms describing how received packets are treated, while the classification and scheduling oracles are modeled using variables. In addition to the network model and oracles, we also provide Z3 with the negation of the invariant, which we specify in terms of a set of packets. Finding a satisfiable assignment to these formulae is equivalent to Z3 finding a set of oracle behaviors that result in the invariant being violated, and proving the formulae unsatisfiable is equivalent to showing that no oracular behavior can result in the invariants being violated.

The search problem solved by SMT solvers is undecidable in general, and they rely on heuristics and timeouts to ensure their search procedure terminates (in which case they are unable to determine if a problem is satisfiable or not). Naïvely generating middlebox and network formulae might yield formulae in an undecidable logic, preventing successful verification. Therefore, one of the core contribution of VMN lies in producing logical formulae for a wide range of networks (and a variety of middleboxes) and invariants in a weak logic that is expressible in Z3 and for which verification succeeds in practice. In the rest of this section we explore these models and formula in greater depth.

3.2 Notation

We begin by presenting the notation used in this section. We express our models and invariants using a simplified form of linear temporal logic (LTL) [24] of events, with past operators. We restrict ourselves to safety properties, and hence only need to model events occurring in the past or events that hold globally for all of time. We use LTL for ease of presentation; VMN automatically converts LTL formulas into first-order logic (as required by Z3) by explicitly quantifying over time.

Our formulas are expressed in terms of three events: $snd(s,d,p)$, the event where a *node* (end host, switch or middlebox) s sends *packet* p to *node* d; and $rcv(d,s,p)$, the event where a node d receives a packet p from node s, and $fail(n)$, the event where a node n has failed. Each event happens at a timestep and logical formulas can refer either to events that occurred in the past (represented using $\blacklozenge$) or properties that hold at all times (represented using $\Box$). For example,

$$\forall d,s,p:\ \Box(rcv(d,s,p) \implies \blacklozenge snd(s,d,p))$$

says that at all times, any packet p received by node d from node s must have been sent by s in the past.

Similarly,

$$\forall p:\ \Box\neg rcv(d,s,p)$$

indicates that d will never receive any packet from s.

Header fields and abstract packet classes are represented using functions, *e.g.,* $src(p)$ and $dst(p)$ represent the source

Listing 1: Model for a learning firewall

```
1  @FailClosed
2  class LearningFirewall (acl: Set[(Address,
       ↪ Address)]) {
3    val established : Set[Flow]
4    def model (p: Packet) = {
5      when established.contains(flow(p)) =>
6        forward (Seq(p))
7      when acl.contains((p.src, p.dest)) =>
8        established += flow(p)
9        forward(Seq(p))
10     _ =>
11       forward(Seq.empty)
12   }
13 }
```

and destination address for packet p, and $skype?(p)$ returns true if and only if p belongs to a Skype session.

3.3 Reachability Invariants

Reachability invariants can be be generally specifies as:

$$\forall n, p: \ \Box\neg(rcv(d,n,p) \wedge predicate(p)),$$

which says that node d should never receive a packet p that matches $predicate(p)$. The $predicate$ can be expressed in terms of packet-header fields, abstract packet classes and past events, this allows us to express a wide variety of network properties as reachability invariants, *e.g.,*:

- Simple isolation: node d should never receive a packet with source address s. We express this invariant using the *src* function, which extracts the source IP address from the packet header:

$$\forall n, p: \Box\neg(rcv(d,n,p) \wedge src(p) = s).$$

- Flow isolation: node d can only receive packets from s if they belong to a previously established flow. We express this invariant using the $flow$ function, which computes a flow identifier based on the packet header:

$$\forall n_0, p_0, n_1, p_1 : \Box\neg(rcv(d,n_0,p_0) \wedge src(p_0) = s \wedge \neg(\blacklozenge snd(d,n_1,p_1) \wedge flow(p_1) = flow(p_0))).$$

- Data isolation: node d cannot access any data originating at server s, this requires that d should not access data either by directly contacting s or indirectly through network elements such as content cache. We express this invariant using an *origin* function, that computes the origin of a packet's data based on the packet header (*e.g.,* using the `x-http-forwarded-for` field in HTTP):

$$\forall n, p: \Box\neg(rcv(d,n,p) \wedge origin(p) = s).$$

In addition, VMN can verify several other invariants, including whether packets traverse a certain link or middlebox.

3.4 Modeling Middleboxes

Middleboxes in VMN are modeled using a high-level loop-free, event driven language. Restricting the language so it is loop free allows us to ensure that middlebox models are expressible in first-order logic (and can serve as input to Z3). We use the event-driven structure to translate this code to logical formulae (axioms) encoding middlebox behavior.

Middlebox models are specified as a class containing the *abstract packet classes* the middlebox depends on, its *forwarding model*, and its *failure behavior*. *Abstract packet classes* are specified as a set of function prototypes. Optionally, models can also specify input constraint that must be met for the implementation to correctly identify that a packet belongs to a particular class, and output constraints restricting the set of classes a packet can belong to. For example, an application firewall might specify an abstract packet class for each application (*e.g.,* `skype?`, `jabber?`), specify that correct identification requires all packets in a flow to go through the same middlebox instance, and specify that a packet can belong to at most one application class (a packet cannot be both a Skype packet and a Jabber packet). Middlebox *forwarding models* are specified as functions which dictate how packets are modified and whether they are forwarded or dropped. Complex packet modification, *e.g.,* encryption or compression, are modeled as replacing the appropriate packet header field (or payload) with a random value, this provides sufficient fidelity for checking reachability invariants. Finally, middlebox failure behavior is specified either explicitly in the forwarding model, using the `fail` predicate provided by VMN, or implicitly by specifying whether a middlebox fails-closed (*i.e.,* packets are dropped during middlebox failures) or fails-open (*i.e.,* all received packets are forwarded unmodified during failure). We provide examples of such specification below.

Listing 1 shows the specification for a stateful firewall. The model accepts a set of ACLs (`acl` on Line 2) as configuration, and maintains flow state (in the `established` variable defined on Line 3). On receiving a packet from an established flow, the firewall forwards the packet (Line 5 and 6), otherwise it checks to see if the packet is permitted by the firewall configuration (Line 6–10). The firewall forwards the packet if permitted and drops it otherwise. The `@FailClosed` annotation on line 1 indicates that the firewall fails closed, and packets are dropped during failure.

Similarly, Listing 2 shows the model for a NAT. In this example we explicitly model failure behavior (Line 6), and the NAT drops packets when failed. We also modify the packet's source and destination port (Line 10–11, 14–15, and 20–21) as a part of the forwarding behavior. We assign ports to new flows at random by calling the `remapped_port` (line 2) method.

VMN translates these high-level specifications into a set of parametrized axioms (the parameters allow more than one instance of the same middlebox to be used in a network). For instance, Listing 1 results in the following axioms:

$$\begin{aligned}
&\textbf{established}(flow(p)) \implies (\blacklozenge((\neg fail(\mathbf{f})) \wedge (\blacklozenge rcv(\mathbf{f},p)))) \\
&\qquad\qquad \wedge \textbf{acl}(src(p), dst(p)) \\
&send(\mathbf{f},p) \implies (\blacklozenge rcv(\mathbf{f},p)) \\
&\qquad \wedge (\textbf{acl}(src(p), dst(p)) \\
&\qquad \vee \textbf{established}(flow(p)))
\end{aligned}$$

The bold-faced terms in this axiom are parameters: for each stateful firewall that appears in a network, VMN adds a new

Listing 2: Model for a NAT

```
class NAT (nat_address: Address){
 abstract remapped_port (p: Packet): int
 val active : Map[Flow, int]
 val reverse : Map[port, (Address, int)]
 def model (p: Packet) = {
   when  fail(this) =>
     forward(Seq.empty)
   dst(p) == nat_address =>
     (dst, port) = reverse[dst_port(p)];
     dst(p) = dst;
     dst_port(p) = port;
     forward(Seq(p))
   active.contains(flow(p)) =>
     src(p) = nat_address;
     src_port(p) = active(flow(p));
     forward(Seq(p))
   _ =>
     address = src(p);
     port = src_port(p)
     src(p) = nat_address;
     src_port(p) = remapped_port(p);
     active(flow(p)) = src_port(p);
     reverse(src_port(p)) = (address, port);
     forward(Seq(p))
 }
}
```

axiom by replacing the terms **f**, **acl** and **established** with a new instance specific term. The first axiom says that the **established** set contains a flow if a packet permitted by the firewall policy (**acl**) has been received by **f** since it last failed. The second one states that packets sent by **f** must have been previously received by it, and are either pr emitted by the **acl**'s for that firewall, or belong to a previously established connection. The axioms generated for the NAT are similar, and are elided due to space constraints.

We require users of VMN to provide middlebox models, however our models are at a high level and depend only on the type of the middlebox, not its placement or implementation details. Previous studies have found that only a limited number of middlebox types are widely deployed [8] in production networks[6], and we believe that in a majority of cases users of our tool can reuse existing models.

3.5 Modeling Networks

VMN uses *transfer functions* which were previously developed by HSA [20] and VeriFlow [21] to specify a network's forwarding behavior during a particular failure scenario. The transfer function for a network is a function from a located packet (a packet augmented with the network port where it is located) to a set of located packets indicating where the packets are next sent. For example, the transfer function for a network with 3 hosts A (with IP address a), B (with IP address b) and C (with IP address c) is given by:

$$f(p, port) \equiv \begin{cases} (p, A) & \text{if } dst(p) = a \\ (p, B) & \text{if } dst(p) = b \\ (p, C) & \text{if } dst(p) = c \end{cases}$$

VMN assumes switch forwarding tables are static, however they might change depending on the failure scenario. Therefore, rather than accepting a single static network topology and configuration as input, VMN accepts a topology and forwarding table corresponding to each failure scenario. Given the topology and switch forwarding tables used by the network in a particular failure scenario, VMN uses VeriFlow to compute a transfer function. In this computed transfer function, all ports correspond to either middleboxes or end-hosts, *i.e.,* the transfer function models the network as a set of end-hosts and middleboxes connected to a single switch. VMN translates this transfer function to axioms by introducing a single pseudo-node (Ω) representing the network, and deriving a set of axioms for this pseudo-node from the transfer function and failure scenario. For example, the previous transfer function is translated to the following axioms ($fail(X)$ here represents the specified failure model).

$$\forall n, p : \Box fail(X) \wedge \ldots snd(A, n, p) \implies n = \Omega$$
$$\forall n, p : \Box fail(X) \wedge \ldots snd(\Omega, n, p) \wedge dst(p) = a \implies n = A \wedge \blacklozenge \exists n' : rcv(n', \Omega, p)$$

VeriFlow (and HSA) can only produce transfer functions when the topology and forwarding table for a network are loop-free. VMN therefore throws an exception when a static forwarding loop is encountered. Not allowing loops in the forwarding logic is also important for allowing us to express network axioms in first-order logic and helps ensure VMN's verification process is decidable.

In addition to the axioms for middlebox behavior and network behavior, VMN also adds axioms restricting the oracles' behavior, *e.g.,* we add axioms to ensure that any packet delivery event scheduled by the scheduling oracle has a corresponding packet send event, and we ensure that new packets generated by hosts are well formed.

3.6 Limitations

To ensure verification is practical and tractable, our models of networks and middleboxes are necessarily abstract. This imposes some limitations for the results returned by VMN. Firstly, we do not verify the classification logic in a middlebox implementation, our verification results are conditioned on packets being correctly classified by the middlebox. Therefore, our results might be wrong when classification logic is incorrectly implemented. This is a separable problem: VMN does not obviate the need to verify and test individual middleboxes, it just provides a mechanism to verify the behavior of combined middleboxes. Providing tools to test or verify individual middleboxes is outside scope of our work.

Secondly, we do not have complete semantic information about abstract packet classes, and this can result in VMN reporting false positives (*i.e.,* invariant violations) where none

[6]The existence of a limited number of middlebox types does not limit the number of deployed middleboxes. Networks commonly include several middleboxes belonging to the same type, this might be for resilience, improving network performance and to reduce the load on each middlebox.

exist. For example, consider a network with two application specific firewalls, one that can identify Skype traffic, and another that can identify streaming audio services. A priori, VMN has no information indicating that these packet classes are mutually exclusive, and will consider packets which meet both criterion when looking for invariant violations. This can be solved by augmenting VMN's models with logical constraints encoding these assumptions, however we do not currently include such constraints.

Finally, our models do not contain semantics for complex packet modifications (*e.g.,* encryption, compression, etc.), and instead just change the affected packet to a random value. Similar to the previous cases this can also result in false positives in the same way as above.

Most of these limitations are fundamental, network verification without the use of abstractions is intractable, and is impractical with large models. Our choices allow us to provide useful verification, as shown §5, and in most practical cases we observed no false positives.

4 Scaling Verification

Z3 (and other SMT solvers) rely on heuristics and timers to ensure satisfiability checking terminates, and cannot always prove (or disprove) satisfiability of large sets of formulae. The size of formulae produced by the techniques in §3 are proportional to the size of the network being verified, and therefore cannot be applied to large networks. Scaling therefore requires that the size of the formulae generated be independent of network size. We rely on *network slices* as described here.

We begin by providing an informal overview of network slices, a more formal description is available in our technical report.[7] Given a network N, a subnetwork Ω is a network formed from a subset of N's nodes (middleboxes, hosts and switches) and links. All packets sent by hosts in subnetwork Ω and received by hosts in Ω are said to belong to Ω. We say Ω is closed under forwarding if and only if all packets belonging to Ω are only forwarded to nodes in Ω.

Define a network N's state to be the cartesian product of the state in all middleboxes in N. We say some state s is reachable in N if and only if there exists a schedule (given by the output of the scheduling oracle) at the end of which the network has state s. A subnetwork Ω of network N is then closed under state if and only if there exists an equivalence relation between states in Ω and states in N such that for all states reachable in N, the equivalent state is reachable in Ω, and vice versa.

A slice is a subnetwork that is both closed under forwarding and state. Any invariant referencing only nodes and packets belonging to a slice holds in the original network if and only if it holds in the slice. Consider an invariant I that is violated in some network N. Proving that an invariant is violated is equivalent to finding a schedule S (*i.e.,* a sequence of events) which lead to the invariant being violated. Now consider Ω, a slice of N, such that I only references nodes and packets belonging to Ω. Intuitively, the closure properties imply that there exists a schedule S' for Ω that is equivalent to S. Furthermore, this equivalence implies that S' also leads to I being violated in Ω. Finally, note that Ω is a subnetwork of N, and hence any schedule for Ω is also a schedule for N.

[7] Anonymized for double blind submission, we can provide proofs and other details on request.

4.1 Finding Slices

Networks with arbitrary middleboxes need not have slices smaller than the network as a whole. However, we find there is a class of networks that do have slices that do not grow with the size of the overall network. These special networks, which we now focus on, obey the following conditions: (a) any middleboxes used in these networks be *flow-parallel*, or *origin-agnostic*, (b) network policies be such that we can divide hosts in the network into a set of policy equivalence classes, two hosts are in the same equivalence class if all packets sent and received by them traverse the same set of middlebox types, and are treated according to the same policy, (c) the number of policy classes be independent of the size of the network, (d) the network's forwarding graph is finite, and (e) the size of forwarding graphs is independent of network size. We define our restrictions in greater detail below.

A middlebox is *flow-parallel* if middlebox state is partitioned by flows, and a flow's state is accessed only when processing that flow; *e.g.,* stateful firewalls maintain state about whether a particular flow is allowed, however this state does not affect other flows, nor is it updated by any other flow. A middlebox is *origin-agnostic* if it is not flow-parallel (*i.e.,* state is shared across flows) and its behavior is agnostic to which flows (and hence hosts) instantiated the state. For example, the behavior of content-caches often does not depend on the connection that led to content being cached.

Any subnetwork that contains only *flow-parallel* middleboxes and is closed under forwarding is also closed under state; the set of packets belonging to the subnetwork can be naturally mapped to a set of flows belonging to the subnetwork, and the state for those flows can only be affected by nodes in the subnetwork. Therefore, finding a slice in a network containing only flow-parallel middleboxes is equivalent to finding a subnetwork that is closed under forwarding. Therefore, verifying an invariant in a network with only *flow-parallel* middleboxes only requires that we consider a subnetwork that is closed under forwarding and includes all nodes specified by the invariant. Since we assumed the size of the forwarding graph is finite and independent of network size, this means that the slices used to verify invariants in such networks are also finite.

A subnetwork that is closed under forwarding and contains only *origin-agnostic* or *flow-parallel* middleboxes is closed under state if and only if it includes a node from each policy class. This is because all nodes in the same policy class are equivalent for *origin-agnostic* middleboxes, *i.e.,* the middlebox cannot distinguish between hosts in the same equivalence class, ensuring that the slice is closed under state.

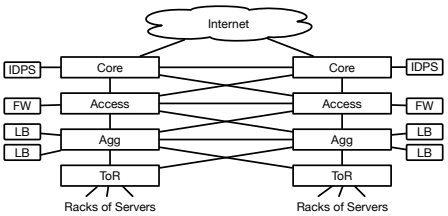


Figure 1: Topology for a datacenter network with middleboxes from [34]. The topology contains firewalls (**FW**), load balancers (**LB**) and intrusion detection and prevention systems (**IDPS**).

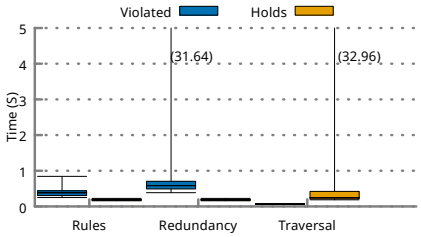


Figure 2: Time taken to verify each network invariant for scenarios in §5.1. We show time for checking both when invariants are violated (Violated) and verified (Holds).

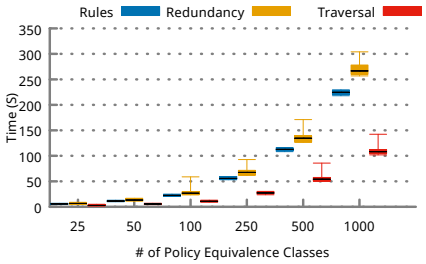


Figure 3: Time taken to verify all network invariants as a function of policy complexity for §5.1. The plot presents minimum, maximum, 5^{th}, 50^{th} and $95th$ percentile time for each.

Therefore, when networks meet our requirements, all reachability invariants can be verified in network slices whose size is independent of network size.

Finally, we note that our restrictions are commonly met by deployed networks: previous studies have shown that firewalls, proxies and IDSes are the most commonly deployed middlebox types; of these firewalls are *flow-parallel*, most proxies are *origin-agnostic* and many IDSes are off path and do not affect reachability invariants. Furthermore, IDSes can be safely treated as *origin-agnostic* middleboxes in VMN, without loss in verification fidelity. For ease of management, network policy in large deployed networks is commonly expressed in terms of policy classes, and the forwarding graph is restricted for performance. Therefore, we do not believe these restrictions pose a severe challenge for VMN. Finally, VMN can still be used to verify moderate sized networks which violate these restrictions.

4.2 Network Symmetry

Slices allow us to scale verification of an individual invariant, however a single network might enforce several invariants, and the number of invariants might grow with network size. However, networks are often symmetric with respect to policy classes, *i.e.,* packets whose source and destination belong to the same policy class traverse the same sequence of middlebox types. When possible VMN takes advantage of this symmetry to reduce the number of invariants to be verified. We say two invariants are symmetric when one can be transformed to another by replacing nodes with other nodes in the same policy class. If an invariant I holds in a symmetric network, then so do all invariants symmetric to I. When networks are symmetric, VMN uses this observation to divide invariants into symmetric groups and proves just one invariant in each symmetry group, allowing us to eliminate many invariant checks.

5 Evaluation

To evaluate VMN we first examine how it would deal with several real-world scenarios and then investigate how it scales to large networks. We ran our evaluation on servers running 10-core, 2.6GHz Intel Xeon processors with 256 GB of RAM. We report times taken when verification is performed using a single core. Verification can be trivially parallelized over multiple invariants. We used Z3 version 4.4.2 for our evaluation. SMT solvers rely on randomized search algorithms, and their performance can vary widely across runs. The results reported here are generated from 100 runs of each experiment.

5.1 Real-World Evaluation

A previous measurement study [34] looked at more than 10 datacenters over a 2 year period, and found that configuration bugs (in both middleboxes and networks) are a frequent cause of failure. Furthermore, the study analyzed the use of redundant middleboxes for fault tolerance, and found that redundancy failed due to misconfiguration roughly 33% of the time. Here we show how VMN can detect and prevent the three most common classes of configuration errors, including errors affecting fault tolerance. For our evaluation we use a datacenter topology (Figure 1) containing 1000 end hosts and three types of middleboxes: stateful firewalls, load balancers and intrusion detection and prevention systems (IDPSs). We use redundant instances of these middleboxes for fault tolerance. For each scenario we report time taken to verify a single invariant (Figure 2), and time taken to verify all invariants (Figure 3); and show how these times grow as a function of policy complexity (as measured by the number of policy equivalence classes). Each box and whisker plot shows minimum, 5^{th} percentile, median, 95^{th} percentile and maximum time for verification.

Incorrect Firewall Rules: According to [34], 70% of all reported middlebox misconfiguration are attributed to incorrect rules installed in firewalls. To evaluate this scenario we begin by assigning each host to one of a few policy groups.[8] We then add firewall rules to prevent hosts in one group from communicating with hosts in any other group. We introduce misconfiguration by deleting a random set of these firewall rules. We use VMN to identify for which hosts the desired invariant holds (*i.e.,* that hosts can only communicate with other hosts in the same group). Note that all middleboxes in this evaluation are flow-parallel, and hence the size of a slice on which invariants are verified is independent of both policy complexity and network size. In our evaluation, we found that VMN correctly identified all violations, and did

[8]Note, policy groups are distinct from policy equivalence class; a policy group signifies how a network administrator might group hosts while configuring the network, however policy equivalence classes are assigned based on the actual network configuration.

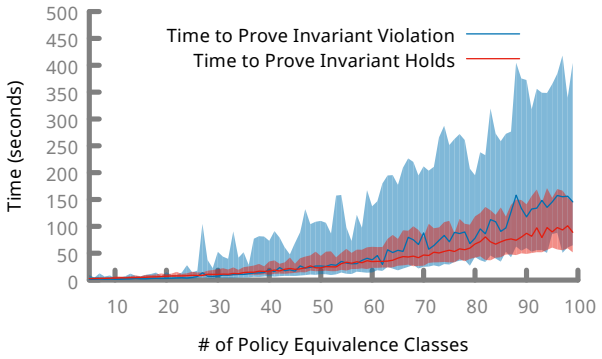


Figure 4: Time taken to verify each data isolation invariant. The shaded region represents the 5^{th}–95^{th} percentile time.

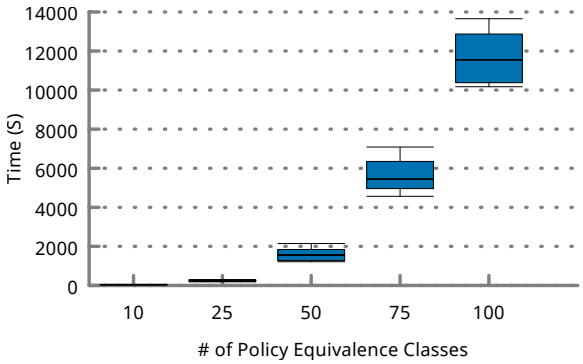


Figure 5: Time taken to verify all data isolation invariants in the network described in §5.2.

not report any false positives. The time to verify a single invariant is shown in Figure 2 under Rules. When verifying the entire network, we only need to verify as many invariants as policy equivalence classes; hosts affected by misconfigured firewall rules fall in their own policy equivalence class, since removal of rules breaks symmetry. Figure 3 (Rules) shows how whole network verification time scales as a function of policy complexity.

Misconfigured Redundant Firewalls Redundant firewalls are often misconfigured so that they do not provide fault tolerance. To show that VMN can detect such errors we took the networks used in the preceding simulations (in their properly configured state) and introduced misconfiguration by removing rules from some of the backup firewall. In this case invariant violation would only occur when middleboxes fail. We found VMN correctly identified all such violations, and we show the time taken for each invariant in Figure 2 under "Redundant", and time taken for the whole network in Figure 3.

Misconfigured Redundant Routing Another way that redundancy can be rendered ineffective by misconfiguration is if routing (after failures) allows packets to bypass the middleboxes specified in the pipeline invariants. To test this we considered, for the network described above, an invariant requiring that all packet in the network traverse an IDPS before being delivered to the destination host. We changed a randomly selected set of routing rules so that some packets would be routed around the redundant IDPS when the primary had failed. VMN correctly identified all such violations, and we show times for individual and overall network verification under "Traversal" in Figures 2 and 3.

We can thus see that verification, as provided by VMN, can be used to prevent many of the configuration bugs reported to affect today's production datacenters. Moreover, the verification time scales linearly with the number of policy equivalence classes (with a slope of about three invariants per second). We now turn to more complicated invariants involving data isolation.

5.2 Data Isolation

Modern data centers also run storage services such as S3 [2], AWS Glacier [1], and Azure Blob Store [3]. These storage services must comply with legal and customer requirements [32] limiting access to this data. Operators often add caches to these services to improve performance and reduce the load on the storage servers themselves, but if these caches are misplaced or misconfigured then the access policies could be violated. VMN can verify these data isolation invariants.

To evaluate this functionality, we used the topology (and correct configuration) from §5.1 and added a few content caches by connecting them to top of rack switches. We also assume that each policy group contains separate servers with private data (only accessible within the policy group), and servers with public data (accessible by everyone). We then consider a scenario where a network administrator inserts caches to reduce load on these data servers. The content cache is configured with ACL entries[9] that can implement this invariant. Similar to the case above, we introduce configuration errors by deleting a random set of ACLs from the content cache and firewalls.

We use VMN to verify data isolation invariants in this network (*i.e.,* ensure that private data is only accessible from within the same policy group, and public data is accessible from everywhere). VMN correctly detects invariant violations, and does not report any false positives. Content caches are origin agnostic, and hence the size of a slice used to verify these invariants depends on policy complexity. Figure 4 shows how time taken for verifying each invariant varies with the number of policy equivalence classes. In a network with 100 different policy equivalence classes, verification takes less than 4 minutes on average. Also observe that the variance for verifying a single invariant grows with the size of slices used. This shows one of the reasons why the ability to use slices and minimize the size of the network on which an invariant is verified is important. Figure 5 shows time taken to verify the entire network as we increase the number of policy equivalence classes.

[9]This is a common feature supported by most open source and commercial caches.

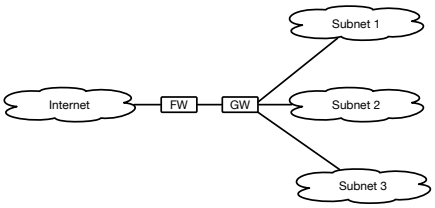


Figure 6: Topology for enterprise network used in §5.3.1, containing a firewall (**FW**) and a gateway (**GW**).

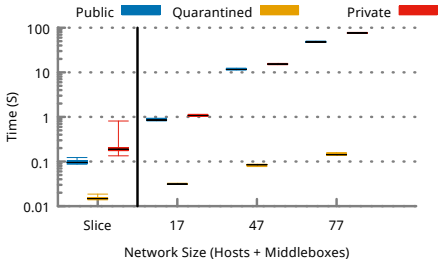


Figure 7: Distribution of verification time for each invariant in an enterprise network (§5.3.1) with network size. The left of the vertical line shows time taken to verify a slice, which is independent of network size, the right shows time taken when slices are not used.

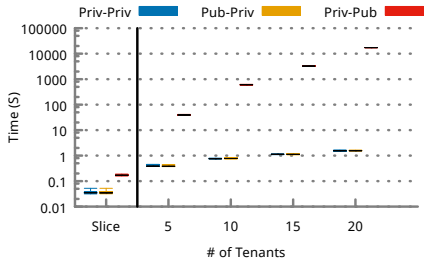


Figure 8: Average verification time for each invariant in a multi-tenant datacenter (§5.3.2) as a function of number of tenants. Each tenant has 10 hosts. The left of the vertical line shows time taken to verify a slice, which is independent of the number of tenants.

5.3 Other Network Scenarios

We next apply VMN to several other scenarios that illustrate the value of slicing (and symmetry) in reducing verification time.

5.3.1 Enterprise Network with Firewall

First, we consider a typical enterprise or university network protected by a stateful firewall, shown in Figure 6. The network interconnects three types of hosts:

1. Hosts in *public* subnets should be allowed to both initiate and accept connections with the outside world.
2. Hosts in *private* subnets should be flow-isolated (*i.e.,* allowed to initiate connections to the outside world, but never accept incoming connections).
3. Hosts in *quarantined* subnets should be node-isolated (*i.e.,* not allowed to communicate with the outside world).

We vary the number of subnets keeping the proportion of subnet types fixed; a third of the subnets are public, a third are private and a third are quarantined.

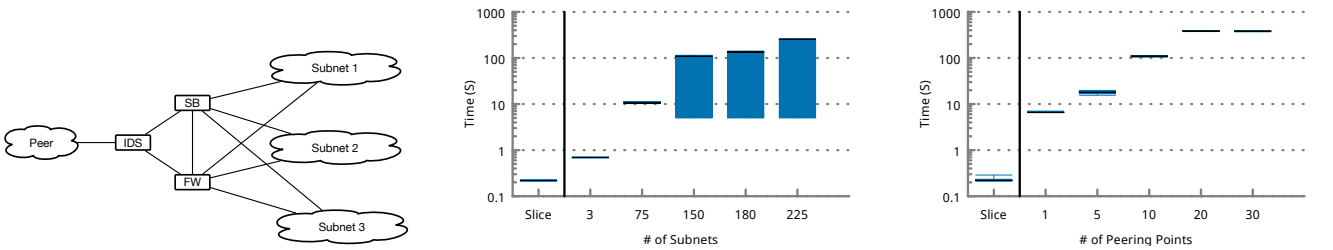
We configure the firewall so as to enforce the target invariants correctly: with two rules denying access (in either direction) for each quarantined subnet, plus one rule denying inbound connections for each private subnet. The results we present below are for the case where all the target invariants hold. Since this network only contains a firewall, using slices we can verify invariants on a slice whose size is independent of network size and policy complexity. We can also leverage the symmetry in both network and policy to reduce the number of invariants that need to be verified for the network. In contrast, when slices and symmetry are not used, the model for verifying each invariant grows as the size of the network, and we have to verify many more invariants. In Figure 7 we show time taken to verify the invariant using slices (Slice) and how verification time varies with network size when slices are not used.

5.3.2 Multi-Tenant Datacenter

Next, we consider how VMN can be used by a cloud provider (*e.g.,* Amazon) to verify isolation in a multi-tenant datacenter. We assume that the datacenter implements the Amazon EC2 Security Groups model [4]. For our test we considered a datacenter with 600 physical servers (which each run a virtual switch) and 210 physical switches (which implement equal cost multi-path routing). Tenants launch virtual machines (VMs), which are run on physical servers and connect to the network through virtual switches. Each virtual switch acts as a stateful firewall, and defaults to denying all traffic (*i.e.,* packets not specifically allowed by an ACL are dropped). To scale policy enforcement, VMs are organized in security groups with associated accept/deny rules. For our evaluation, we considered a case where each tenant organizes their VMs into two security groups:

1. VMs that belong to the *public* security group are allowed to accept connections from any VMs.
2. VMs that belong to the *private* security group are flow-isolated (*i.e.,* they can initiate connections to other tenants' VMs, but can only accept connections from this tenant's public and private VMs).

We also assume that firewall configuration is specified in terms of security groups (*i.e.,* on receiving a packet the firewall computes the security group to which the sender and receiver belong and applies ACLs appropriately). For this evaluation, we configured the network to correctly enforce tenant policies. We added two ACL rules for each tenant's public security group allowing incoming and outgoing packets to anyone, while we added three rules for private security groups; two allowing incoming and outgoing traffic from the tenant's VM, and one allowing outgoing traffic to other tenants. For our evaluation we consider a case where each tenant has 10 VMs, 5 public and 5 private, which are spread across physical servers. These rules result in flow-parallel middleboxes, so we can use fixed size slices to verify each invariant. The number of invariants that need to be verified grow as a function of the number of tenants. In Figure 8 we show time taken to verify one instance of the invariant when slices are used (Slice) and how verification time varies with network size when slices are not used. The invariants checked are: (a) private hosts in one group cannot reach private hosts in another group (Priv-Priv), (b) public hosts in one group cannot reach private hosts in another group (Priv-Pub), and (c) private hosts in one group *can* reach public hosts in another.



(a) (b) (c)

Figure 9: (a) shows the pipeline at each peering point for an ISP; (b) distribution of time to verify each invariant given this pipeline when the ISP peers with other networks at 5 locations; (c) average time to verify each invariant when the ISP has 75 subnets. In both cases, to the left of the black line we show time to verify on a slice (which is independent of network size) and vary sizes to the right.

5.3.3 ISP with Intrusion Detection

Finally, we consider an Internet Service Provider (ISP) that implements an intrusion detection system (IDS). We model our network on the SWITCHlan backbone [41], and assume that there is an IDS box and a stateful firewall at each peering point (Figure 9(a)). The ISP contains public, private and quarantined subnets (with policies as defined in §5.3.1) and the stateful firewalls enforce the corresponding invariants. Additionally, each IDS performs lightweight monitoring (*e.g.,* based on packet or byte counters) and checks whether a particular destination prefix (*e.g.,* a customer of the ISP) might be under attack; if so, all traffic to this prefix is rerouted to a scrubbing box that performs more heavyweight analysis, discards any part of the traffic that it identifies as "attack traffic," and forwards the rest to the intended destination. This combination of multiple lightweight IDS boxes and one (or a few) centralized scrubbing boxes is standard practice in ISPs that offer attack protection to their customers.[10]

To enforce the target invariants (for public, private, and quarantined subnets) correctly, all inbound traffic must go through at least one stateful firewall. We consider a misconfiguration where traffic rerouted by a given IDS box to the scrubbing box bypasses all stateful firewalls. As a result, any part of this rerouted traffic that is *not* discarded by the scrubbing box can reach private or quarantined subnets, violating the (simple or flow-) isolation of the corresponding hosts.

When verifying invariants in a slice we again take advantage of the fact that firewalls and IDSes are flow-parallel.[11] For each subnet, we can verify invariants in a slice containing a peering point, a host from the subnet, the appropriate firewall, IDS and a scrubber. Furthermore, since all subnets belong to one of three policy equivalence classes, and the network is symmetric, we only need run verification on three slices.

We begin by evaluating a case where the ISP, similar to the SWITCHlan backbone has 5 peering points with other networks. We measure verification time as we vary the number of subnets (Figure 9(b)), and report time taken, on average, to verify each invariant. When slices are used, the median time for verifying an invariant is 0.21 seconds, by contrast when verification is performed on the entire network, a network with 250 subnets takes approximately 6 minutes to verify. Furthermore, when verifying all invariants, only 3 slices need to be verified when we account for symmetry, otherwise the number of invariants verified grows with network size.

In Figure 9(c) we hold the number of subnets constant (at 75) and show verification time as we vary the number of peering points. In this case the complexity of verifying the entire network grows more quickly (because the IDS model is more complex leading to a larger increase in problem size). In this case, verifying correctness for a network with 50 peering points, when verification is performed on the whole entire network, takes approximately 10 minutes. Hence, being able to verify slices and use symmetry is crucial when verifying such networks.

6 Related Work

Next, we discuss related work in network verification and formal methods.

Testing Networks with Middleboxes The work most closely related to us is Buzz [13], which uses symbolic execution to generate sequences of packets that can be used to test whether a network enforces an invariant. Testing, as provided by Buzz, is complimentary to verification. Our verification process does not require sending traffic through the network, and hence provides a non-disruptive mechanism for ensuring that changes to a network (*i.e.,* changing middlebox or routing configuration, adding new types of middleboxes, etc.) do not result in invariant violation. Verification is also useful when initially designing a network, since designs can be evaluated to ensure they uphold desirable invariants. However, as we have noted in §3.6, our verification results hold if and only if middlebox implementations are correct, *i.e.,* packets are correctly classified, etc. Combining a verification tool like VMN with a testing tool such as Buzz allows us to circumvent this problem, when possible (*i.e.,* when the network is not heavily utilized, or when adding new types of middleboxes), Buzz can be used to test if invariants hold. This is similar to the complimentary use of verification and testing in software development today. More specifically, it is

[10] This setup is preferred to installing a separate scrubbing box at each peering point because of the high cost of these boxes, which can amount to several million dollars for a warranteed period of 3 years.

[11] While IDSes in general might not be flow-parallel, the specific IDS used here is flow-parallel with respect to a slice.

almost identical to the relationship between ATPG (testing) and HSA (verification).

Beyond the difference of purpose, there are some other crucial difference between Buzz and VMN: (a) Buzz's middlebox models are context specific, and must be specialized for each network, while VMN's models are more general and designed to be reused across networks; (b) Buzz's testing does not consider the effect of middlebox failure, while our approach can be used to verify that invariants hold despite network failures; and (c) scaling to large networks with slicing is one of our major contributions. Buzz can scale to networks with 100s of nodes before running into scaling limits, while in many cases we can scale to arbitrary sized networks. We believe slicing can also be used by Buzz to improve scaling.

SymNet [40] has also suggested the need to extend these mechanisms to handle mutable datapath elements. SymNet uses symbolic execution to check reachability properties, however their technique is only applicable when state is not shared across a flow and was only applied to cases where the middlebox can punch holes, but do no more. They can therefore only deal with a few kinds of middleboxes, and it is unclear if their technique can be extended to scalably handle other middlebox types.

Verifying Forwarding Rules Recent efforts in network verification [5, 7, 15, 20, 21, 26, 36, 38] have focused on verifying the network dataplane by analyzing forwarding tables. Some of these tools including HSA [19], Libra [46] and VeriFlow [21] have also developed algorithms to perform near real-time verification of simple properties such as loop-freedom and the absence of blackholes. Recent work [33] has also shown how techniques similar to slicing can be used to scale these techniques. Our approach to slicing generalizes this work by accounting for state. While these techniques are 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 [18, 42]. Recent efforts [25] 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.

Verifying Network Applications Other work has looked at verifying the correctness of control and data plane applications. NICE [7] 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 [35] at the cost of completeness. More recently, Vericon [6] has looked at sound verification of a restricted class of controllers.

Recent work [11] 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 [9]. 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 [7], HSA [20] and others [39] rely on this technique. However these techniques scale exponentially with the number of states and for even moderately large problems one must choose 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 use of simple logic allows verification to terminate in a decidable manner for practical networks.

Language Abstractions Several recent works in software-defined networking [14, 16, 22, 30, 44] 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.

7 Conclusion

We started this work aiming to extend the benefits of verification to networks with middleboxes. In building VMN, we had three significant realizations: first, we should separate the classification and forwarding behavior of middleboxes, and only abstractly model classification. Second, scaling verification requires us to take advantage of how state is partitioned by middleboxes, and of the symmetry in the network topology and policies. Lastly, verifying invariants in the presence of failures is essential to making middlebox verification useful in existing networks. The combination of the first two steps enables verification of reachability invariants on extremely large networks; while the last allows us to verify most previously reported configuration bugs. Since our results depend on network symmetry they do not entirely apply to random networks, and scalability for general networks remains an important open problem. However, we note that in other domains where verification has been successfully employed, scaling has been achieved by taking advantage of the problem structure, and we believe our work on slices is analogous to this work.

8 References

[1] Amazon Glacier. Retrieved 01/23/2016 `https://aws.amazon.com/glacier/`.

[2] Amazon S3. Retrieved 01/23/2016 `https://aws.amazon.com/s3/`.

[3] Azure Storage. Retrieved 01/23/2016 https://azure.microsoft.com/en-us/services/storage/.

[4] AMAZON. Amazon EC2 Security Groups. http://goo.gl/GfYQmJ, Oct. 2014.

[5] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic foundations for networks. In *POPL* (2014).

[6] BALL, T., BJØRNER, N., GEMBER, A., ITZHAKY, S., KARBYSHEV, A., SAGIV, M., SCHAPIRA, M., AND VALADARSKY, A. VeriCon: Towards Verifying Controller Programs in Software-Defined Networks. In *PLDI* (2014).

[7] CANINI, M., VENZANO, D., PERES, P., KOSTIC, D., AND REXFORD, J. A NICE Way to Test OpenFlow Applications. In *NSDI* (2012).

[8] CARPENTER, B., AND BRIM, S. RFC 3234: Middleboxes: Taxonomy and Issues, Feb. 2002.

[9] CLARKE, E. M., GRUMBERG, O., AND PELED, D. *Model checking*. MIT Press, 2001.

[10] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[11] DOBRESCU, M., AND ARGYRAKI, K. Software Dataplane Verification. In *NSDI* (2014).

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

[13] FAYAZ, S. K., YU, T., TOBIOKA, Y., CHAKI, S., AND SEKAR, V. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *NSDI* (2016).

[14] FOSTER, N., GUHA, A., REITBLATT, M., STORY, A., FREEDMAN, M. J., KATTA, N. P., MONSANTO, C., REICH, J., REXFORD, J., SCHLESINGER, C., WALKER, D., AND HARRISON, R. Languages for software-defined networks. *IEEE Communications Magazine 51*, 2 (2013), 128–134.

[15] FOSTER, N., KOZEN, D., MILANO, M., SILVA, A., AND THOMPSON, L. A Coalgebraic Decision Procedure for NetKAT. In *POPL* (2015).

[16] GUHA, A., REITBLATT, M., AND FOSTER, N. Machine-verified network controllers. In *PLDI* (2013), pp. 483–494.

[17] JONES, C. B. Specification and design of (parallel) programs. In *IFIP Congress* (1983), pp. 321–332.

[18] KATTA, N. P., REXFORD, J., AND WALKER, D. Incremental consistent updates. In *NSDI* (2013).

[19] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013).

[20] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012).

[21] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. Veriflow: Verifying network-wide invariants in real time. In *NSDI* (2013).

[22] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., GUDE, N., INGRAM, P., JACKSON, E., LAMBETH, A., LENGLET, R., LI, S.-H., PADMANABHAN, A., PETTIT, J., PFAFF, B., RAMANATHAN, R., SHENKER, S., SHIEH, A., STRIBLING, J., THAKKAR, P., WENDLANDT, D., YIP, A., AND ZHANG, R. Network virtualization in multi-tenant datacenters. In *NSDI* (2014).

[23] KUZNIAR, M., PERESINI, P., CANINI, M., VENZANO, D., AND KOSTIC, D. A SOFT Way for OpenFlow Switch Interoperability Testing. In *CoNEXT* (2012).

[24] LICHTENSTEIN, O., PNUELI, A., AND ZUCK, L. D. The glory of the past. In *Logics of Programs, Conference, Brooklyn College, June 17-19, 1985, Proceedings* (1985), pp. 196–218.

[25] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Dna pairing: Using differential network analysis to find reachability bugs. Tech. rep., Microsoft Research, 2014. research.microsoft.com/pubs/215431/paper.pdf.

[26] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P., AND KING, S. T. Debugging the data plane with Anteater. In *SIGCOMM* (2011).

[27] MISRA, J., AND CHANDY, K. M. Proofs of networks of processes. *IEEE Trans. Software Eng. 7*, 4 (1981), 417–426.

[28] MITRE. Common Vulnerabilities and Exposures. https://cve.mitre.org/.

[29] MSR. Z3. http://z3.codeplex.com/, 2015.

[30] NELSON, T., FERGUSON, A. D., SCHEER, M. J. G., AND KRISHNAMURTHI, S. A balance of power: Expressive, analyzable controller programming. In *NSDI* (2014).

[31] OPENSTACK. Congress. https://wiki.openstack.org/wiki/Congress retrieved 01/08/2015.

[32] PASQUIER, T., AND POWLES, J. Expressing and enforcing location requirements in the cloud using information flow control. In *International Workshop on Legal and Technical Issues in Cloud Computing (CLaw). IEEE* (2015).

[33] PLOTKIN, G. D., BJØRNER, N., LOPES, N. P., RYBALCHENKO, A., AND VARGHESE, G. Scaling network verification using symmetry and surgery. In *POPL* (2016).

[34] POTHARAJU, R., AND JAIN, N. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In *IMC* (2013).

[35] SEN, K., AND AGHA, G. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV* (2006).

[36] SETHI, D., NARAYANA, S., AND MALIK, S. Abstractions for Model Checking SDN Controllers. In *FMCAD* (2013).

[37] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making middleboxes someone else's problem: Network processing as a cloud service. In *SIGCOMM* (2012).

[38] SKOWYRA, R., LAPETS, A., BESTAVROS, A., AND KFOURY, A. A verification platform for sdn-enabled applications. In *HiCoNS* (2013).

[39] SOSNOVICH, A., GRUMBERG, O., AND NAKIBLY, G. Finding security vulnerabilities in a network protocol using parameterized systems. In *CAV* (2013).

[40] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Symnet: static checking for stateful networks. In *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization* (2013), ACM, pp. 31–36.

[41] SWITCH. SWITCHlan Backbone. `http://goo.gl/iWm9VE`.

[42] VANBEVER, L., REICH, J., BENSON, T., FOSTER, N., AND REXFORD, J. HotSwap: Correct and Efficient Controller Upgrades for Software-Dfined Networks. In *HOTSDN* (2013).

[43] VELNER, Y., ALPERNAS, K., PANDA, A., RABINOVICH, A., SAGIV, M., SHENKER, S., AND SHOHAM, S. Some complexity results for stateful network verification. In *TACAS* (2016).

[44] VOELLMY, A., WANG, J., YANG, Y. R., FORD, B., AND HUDAK, P. Maple: simplifying sdn programming using algorithmic policies. In *SIGCOMM* (2013).

[45] ZENG, H., KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Automatic Test Packet Generation. In *CoNext* (2012).

[46] ZENG, H., ZHANG, S., YE, F., JEYAKUMAR, V., JU, M., LIU, J., MCKEOWN, N., AND VAHDAT, A. Libra: Divide and conquer to verify forwarding tables in huge networks. In *NSDI* (2014).