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 [26–28, 35]. 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, both for the original network and in the presence of failures. The main challenge lies in handling large and complicated networks. We achieve scaling 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

Network operators have long relied on best-guess configurations and a "we'll fix it when it breaks" approach. However, as networking matures as a field, and institutions increasingly expect networks to provide reachability, isolation, and other behavioral invariants, there is growing interest in developing rigorous verification tools that can check whether these invariants are enforced by the network configuration.

The first generation of such tools [26–28, 35] check reachability and isolation invariants in near-real time, but assume that network devices have "static datapaths, " *i.e.,* their forwarding behavior is set by the control plane and not altered by observed traffic. This assumption is entirely sufficient for networks of routers but not for networks that contain middleboxes with "mutable datapaths" whose forwarding behavior may depend on the entire packet history they have seen. Examples of such middleboxes include firewalls that allow end hosts to establish flows to the outside world through "hole punching" and network optimizers that cache popular content. Middleboxes are prevalent – in fact, they constitute as much as a third of network devices in enterprise networks [49] – and expected to become more so with the rise of Network Function Virtualization (NFV) because the latter makes it easy to deploy additional middleboxes without changes in the physical infrastructure [13]. Given their complexity and prevalence, middleboxes are the cause of many network failures; for instance, 43% of a network provider's failure incidents involved middleboxes, and between 4% and 15% of these incidents were the result of middlebox misconfiguration [41].

Our goal is to reduce such misconfigurations by extending verification to large networks that contain middleboxes with mutable datapaths. In building our system for verifying reachability and isolation properties in mutable networks – which we call VMN (for *v*erifying *m*utable *n*etworks) – we do not take the direct approach of attempting to verify middlebox code itself, and then extend this verification to the network as a whole, for two reasons. First, such an approach does not scale to large networks. The state-of-the-art in verification is still far from able to automatically analyze the source code or executable of most middleboxes, let alone the hundreds of interconnected devices that it interacts with [51]. Thus, verifying middlebox code directly is *practically infeasible*.

Second, middlebox code does not always work with easily verified abstractions. For example, some IDSes attempt to identify suspicious traffic. No method can possibly verify whether their code is successful in identifying all suspicious traffic because there is no accepted definition of what constitutes suspicious. Thus, verifying such middlebox code is *conceptually impossible*.

Faced with these difficulties, we return to the problem operators want to solve. They recognize that there may be imprecision in identifying suspicious traffic, but they want to ensure that all traffic that the middlebox identifies as being suspicious is handled appropriately (*e.g.,* by being routed to a scrubber for further analysis). The first problem – perfectly identifying suspicious traffic – is not only ill-defined, it is not controlled by the operator (in the sense that any errors in identification are beyond the reach of the operator's control). The second problem – properly handling traffic considered suspicious by a middlebox – is precisely what an operator's configuration, or misconfiguration, can impact.

The question, then is how to abstract away unnecessary complexity so that we can provide useful answers to operators. We do so by leveraging two insights. First,

middlebox functionality can be logically divided into two parts: forwarding (*e.g.,* forward suspicious and non-suspicious packets through different output ports) and packet classification (*e.g.,* whether a packet is suspicious or not). Verifying this kind of amorphous packet classification it not our concern. Second, there exist a large number of different middleboxes, but most of them belong to a relatively small set of middlebox types – firewalls, IDS/IPSes, and network optimizers (the latter police traffic, eliminate traffic redundancy, and cache popular content) [6]; middleboxes of the same type define similar packet classes (*e.g.,* "suspicious traffic") and use similar forwarding algorithms, but may differ dramatically in how they implement packet classification.

Hence, we model a middlebox as: a *forwarding model*, a set of *abstract packet classes*, and a set of *oracles* that automatically determine whether a packet belongs to an abstract class – so, the oracle abstracts away the implementation of packet classification. With this approach, we do not need a new model for every middlebox, only one per middlebox type.

This modeling approach avoids the conceptual difficulties, but does not address the practical one of scaling to large networks. One might argue that, once we abstract away packet classification, what remains of middlebox functionality is simple forwarding logic (how to forward each packet class), hence it should be straightforward to extend prior tools to handle middleboxes. However, while checking reachability property in static networks is PSPACE-complete [2], it is EXPSPACE-complete when mutable datapaths are considered [56]. Mutable verification is thus algorithmically more complicated. Furthermore, recent work has shown that even verifying such properties in large static networks requires the use of "reduction techniques", which allow invariants to be verified while reasoning about a small part of the network [40]. Applying such techniques to mutable datapaths is more complex, because parts of the network may *effect* each other through state, without explicitly exchanging traffic – making it hard to partition the network.

To address this, we exploit the fact that, even in networks with mutable datapaths, observed traffic often affects only a well-defined subset of future traffic, *e.g.,* packets from the same TCP connection or between the same source and destination. We formalize this behavior in the form of two middlebox properties: *flow-parallelism* and *origin-independence*; when combined with structural and policy symmetry, as is often the case in modern networks [40], these properties enable us to use reduction effectively and verify invariants in arbitrarily large networks in a few seconds (§5).

The price we pay for model simplicity and scalability is that we cannot use our work to check middlebox implementations and catch interesting middlebox-specific bugs [10]; however, we argue that it makes sense to develop separate tools for that purpose, and not unnecessarily complicate verification of reachability and isolation.

Listing 1: Model for an example firewall

```
1  class Firewall (acls:
       ↪ Set[(Address, Address)]) {
2    abstract malicious(p: Packet): bool
3    val tainted: Set[Address]
4    def model (p: Packet) = {
5      tainted.contains(p.src)
           ↪ => forward(Empty)
6      acls.contains((p.src,
           ↪ p.dst)) => forward(Empty)
7      malicious(p)
           ↪ => tainted.add(p.src);
           ↪ forward(Empty)
8      _ => forward(Seq(p))
9    }
10 }
```

2 Modeling Middleboxes

We begin by looking at how middleboxes are modeled in VMN. First, we provide a brief overview of how these models are expressed (§2.1), then we present the rationale behind our choices (§2.2), and finally we discuss discuss real-world examples (§2.3).

2.1 Middlebox Models

We illustrate our middlebox models through the example in Listing 1, which shows the model for a simplified firewall. The particular syntax is not important to our technique; we use a Scala-like language, because we found the syntax to be intuitive for our purpose, and in order to leverage the available libraries for parsing Scala code. We limit our modeling language to not support looping (*e.g.,* `for`, `while`, etc.) and only support branching through partial functions (Lines 5–7).

A VMN model is a class that implements a `model` method (Line 4). It may define *oracles* (*e.g.,* `malicious` on Line 3), which are abstract functions with specified input (`Packet` in this case) and output (`boolean` in this case) type. It may also define data structures—sets, lists, and maps—to store state (*e.g.,* `tainted` on Line 3) or to accept input that specifies configuration (*e.g.,* `acls` on Line 1). Finally, it may access predefined packet-header fields (*e.g.,* `p.src` and `p.dst` on Lines 6 and 7). We limit function calls to oracles and a few built-in functions for extracting information from packet-header fields (our example model does not use the latter).

The `model` method specifies the middlebox's forwarding behavior. It consists of a set of variable declarations, followed by a set of guarded forwarding rules (Lines 5–8). Each rule must be terminated by calling the `forward`

function with a set of packets to forward (which could be the empty set). In our example, the first three rules (Line 5–7) drop packets by calling `forward` with an empty set, while the last one (Line 8) forwards the received packet `p`.

Putting it all together, the model in Listing 1 captures the following middlebox behavior: On receiving a new packet, first check if the packet's source has previously contributed malicious packets, in which case drop the packet (Line 5). Otherwise, check if the packet is prohibited by the provided access control list (ACL), in which case drop the packet (Line 6). Otherwise, checks if the packet is malicious, in which case record the packet's source and drop the packet (Line 7). Finally, if none of these conditions are triggered, forwards the packet (Line 8).

The model in Listing 1 does *not* capture how the firewall determines whether a packet is malicious or not; that part of its functionality is abstracted away through the `malicious` oracle. We determine what classification choices are made by an oracle (*e.g.,* `malicious`) and what are made as a part of the forwarding model (*e.g.,* our handling of ACLs) based on whether the packet can be fully classified by just comparing header fields to known values (these values might have been set as a part of processing previous packets) – as is the case with checking ACLs and whether a flow is tainted – or does it require more complex logic (*e.g.,* checking the content, etc.) – as is required to mark a packet as malicious.

2.2 Rationale and Implications

Why did we choose to model middleboxes as a *forwarding model* which can call a set of *oracles*?

First, we wanted to express middlebox behavior in the same terms as those used by network operators to express reachability and isolation invariants. Network operators typically refer to traffic they wish to allow or deny in two different ways: in terms of packet-header fields that have semantic meaning in their network (*e.g.,* a packet's source IP address indicates the particular end host or user that generated that packet), or in terms of semantic labels attached to packets or flows by middleboxes (*e.g.,* "contains exploits," "benign," or "malicious"). This is why a VMN model operates based on two kinds of information for each incoming packet: predefined header fields and abstract packet classes defined by the model itself, which represent semantic labels.

Second, like any modeling work, we wanted to strike a balance between capturing relevant behavior and abstracting away complexity. Two elements guided us: first, the middlebox configuration that determines which semantic labels are attached to each packet is typically not written by network operators: it is either embedded in middlebox code, or provided by third parties, *e.g.,* Emerging Threats rule-set [12] or vendor provided virus definitions [52]. Second, the middlebox code that uses such rulesets and/or virus definitions is typically sophisticated and performance-optimized, *e.g.,* IDSes and IPSes typically extract the packet payload, reconstruct the byte stream, and then use regular expression engines to perform pattern matches. So, the part of middlebox functionality that maps bit patterns to semantic labels (*e.g.,* determines whether a packet sequence is "malicious") is hard to analyze, yet unlikely to be of interest to an operator who wants to check whether they configured their network as intended. This is why we chose to abstract away this part with the oracles – and model each middlebox only in terms of how it treats a packet based on the packet's headers and abstract classes.

Mapping low-level packet-processing code to semantic labels is a challenge that is common to network-verification tools that handle middleboxes. We address it by explicitly abstracting such code away behind the oracles. Buzz [14] provides ways to automatically derive models from middlebox code, yet expects the code to be written in terms of meaningful semantics like addresses. In practice, this means that performance-optimized middleboxes (*e.g.,* ones that build on DPDK [22] and rely on low level bit fiddling) need to be hand-modeled for use with BUZZ. Similarly, SymNet [51] claims to closely matches executable middlebox code. However, SymNet also requires that code be written in terms of access to semantic fields; in addition, it allows only limited use of state and limits loops. In reality, therefore, neither Buzz nor SymNet can model the behavior of general middleboxes (*e.g.,* IDSes and IPSes). We recognize that modeling complex classification behavior, *e.g.,* from IDSes and IPSes, either by expressing these in a modeling language or deriving them from code is impractical, and of limited practical use when verifying reachability and isolation. Therefore rather than ignoring IDSes and IPSes (as done explicitly by SymNet, and implicitly by Buzz), we use oracles to abstract away classification behavior.

How many different models? We need one model per middlebox type, *i.e.,* one model for all middleboxes that define the same abstract packet classes and use the same forwarding algorithm. A 2012 study showed that, in enterprise networks, most middleboxes belong to a small number of types: firewalls, IDS/IPS, and network optimizers [49]. As long as this trend continues, we will also need a small number of models.

Who will write the models? Because we need only a few models, and they are relatively simple, they can come from many sources. Operators might provide them as part of their requests for bids, developers of network-configuration checkers (*e.g.,* Veriflow Inc. [57] and Forward Network [15]) might develop them as part of their offering, and middlebox manufactures might provide them to enable reliable configuration of networks which deploy their products. The key point is that one can write a VMN model without access to the corresponding middlebox's source code or executable; all one needs is

the middlebox's manual—or any document that describes the high-level classification performed by the middlebox defines, whether and how it modifies the packets of a given class, and whether it drops or forwards them.

What happens to the models as middleboxes evolve? There are two ways in which middleboxes evolve: First, packet-classification algorithms change, *e.g.,* what constitutes "malicious" traffic evolves over time. This does not require any update to VMN models, as they abstract away packet-classification algorithms. Second semantic labels might change (albeit more slowly), *e.g.,* an operator may label traffic sent by a new applications. A new semantic label requires updating a model with a new oracle and a new guided forwarding rule.

Limitations: Our approach cannot help find bugs in middlebox code—*e.g.,* in regular expression matching or flow lookup—that would cause a middlebox to forward packets it should be dropping or vice versa. In our opinion, it does not make sense to incorporate such functionality in our tool: such debugging is tremendously simplified with access to middlebox code, while it does not require access to the network topology where the middlebox will be placed. Hence, we think it should be targeted by separate debugging tools, to be used by middlebox developers, not network operators trying to figure out whether they configured their network as they intended.

2.3 Real-world Examples

Next, we present some examples of how existing middleboxes can be modeled in VMN. For brevity, we show models for firewalls and intrusion prevention systems in this section. We include other examples including NATs (from `iptables` and pfSense), gateways, load-balancers (HAProxy and Maglev [11]) and caches (Squid, Apache Web Proxy) in Appendix A. We also use Varnish [20], a protocol accelerator to demonstrate VMN's limitations.

Firewalls We examined two popular open-source firewalls, `iptables` [42] and pfSense [38], written by different developers and for different operating systems (`iptables` targets Linux, while pfSense requires FreeBSD). These tools also provide NAT functions, but for the moment we concentrate on their firewall functions.

For both firewalls, the configuration consists of a list of match-action rules. Each action dictates whether a matched packet should be forwarded or dropped. The firewall attempts to match these rules in order, and packets are processed according to the first rule they match. Matching is done based on the following criteria:

- Source and destination IP prefixes.
- Source and destination port ranges.
- The network interface on which the packet is received and will be sent.
- Whether the packet belongs to an established connection, or is "related" to an established connection.

The two firewalls differ in how they identify related connections: `iptables` relies on independent, protocol-specific helper modules [32]. pfSense relies on a variety of mechanisms: related FTP connections are tracked through a helper module, related SIP connections are expected to use specific ports, while other protocols are expected to use a pfSense proxy and connections from the same proxy are treated as being related.

Listing 2 shows a VMN model that captures the forwarding behavior of both firewalls—and, to the best of our knowledge, any shipping IP firewall. The configuration input is a list of rules (Line 12). The `related` oracle abstracts away the mechanism for tracking related connections (Line 13). The `established` set tracks established connections (Line 14). The forwarding model searches for the first rule that matches each incoming packet (Lines 17–26); if one is found and it allows the packet, then the packet is forwarded (Line 27), otherwise it is dropped (Line 28).

Listing 2: Model for `iptables` and pfSense

```
1  case class Rule (
2    src: Option[(Address, Address)],
3    dst: Option[(Address, Address)],
4    src_port: Option[(Int, Int)],
5    dst_port: Option[(Int, Int)],
6    in_iface: Option[Int],
7    out_iface: Option[Int],
8    conn: Option[Bool],
9    related: Option[Bool],
10   accept: Bool
11 )
12 class Firewall (acls: List[Rule]) {
13   abstract related (p: Packet): bool
14   val established: Set[Flow]
15   def model (p: Packet) = {
16     val f = flow(p);
17     val match = acls.findFirst(
18       acl => (acl.src.isEmpty ||
19               acl.src._1 <=
                    ↪ p.src && p.src
                    ↪ < acl.src._2) &&
20              ...
21              (acl.conn.isEmpty ||
22               acl.conn ==
                    ↪ established(f))&&
23              (acl.related.isEmpty ||
24               acl.related
                    ↪ == related(f)));
25     match.isDefined && match.accept
         ↪ => forward(Seq(p))
         ↪ // We found
         ↪ a match which said the
         ↪ packet should be forwarded
26     _ => forward(Empty)
         ↪ // Drop all other packets
27   }
28 }
```

Intrusion Prevention Systems We considered two kinds: general systems like Snort [45], which detect a variety of attacks by performing signature matching on packet payloads, and web application firewalls like ModSecurity [44] and IronBee [43], which detect attacks on web applications by analyzing HTTP requests and bodies sent to a web server. Both kinds accept as configuration "rulesets" that specify suspicious payload strings and other conditions, *e.g.,* TCP flags, connection status, etc., that indicate malicious traffic. Network operators typically rely on community maintained rulesets, *e.g.,* Emerging Threats [12] (with approximately 20,000 rules) and Snort rulesets (with approximately 3500 rules) for Snort; and OWASP [24] for ModSecurity and IronBee.

Listing 3: Model for IPS

```
1 class IPS {
2   abstract malicious(p: Packet): bool
3   val infected: Set[Flow]
4   def model (p: Packet) = {
5     infected.contains(flow(p))
          ↪ => forward(Empty)
6     malicious(p)
          ↪ => infected.add(flow(p);
          ↪ forward(Empty)
7     _ => forward(Seq(p))
8   }
9 }
```

Listing 3 shows a VMN model that captures the forwarding behavior of these systems. The `malicious` oracle abstracts away how malicious packets are identified (Line 2). The `infected` set keeps track of connections that have contributed malicious packets (Line 3). The forwarding model simply drops a packet if the connection is marked as infected (Line 5) or is malicious according to the oracle (Line 6). It may seem counter-intuitive that this model is simpler than that of a firewall (Listing 2), because we tend to think of IDS/IPSes as more complex devices; however, a firewall has more sophisticated forwarding behavior, which is what we model, whereas the complexity of an IDS/IPS lies in packet classification, which is what we abstract away.

Programmable web accelerators The Varnish cache [20] demonstrates the limits of our approach. Varnish allows the network operator to specify, in detail, the handling of each HTTP(S) request, *e.g.,* building dynamic pages that reference static cached content, or even generating simple dynamic pages. Varnish's configuration therefore acts more like a full-fledged program – and it is hard to separate out "configuration" from forwarding implementation. We can model Varnish by either developing a model for each configuration or abstracting away the entire configuration. The former impedes reuse, while the later is imprecise and neither is suitable for verification.

3 Modeling Networks

Having described VMN's middlebox models, we turn to how VMN models an entire network of middleboxes and how we scale verification to large networks. Here we build on existing work on static network verification [27,28]. For scalability, which is one of our key contributions, we identify small network subsets—*slices*—where we can check invariants efficiently. For some common middleboxes we can find slices whose size is independent of network size.

3.1 Network Models

Veriflow [28] and header-space analysis [27] (HSA) summarize network behavior as a *network transfer function*. This builds on the concept of a *located packet*, *i.e.,* a packet augmented with the input port on which it is received. A network transfer function takes as input a located packet and produces a set of located packets. The transfer function ensures that output packets are located at feasible ports, *i.e.,* at ports physically connected to the input location.

VMN models a network as a collection of end hosts and middleboxes connected by a network transfer function. More formally, we define a network $N = (V, E, P)$ to be a set of nodes V, a set of links (edges) E connecting the nodes, and a possibly infinite set of packets P. Nodes include both end hosts and middleboxes. For notational convenience, each packet $p \in P$ includes its network location (port), which is given by $p.loc$. For such a network N, we define the network transfer function N_T as

$$N_T : p \rightarrow P' \subseteq P,$$

where $p \in P$ is a packet and $P' \subseteq P$ is a set of packets.

Given a network, we treat all end hosts and middleboxes as endpoints from which packets can be sent and at which they can be received; we then use Veriflow to generate a transfer function for this network, and we turn the resulting output into a form that can be accepted by an SMT solver. This essentially transforms the network into one where all packets sent from end hosts traverse a sequence of middleboxes before being delivered to their final destination. Our verification then just focuses on checking whether the sequence of middleboxes encountered by a packet correctly enforces any desired reachability invariant. Note that, if a reachability invariant is enforced merely by static-datapath—e.g., router—configuration, we do successfully detect that, i.e., VMN's verification is a generalization of static network verification.

Veriflow and HSA cannot verify networks with forwarding loops, and we inherit this limitation; we check to ensure that the network does not have any forwarding loop and raise an error whenever one is found.

3.2 Scaling Verification: Slicing

While network transfer functions reduce the number of distinct network elements that need to be considered

in verification, this reduction is often insufficient. For example, Microsoft's Chicago Datacenter [8] contains over 224,000 servers running virtual machines connected over virtual networks. In such an environment, each server typically acts as a middlebox, resulting in a network with several 100,000 middleboxes, *e.g.,* firewalls, load balancers, SSL proxies, etc.. We want to be able to check invariants in such large networks within a few minutes, however, typically verifying such large instances is infeasible or takes several days.

Our approach to achieving this goal is to identify subnetworks which can be used to efficiently verify invariants. We provide a brief overview of our techniques in this section, and deal a more complete treatment to Appendix C.

First, we formally define a *subnetwork*: Given a network $N = (V, E, P)$ with network transfer function N_T, a subnetwork Ω of N is a subgraph of N consisting of: a subset of the nodes in V; all links in E that connect this subset of nodes; and all packets in P whose location, source and destination *are in* Ω. We say that a packet's source (destination) *is in* Ω, if and only if a node in Ω has the right to use the packet' source (destination) address.[1] We compute a restricted transfer function $N_T|_\Omega$ for subnetwork Ω by modifying N_T such that its domain and range are restricted to packets in Ω. We say that subnetwork Ω is *closed under forwarding*, if, for any packet p in Ω, $N_T|_\Omega(p) = N_T(p)$, *i.e.,* the packet is not forwarded out of Ω.

A *slice* is a special kind of subnetwork. We treat the network as a state machine whose state is defined by the set of packet that have been delivered, the set of packets pending delivery and the state of all middleboxes (see Appendix C for details). State transitions in this model represent the creation of a new packet at an endhost or the delivery and processing of a pending packet at a node. We say that a state S is *reachable in the network*, if and only if there is a valid sequence of transitions starting from an initial state[2] that results in the network being in state S. A subnetwork Ω is *closed under state*, if and only if (a) it is closed under forwarding and (b) every state that is reachable in the entire network has an equivalent reachable state in Ω (*i.e.,* a surjection exists between the network state and subnetwork's state). A slice is a subnetwork that is closed under state.

In our formalism, an invariant I is a predicate on the state of a network, and an invariant is violated if and only if the predicate does not hold for a reachable state. We say an invariant is *evaluable on a subnetwork* Ω, if the corresponding predicate refers only to packets and middlebox state contained within Ω. As we show in Appendix C, any invariant evaluable on some slice Ω of network N, holds in Ω if and only if it also holds in N.

[1] We assume that we are provided with a mapping from each node to the set of addresses that it can use.

[2] The initial state represents a network where no packets have been created or delivered.

The remaining challenge is to identify such a slice given a network N and an invariant I, and we do so by taking advantage of how middleboxes update and use state. We identify two types of middleboxes: *flow-parallel* middlebox, whose state is partitioned such that two distinct flows cannot affect each other; and *origin-agnostic* middleboxes whose behavior is not affected by the origin (*i.e.,* sequence of transitions) of its current state. If all middleboxes in the network have one of these properties, then invariants can be verified on slices whose size is independent of the size of the network.

3.2.1 Flow-Parallel Middleboxes

Several common middleboxes partition their state by flows (*e.g.,* TCP connections), such that the handling of a packet is dictated entirely by its flow and is independent of any other flows in the network. Examples of such middleboxes include firewalls, NATs, IPSes, and load balancers. For analysis, such middleboxes can be decomposed into a set of middleboxes, each of which processes exactly one flow without affect network behavior. From this observation we deduce that any subnetwork that is closed under forwarding and contains only flow-parallel middleboxes is also closed under state, and is therefore a slice.

Therefore, , if a network N contains only flow-parallel middleboxes, then we can find a slice on which invariant I is evaluable by picking the smallest subnetwork Ω on which I is evaluable (which always exists) and adding the minimal set of nodes from network N required to ensure that it is closed under forwarding. This minimal set of nodes is precisely the set of all middleboxes that appear on any path connecting hosts in Ω. Since path lengths in a network are typically independent of the size of the network, the size of a slice is generally independent of the size of the network. We present an example using slices comprised of flow-parallel middleboxes, and evaluate its efficiency in §5.1.

3.2.2 Origin Agnostic Middleboxes

Even when middleboxes, *e.g.,* caches, must share state across flows, their behavior is often dependent only on the state of a middlebox not on the sequence of transitions leading up to that state, we call such middleboxes *origin-agnostic*. Examples of origin-agnostic middleboxes include caches—whose behavior depends only on whether some content is cached or not; IDSes, etc. We also observe that network policy commonly partitions end hosts into *policy equivalence classes*, *i.e.,* into set of end hosts to which the same policy applies and whose packets are treated equivalently. In this case, any subnetwork that is closed under forwarding, and contains only origin-agnostic (or flow-parallel) middleboxes, and has an end host from each policy equivalence class in the network is also closed under state, hence, it is a slice.

Therefore, given a network N containing only flow-

parallel and origin-agnostic middleboxes and an invariant I, we can find a slice on which I is evaluable by using a procedure similar to the one for flow-parallel middleboxes. This time round, the slice must contain an end host from each policy equivalence class, and hence the size of the slice depends on the number of such classes in the network. Networks with complex policy are harder to administer, and generally scaling a network does not necessitate adding more policy classes. Therefore, the size of slices in this case is also independent of the size of the network. We present an example using slices comprised of origin-agnostic middleboxes, and evaluate its efficiency in §5.2.

3.3 Scaling Verification: Symmetry

Slicing allows us to verify an individual invariant in an arbitrarily large network. However, the correctness of an entire network depends on several invariants holding simultaneously, and the number of invariants needed to prove that an entire network is correct grows with the size of the network. Therefore, to scale verification to the entire network we must reduce the number of invariants that need to be checked. For this we turn to symmetry; we observe that networks (especially as modeled in VMN) have a large degree of symmetry, with traffic between several end host pairs traversing the same sequence of middlebox types, with identical configurations. Based on this observation, and on our observation about *policy equivalence classes* we can identify invariants which are symmetric; we define two invariants I_1 and I_2 to be symmetric if I_1 holds in the network N if and only if I_2 also holds in the network. Symmetry can be determined by comparing the smallest slices in which each invariant can be evaluated, and seeing if the graphs are isomorphic up to the type and configuration of individual middleboxes. When possible VMN uses symmetry to reduce the number of invariants that need to be verified, enabling correctness to be checked for the entire network.

4 Checking Reachability

VMN accepts as input a set of middlebox models connected through a network transfer function representing a *slice* of the original network, and one or more invariants; it then runs a *decision procedure* to check whether the invariants hold in the given slice or are *violated*. In this section we first describe the set of invariants supported by VMN (§4.1) and then describe the decision procedure used by VMN (§4.2).

4.1 Invariants

VMN focuses on checking *isolation invariants*, which are of the form "do *packets* (or *data*) belonging to some *class* reach one or more *end hosts*, given certain *conditions* hold?" We say that an invariant holds if and only if the answer to this question is no. We verify this fact assuming worse-case behavior from the oracles. We allow invariants to *classify* packet or data using (a) header fields (source address, destination address, ports, etc.); (b) origin, indicating what end host originally generated some content; (c) oracles *e.g.,* based on whether a packet is *infected*, etc.; or (d) any combination of these factors, *i.e.,* we can choose packets that belong to the intersection of several classes (using logical conjunction), or to the union of several classes (using disjunction). Invariants can also specify temporal *conditions* for when they should hold, *e.g.,* invariants in VMN can require that packets (in some class) are blocked until a particular packet is sent. We now look at a few common classes of invariants in VMN.

Simple Isolation Invariants are of the form "do packets belonging to some class reach destination node d?" This is the class of invariants supported by stateless verification tools *e.g.,* Veriflow and HSA. Concrete examples include "Do packets with source address a reach destination b?", "Do packets sent by end host a reach destination b?", "Are packets deemed mallicious by Snort delivered to server s?", etc.

Flow Isolation Invariants extend simple isolation invariants with temporal constraints. This includes invariants like "Can a receive a packet from b without previously opening a connection?"

Data Isolation Invariants make statements about what nodes in the network have access to some data, this is similar to invariants commonly found in information flow control [59] systems. Examples include "Can data originating at server s be accessed by end host b?'

Pipeline Invariants ensure that packets of a certain class have passed through a particular middlebox. Examples include "Do all packets marked as suspicious by a light IDS pass through a scrubber before being delivered to end host b?"

4.2 Decision Procedure

VMN's verification procedure builds on Z3 [9], a modern SMT solver. Z3 accepts as input logical formulae (written in first-order logic with additional "theories" that can be used to express functions, integers and other objects) which are expressed in terms of variables, and returns whether there exists a satisfying assignment of values to variables, *i.e.,* whether their exists an assignment of values to variables that render all of the formulae true. Z3 returns an example of a satisfying assignment when the input formulae are *satisfiable*, and labels them *unsatisfiable* otherwise. Checking the satisfiability of first-order logic formuals is undecidable in general, and even determining whether satisfiablity can be successfully checked for a particular input is undecidable. As a result Z3 relies on timeouts to ensure termination, and reports *unknown* when a timeout is triggered while checking satifiability.

The input to VMN's verification procedure is comprised of a set of invariants and a network slice. The network slice is expressed as a set of middlebox models, a set of middle-

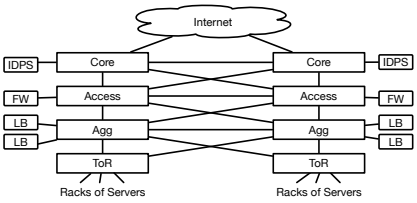


Figure 1: Topology for a datacenter network with middleboxes from [41]. The topology contains firewalls (**FW**), load balancers (**LB**) and intrusion detection and prevention systems (**IDPS**).

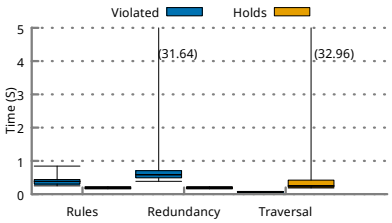


Figure 2: Time taken to verify each network invariant for scenarios in §5.1. We show time for checking both when invariants are violated (Violated) and verified (Holds).

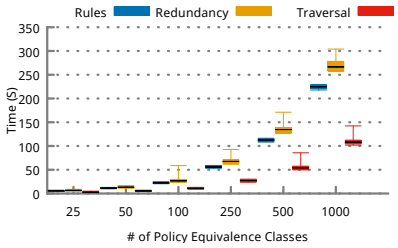


Figure 3: Time taken to verify all network invariants as a function of policy complexity for §5.1. The plot presents minimum, maximum, 5^{th}, 50^{th} and $95th$ percentile time for each.

box instances, a set of end hosts and a network transfer function connecting these nodes. VMN converts this input into a set of first order formulae, which we refer to as the *slice model*. We give details of our logical models in Appendix B, but at a high-level: VMN first adds formulas for each middlebox instance, this formula is based on the provided models; next it compiles the network transfer function into a logical formula; finally it adds formula for each end host.

In VMN, invariants are expressed as logical formula, and we provide convenience functions designed to simplify the task of writing such invariants. As is standard in verification the logical formula representing an invariant is a negation of the invariant itself, and as a result the logical formulation is *satisfiable* if and only if the invariants is *violated*.

VMN checks invariants by invoking Z3 to check whether the conjunction of the slice model and invariants is satisfiable. We report that the invariant holds if and only if Z3 proves this formal to be unsatisfiable, and report the invariant is vilated when a satisfying assignment is found. A convenient feature of such a mechanism is that when a violation is found we can use the satisfying assignment to generate an example where the invariant is violated. This is useful for diagnosing and fixing the configuration error that led to the violation.

Finally, as shown in Appendix D, the formulae generated by VMN lie in a fragment of first order logic called EPR-F, which is an extension of "Effectively Propositional Reasoning" (EPR) [39] a decidable fragment of first-order logic. The logical models produced by VMN are therefore decidable, however when verifying invariants on large network slices Z3 might timeout, and thus VMN may not always be able to determine whether an invariant holds or is violated in a network. In our experience, verification of invariants generally succeeds for slices with up to a few hundred middleboxes.

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 [41] 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. We use load balancers to model the effect of faults (*i.e.,* the load balancer can non-deterministically choose to send traffic to a redundant middlebox). 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 [41], 70% of all reported middlebox misconfiguration are attributed to incorrect rules installed in firewalls. To evaluate this scenario we begin by assigning each end host to one of a few policy groups.[3] We then add firewall rules to prevent end hosts in one group from communicating with end hosts in any other group. We introduce misconfiguration by deleting a random set of these firewall rules. We use VMN to identify for which end hosts the desired invariant

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

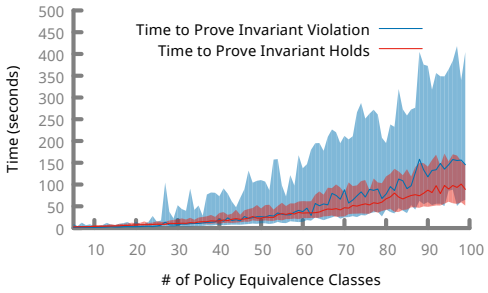


Figure 4: Time taken to verify each data isolation invariant. The shaded region represents the 5^{th}–95^{th} percentile time.

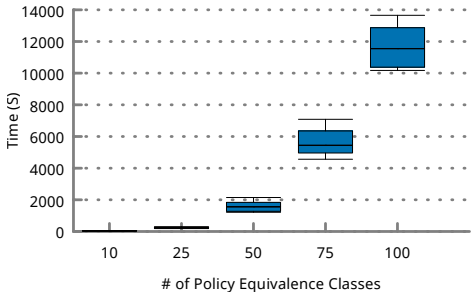


Figure 5: Time taken to verify all data isolation invariants in the network described in §5.2.

holds (*i.e.,* that end hosts can only communicate with other end 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 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; end 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 end 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 [46], AWS Glacier [19], and Azure Blob Store [3]. These storage services must comply with legal and customer requirements [37] 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[4] 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 invari-

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

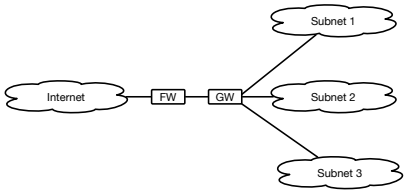


Figure 6: Topology for enterprise network used in §5.3.1, containing a firewall (**FW**) and a gateway (**GW**).

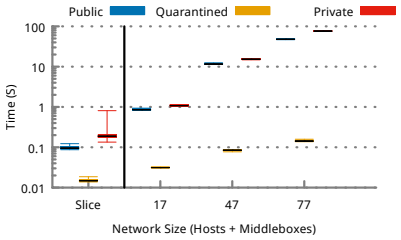


Figure 7: Distribution of verification time for each invariant in an enterprise network (§5.3.1) with network size. The left of the vertical line shows time taken to verify a slice, which is independent of network size, the right shows time taken when slices are not used.

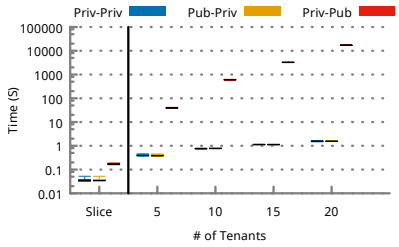


Figure 8: Average verification time for each invariant in a multi-tenant datacenter (§5.3.2) as a function of number of tenants. Each tenant has 10 end hosts. The left of the vertical line shows time taken to verify a slice, which is independent of the number of tenants.

ant 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.

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 end 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.

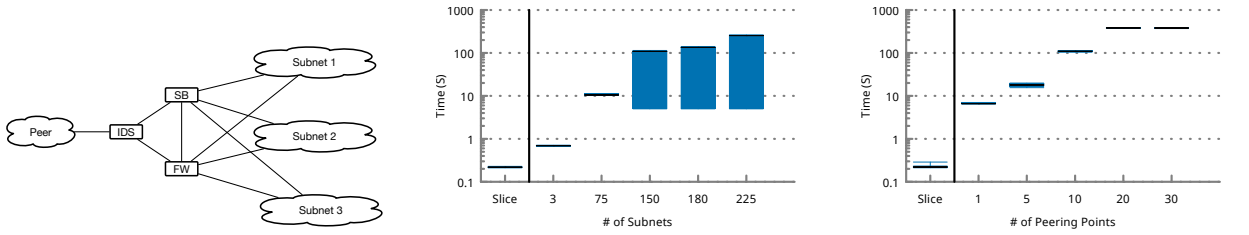
We configure the firewall so as to enforce the target invariants correctly: with two rules denying access (in either direction) for each quarantined subnet, plus one rule denying inbound connections for each private subnet. The results we present below are for the case where all the target invariants 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 [1]. 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



(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.

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.

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 [53], 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.[5]

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 end hosts.

When verifying invariants in a slice we again take advantage of the fact that firewalls and IDSes are flow-parallel. For each subnet, we can verify invariants in a slice containing a peering point, a end 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 [14], 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,

[5] 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.

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 relationship between ATPG (testing) and HSA (verification), and more generally to the complimentary use of verification and testing in software development today.

Beyond the difference of purpose, there are some other crucial difference between Buzz and VMN: (a) in contrast to VMN, Buzz's middlebox models are specialized to a context and cannot be reused across networks, and (b) Buzz does not use techniques such as slicing, limiting its applicability to networks with only several 100 nodes. We believe our slicing techniques can be adopted to Buzz.

Similarly, SymNet [51] proposes the use of symbolic execution for verifying network reachability. They rely on models written in a symbolic execution friendly language SEFL where models are supposed to closely resemble middlebox code. However, to ensure feasibility for symbolic execution, SEFL does not allow unbounded loops, or pointer arithmetic and limits access to semantically meaningful packet fields. These models are therefore very different from the implementation for high performance middleboxes. Furthermore, due to these limitations SEFL cannot model middleboxes like IDSes and IPSes, and is limited to modeling *flow-parallel* middleboxes.

Verifying Forwarding Rules Recent efforts in network verification [2, 5, 17, 27, 28, 35, 48, 50] have focused on verifying the network dataplane by analyzing forwarding tables. Some of these tools including HSA [26], Libra [60] and VeriFlow [28] have also developed algorithms to perform near real-time verification of simple properties such as loop-freedom and the absence of blackholes. Recent work [40] has also shown how techniques similar to slicing can be used to scale these techniques. Our approach generalizes this work by accounting for state and thus extends verification to mutable 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 [25, 55]. Recent efforts [34] have generalized these mechanisms and can determine what parts of configuration are affected by an update, and verify invariants on this subset of the configuration. This work does not consider dynamic, stareful datapath elements with more frequent state updates.

Verifying Network Applications Other work has looked at verifying the correctness of control and data plane applications. NICE [5] proposed using static analysis to verify the correctness of controller programs. Later extensions including [31] have looked at improving the accuracy of NICE using concolic testing [47] by trading away completeness. More recently, Vericon [4] has looked at sound verification of a restricted class of controllers.

Recent work [10] 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, 7, 27]. 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). 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 [16, 21, 29, 36, 58] 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

In this paper we presented VMN, a system for verifying isolation invariants in networks with middleboxes. The key insights behind our work is that abstract middleboxes are well suited to verifying network configuration; and the use of slices which allow invariants to be verified on a subset of the network is essential to scaling.

8 Acknowledgment

We thank Shivaram Venkatraman, Colin Scott, Kay Ousterhout, Amin Tootoonchian, Justine Sherry, Sylvia Ratnasamy, Nate Foster, the anonymous reviewers and our shepherd Boon Thau Loo for the many helpful comments that have greatly improved both the techniques described within this paper and their presentation. This work was supported in part by a grant from Intel Corporation, NSF grant 1420064, and an ERC grant (agreement number 321174-VSSC).

References

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

[2] 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).

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

[4] 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).

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

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

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

[8] DATACENTER WORLD. Largest Data Centers: i/o Data Centers, Microsoft. https://goo.gl/dB9VdO retrieved 09/18/2016, 2014.

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

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

[11] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A fast and reliable software network load balancer. In *NSDI* (2016).

[12] EMERGING THREATS. Emerging Threats open rule set. https://rules.emergingthreats.net/ retrieved 09/18/2016, 2016.

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

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

[15] FORWARD NETWORKS. Forward Networks. https://www.forwardnetworks.com/, 2016.

[16] 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.

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

[18] GADDE, S., CHASE, J., AND RABINOVICH, M. A Taste of Crispy Squid. In *Workshop on Internet Server Performance* (1998).

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

[20] GRAZIANO, P. Speed Up Your Web Site with Varnish. *Linux Journal 2013* (Mar. 2013).

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

[22] INTEL. Data Plane Develpment Kit. http://dpdk.org/, 2016.

[23] ITZHAKY, S., BANERJEE, A., IMMERMAN, N., LAHAV, O., NANEVSKI, A., AND SAGIV, M. Modular reasoning about heap paths via effectively propositional formulas. In *POPL* (2014).

[24] JUNKER, H. OWASP Enterprise Security API. *Datenschutz und Datensicherheit 36* (2012), 801–804.

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

[26] 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).

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

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

[29] 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).

[30] KOROVIN, K. Non-cyclic sorts for first-order satisfiability. In *Frontiers of Combining Systems*, P. Fontaine, C. Ringeissen, and R. Schmidt, Eds., vol. 8152 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 214–228.

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

[32] LEBLOND, E. Secure use of iptables and connection tracking helpers. `https://goo.gl/wENPDy` retrieved 09/18/2016, 2012.

[33] 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.

[34] 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.

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

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

[37] 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).

[38] PEREIRA, H., RIBEIRO, A., AND CARVALHO, P. L7 classification and policing in the pfsense platform. *Atas da CRC* (2009).

[39] PISKAC, R., DE MOURA, L. M., AND BJØRNER, N. Deciding Effectively Propositional Logic using DPLL and substitution sets. *J. Autom. Reasoning 44*, 4 (2010).

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

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

[42] PURDY, G. N. *Linux iptables - pocket reference: firewalls, NAT and accounting*. 2004.

[43] QUALYS INC. IronBee. `http://ironbee.com/` retrieved 09/18/2016, 2016.

[44] RISTIC, I. *ModSecurity Handbook*. 2010.

[45] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *LISA* (1999).

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

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

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

[49] 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).

[50] SKOWYRA, R., LAPETS, A., BESTAVROS, A., AND KFOURY, A. A Verification Platform for SDN-Enabled Applications. In *HiCoNS* (2013).

[51] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. SymNet: scalable symbolic execution for modern networks. *CoRR abs/1604.02847* (2016).

[52] STONESOFT. StoneGate Administrator's Guide v5.3. `https://goo.gl/WNcaQj` retrieved 09/18/2016, 2011.

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

[54] TARREAU, W. HAProxy-the reliable, high-performance TCP/HTTP load balancer. `http://haproxy.lwt.eu`, 2012.

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

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

[57] VERIFLOW INC. Veriflow. `http://www.veriflow.net/`, 2016.

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

[59] ZELDOVICH, N., BOYD-WICKIZER, S., AND MAZIÈRES, D. Securing distributed systems with information flow control. In *NSDI* (2008).

[60] 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).

A Real-world Models

Here we provide models for real world middleboxes, in addition to the ones listed in §2.3.

NATs We also examined the NAT functions of `iptables` and pfSense. Each of them provides both a "source NAT" (SNAT) function, which allows end-hosts with private addresses to initiate Internet connections, and a "destination NAT" (DNAT) function, which allows end-hosts with private addresses to accept Internet connections.

Listing 4 shows a VMN model for both SNATs. The configuration input is the box's public address (Line 1). The `remapped_port` oracle returns an available port to be used for a new connection, abstracting away the details of how the port is chosen (Line 2); during verification, we assume that the oracle may return any port. The `active` and `reverse` maps associate private addresses to ports and vice versa (Lines 3–4). The forwarding model: on receiving a packet that belongs to an established connection, the necessary state is retrieved from either the `reverse` map—when the packet comes from the public network (Lines 6–10)—or the `active` map—when the packet comes from the private network (Lines 11–14); on receiving a packet from the private network that is establishing a new connection, the oracle is consulted to obtain a new port (Line 19) and the relevant state is recorded in the maps (Lines 20–21) before the packet is forwarded appropriately (Line 22). To model a SNAT that uses a pool of public addresses (as opposed to a single address), we instantiate one SNAT object (as defined in Listing 4) per address and define an oracle that returns which SNAT object to use for each connection.

Listing 4: Model for a Source NAT

```
1  class SNAT (nat_address: Address){
2    abstract
       ↪ remapped_port (p: Packet): int
3    val active : Map[Flow, int]
4    val reverse
       ↪ : Map[port, (Address, int)]
5    def model (p: Packet) = {
6      dst(p) == nat_address =>
7        (dst, port)
           ↪ = reverse[p.dst_port];
8        p.dst = dst;
9        p.dst_port = port;
10       forward(Seq(p))
11     active.contains(flow(p)) =>
12       p.src = nat_address;
13       p.src_port = active(flow(p));
14       forward(Seq(p))
15     _ =>
16       address = p.src;
17       port = p.src_port
18       p.src = nat_address;
19       p.src_port = remapped_port(p);
20       active(flow(p)) = p.src_port;
21       reverse(p.src_port)
           ↪ = (address, port);
22       forward(Seq(p))
23   }
24 }
```

Listing 5 shows a VMN model for both DNATs. The configuration input is a map associating public address/port pairs to private ones (Line 1). There are no oracles. The `reverse` map associates private address/port pairs to public ones (Line 2). The forwarding model: on receiving, from the public network, a packet whose destination address/port were specified in the configuration input, the packet header is updated accordingly and the original destination address/port pair recorded in the `reverse` map (Lines 3–9); conversely, on receiving, from the private network, a packet that belongs to an established connection, the necessary state is retrieved from the `reverse` map and the packet updated accordingly (Lines 10–13); any other received packets pass through unchanged.

Listing 5: Model for a Destination NAT

```
1  class DNAT(translations:
     ↪ Map[(Address,
     ↪ int), (Address, int)]) {
2    val reverse:
       ↪ Map[Flow, (Address, int)]
3    def model (p: Packet) = {
4      translations.contains((p.dst,
         ↪ p.dst_port)) =>
5        dst = p.dst;
6        dst_port = p.dst_port;
7        p.dst = translations[(p.dst,
           ↪ p.dst_port)]._1;
8        p.dst_port
           ↪ = translations[(p.dst,
```

```
            ↪ p.dst_port)]._2;
        reverse[flow(p)]
            ↪ = (dst, dst_port);
        forward(Seq(p))
      reverse.contains(flow(p)) =>
        p.src = reverse[flow(p)]._1;
        p.src_port
            ↪ = reverse[flow(p)]._2;
        forward(Seq(p))
      _ => forward(Seq(p))
    }
}
```

Gateways A network gateway often performs firewall, NAT, and/or IDS/IPS functions organized in a chain. In fact, both `iptables` and pfSense are modular gateways that consist of configurable chains of such functions (*e.g.,* in `iptables` one can specify a `CHAIN` parameter for NAT rules). To model such a modular gateway, we have written a script that reads the gateway's configuration and creates a pipeline of models, one for each specified function.

Listing 6: Model for a Load Balancer

```
class LoadBalancer(backends:
    ↪ List[Address]) {
  val assigned: Map[Flow, Address]
  abstract
      ↪ pick_backend(p: Packet): int
  def model (p: Packet) = {
    assigned.contains(flow(p)) =>
      p.dst = assigned[flow(p)]
      forward(Seq(p))
    _ =>
      assigned[flow(p)] =
          ↪ backends[pick_backend(p)]
      p.dst = assigned[flow(p)]
      forward(Seq(p))
  }
}
```

Load-balancers A load-balancer performs a conceptually simple function: choose the backend server that will handle each new connection and send to it all packets that belong to the connection. We considered two kinds: systems like HAProxy [54], which control the backend server that will handle a packet by rewriting the packet's destination address, and systems like Maglev [11], which control the backend server through the packet's output network interface. Both kinds accept as configuration the set of available backend servers (either their addresses or the network interface of the load balancer that leads to each server). Our load-balancer (Listing 6) uses an oracle to determine which server handles a connection, during verification the decission process can therefore choose from any of the available servers.

Listing 7: Model for a simple cache

```
class Cache(address: Address, acls:
    ↪ Set[(Address, Address)]) {
 abstract request(p: Packet): int
 abstract response(p: Packet): int
 abstract new_port(p: Packet): int
 abstract cacheable(int): bool
 val outstanding_requests:
     ↪ Map[Flow, int]
 val outstanding_conns:
     ↪ Map[Flow, Flow]
 val cache_origin: Map[int, Host]
 val origin_addr: Map[int, Address]
 val cached: Map[int, int]
 def model (p: Packet) = {
  val p_flow = flow(p);
  outstanding_request.contains(p_flow)
      ↪ &&
  cacheable(
    outstanding_request[p_flow]) =>
    cached[outstanding_request[p_flow]]
            = response(p);
    ...
    p.src =
      outstanding_conns[p_flow].src;
    p.dst =
      outstanding_conns[p_flow].dst;
    ...
    forward(Seq(p))
  outstanding_request.contains(p_flow)
      ↪ &&
  !cacheable(
     outstanding_request[p_flow]] =>
    p.src =
      outstanding_conns[p_flow].src;
    ...
    forward(Seq(p))
  cached.contains(request(p)) &&
  !acls.contains(
    p.src, origin_addr[request(p)]) =>
      p.src = p_flow.dst;
      ...
      p.origin =
          ↪ cache_origin[request(p)];
      forward(Seq(p))
  !acls.contains(p.src, p.dst) =>
      p.src = address;
      p.src_port = new_port(p);
      outstanding_conns[flow(p)]
          ↪ = p_flow;
      outstanding_requests[flow(p)]
          ↪ = request(p)
      forward(Seq(p))
  _ =>
      forward(Empty)
  }
}
```

Caches We examined two caches: Squid [18] and Apache Web Proxy. These systems have a rich set of config-

uration parameters that control, *e.g.,* the duration for which a cached object is valid, the amount of memory that can be used, how DNS requests are handled, etc. However, most of them are orthogonal to our invariants. We therefore consider only a small subset of the configuration parameters, in particular, those that determine whether a request should be cached or not, and who has access to cached content.

Listing 7 shows a model that captures both caches. Configuration input is a list of rules specifying which hosts have access to content originating at particular servers (Line 1). We define four oracles, among them `cacheable`, which determines whether policy and cache-size limits allow a response to be cached (Line 5). The forwarding model captures the following behavior: on receiving a request for content that is permitted by configuration, we check whether this content has been cached (Line 32–38); if so, we respond to the request, otherwise we forward the request to the corresponding server (Line 39–44); on receiving a response from a server, we check if the corresponding content is cacheable, if so, we cache it (Line 15–24); and regardless of its cacheability forward the response to the client who originally requested this content.

We treat each request and response as a single packet, whereas, in reality, they may span multiple packets. This greatly simplifies the model and—since we do not check performance-related invariants—does not affect verification results.

B Logical Models

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, or a middlebox can process a previously received packet. We do not attempt to model the *likely* order of these various events, but instead consider all such orders in search of invariant violations. In this case Z3 acts as a *scheduling oracle* that assigns an event to each timestep, subject to the standard causality constraints, *i.e.,* we ensure that packets cannot be received before being sent, and packets sent on the same link are not reordered.

VMN models middleboxes and networks using quantified logical formula, which are axioms describing how received packets are treated. Oracles in VMN are modeled as uninterpreted function, *i.e.,* Z3 can assign any (convinient) value to a given input to an oracle. We also provide Z3 with the negation of the invariant to be verified, which is specified in terms of sets of packets (or data) that are sent or received. 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.

Symbol	Meaning
Events	
$rcv(d,s,p)$	Destination d receives packet p from source s.
$snd(s,d,p)$	Source s sends packet p to destination d.
Logical Operators	
$\Box P$	Condition P holds at all times.
$\blacklozenge P$	Event P occurred in the past.
$\neg P$	Condition P does not hold (or event P does not occur).
$P_1 \wedge P_2$	Both conditions P_1 and P_2 hold.
$P_1 \vee P_2$	Either condition P_1 or P_2 holds.

Table 1: Logical symbols and their interpretation.

B.1 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) [33] 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 converts LTL formulae to first-order formulas (required by Z3) by explicitly quantifying over time. Table 1 lists the symbols used to express formula in this section.

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,

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

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,

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

indicates that d will never receive any packet from s.

Header fields and oracles are represented using functions, *e.g., src*(p) and $dst(p)$ represent the source and destination address for packet p, and $mallicious(p)$ acts as the `mallicious` oracle from Listing 1.

B.2 Reachability Invariants

Reachability invariants can be be generally specifies as:

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

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

Listing 8: Model for a learning firewall

```
class LearningFirewall (acl:
    ↪ Set[(Address, Address)]) {
  val established : Set[Flow]
  def model (p: Packet) = {
    established.contains(flow(p)) =>
      forward (Seq(p))
    acl.contains((p.src, p.dest)) =>
      established += flow(p)
      forward(Seq(p))
    _ =>
      forward(Seq.empty)
  }
}
```

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):

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

B.3 Modeling Middleboxes

Middleboxes in VMN are specified 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.

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 8 results in the following axioms:

$$\textbf{established}(flow(p)) \implies (\blacklozenge((\neg fail(\mathbf{f})) \wedge (\blacklozenge rcv(\mathbf{f},p)))) \wedge \textbf{acl}(src(p),dst(p))$$

$$send(\mathbf{f},p) \implies (\blacklozenge rcv(\mathbf{f},p)) \wedge (\textbf{acl}(src(p),dst(p)) \vee \textbf{established}(flow(p)))$$

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

We translate models to formulas by finding the set of packets appearing in the `forward` function appearing at the end of each match statement, and translating the statement so that the middlebox sending that set of packet implies that (a) previously a packet matching an appropriate criterion was received; and (b) middlebox state was appropriately updated. We combine all branches where the same set of packets are updated using logical conjunction, *i.e.,* implying that one of the branches was taken.

B.4 Modeling Networks

VMN uses *transfer functions* to specify a network's forwarding behavior. The transfer function for a network is a function from a located packet 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 translates such a 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).

$$\forall n,p : \Box fail(X) \wedge ... snd(A,n,p) \implies n = \Omega$$

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

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.

C Formal Definition of Slices

Given a network $N = (V, E, P)$, with network transfer function N_T, we define a subnetwork Ω to be the network formed by taking a subset $V|_\Omega \subseteq \Omega$ of nodes, all the links connecting nodes in $V|_\Omega$ and a subset of packets $P|_\Omega \subseteq P$ from the original network. We define a subnetwork Ω to be a slice if and only if Ω is closed under *forwarding* and *state*, and $P|_\Omega$ is maximal. We define $P|_\Omega$ to be maximal if and only if $P|_\Omega$ includes all packets from P whose source and destination are in $V|_\Omega$.

We define a subnetwork to be *closed under forwarding* if and only if (a) all packets $p \in P|_\Omega$ are located inside Ω, *i.e.,* $p.loc \in V|_\Omega \forall p \in P|_\Omega$; and (b) the network transfer function forwards packets within Ω, *i.e.,* $N_T(p) \subseteq P|_\Omega \forall p \in P|_\Omega$.

The definition for being *closed under state* is a bit more involved, informally it states that all states reachable by a middlebox in the network is also reachable in the slice. More formally, associate with the network a set of states S where each state $s \in S$ contains a multiset of pending packets $s.\Pi$ and the state of each middlebox ($s.m_0$ for middlebox m_0). Given this associated set of states we can treat the network as a state machine, where each transition is a result of one of two actions:

- An end host $e \in V$ generates a packet $p \in P$, in this case the system transitions to the state where all packets in $N_T(p)$ (where N_T is the network transfer function defined above) are added to the multiset of pending packets.
- A packet p contained in the pending state is delivered to $p.loc$. In cases where $p.loc$ is an end host, this merely results in a state where one p is removed from the multiset of pending packets. If however, $p.loc$ is a middlebox we transition to the state gotten by (a) removing p from pending packets, (b) updating the state for middlebox $p.loc$ and (c) for all packets p' forwarded by the middlebox, adding $N_T(p')$ to the set of pending packets.

In this model, invariants are predicates on states, an invariant is violated if and only if the system transitions to a state where the invariant is true.

Observe that this definition of state machines can be naturally restricted to apply to a subnetwork Ω that is closed under forwarding, by associating a set of states S_Ω containing the state only for those middleboxes in Ω. Finally, given some subnetwork Ω we define a restriction function σ_Ω that relates the state space for the whole network S to S_Ω the state space for the subnetwork. For any state $s \in S$, σ simply drops all packets not in $P|_\Omega$ and drops the state for all middleboxes $m \notin V|_\Omega$.

Finally, we define some state $s \in S$ as reachable in N if and only if there exists a sequence of actions starting from the initial state (where there are no packets pending and all middleboxes are set to their initial state) that results in network N getting to state s. A similar concept of reachability of course also applies to the state machine for Ω. Finally, we define a subnetwork Ω to be *closed under state* if and only if S_Ω, the set of states reachable in Ω is the same as the projection of the set of states reachable in the network, more formally $S_\Omega = \{\sigma_\Omega(s), s \in S, s \text{ reachable in } N\}$.

When a subnetwork Ω is closed under *forwarding* and *state*, one can establish a bisimulation between the slice and the network N; informally this implies that one can find a relation such that when we restrict ourselves to packets in $p \in P|_\Omega$ then all transitions in N have a corresponding transition in Ω, corresponding here implies that the states in N are the same as the states in Ω after projection. Since by definition for any slice $P|_\Omega$ the set of packets is maximal, this means that every state reachable in N has an equivalent projection in Ω.

Finally, we define an invariant I to be evaluable in a subnetwork Ω if and only if for all states $s_1, s_2 \in S$ $\sigma_\Omega(s_1) = \sigma_\Omega(s_2) \implies I(s_1) = I(s_2)$, *i.e.,* if the invariant does not depend on any state not captured by Ω. As a result of the bisimulation between network N and slice Ω, it is simple to see that an invariant I evaluable in Ω holds in network N if and only if it also holds in Ω. Thus once a slice is found, we can verify any invariants evaluable on it and trivially transfer the verification results to the whole network.

Note, that we can always find a slice of the network on which an invariant can be verified, this is trivially true since the network itself is its own slice. The challenge therefore lies in finding slices that are significantly smaller than the entire network, and of sizes that do not grow as more devices are added to the network. The nodes that are included in a slice used to verify an invariant trivially depend on the invariant being verified, since we require that the invariant be evaluable on the slice. However, since slices must be closed under state, their size is also dependent on the types of middleboxes present in the network. Verification for network where all middleboxes are such that their state can be partitioned (based on any criterion, *e.g.,* flows, policy groups, etc.) are particularly amenable to this approach for scaling. We present two concrete classes of middleboxes that contain all of the examples we have listed previously in §2.3 that allow verification to be performed on slices whose size is independent of the network's size.

D Decidability

As noted in §4.2, first-order logic is undecidable. Further, verifying a network with mutable datapaths is undecidable in general, such networks are Turing complete. However, we believe that we can express networks obeying the following restrictions in a decidable fragment of first-order logic:

1. All middleboxes used in the network are passive, *i.e.,* they send packets only in response to packets received by them. In particular this means that every packet sent by

a middlebox is causally related to some packet previously sent by an end-host.

2. A middlebox sends a finite number of packets in response to any packets it receives.

3. All packet processing pipelines are loop free, *i.e.,* any packets that enter the network are delivered or dropped in a finite number of steps.

We now show that we can express middleboxes and networks meeting this criterion in a logic built by extending "effectively propositional logic" (*EPR*). EPR is a fundamental, decidable fragment of first-order logic [39], where all axioms are of the form $\exists^*\forall^*$, and the invariant is of the form $\forall^*\exists^*$. Neither axioms nor invariants in this logic can contain function symbols. EPR-F extends EPR to allow some unary functions. To guarantee decidability, EPR-F requires that there exist a finite set of compositions of unary functions U, such that any composition of unary functions can be reduced to a composition in U. For example, when a single unary function f is used, we require that there exist k such that $f^k(x) = f^{k-1}(x)$ for all k. Function symbols that go from one type to another are allowed, as long as their inverse is not used [30] (*e.g.,* we can use *src* in our formulas since it has no inverse). Prior work [23] has discussed mechanisms to reduce EPR-F formulas to EPR.

We can translate our models to EPR-F by:

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 statement

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

is translated to the formula

$$\forall e : rcv(e) \Rightarrow snd(cause(e)) \wedge \ldots \wedge t(cause(e)) < t(e)$$

We also add the axiom, $\forall e : snd(e) \Rightarrow cause(e) = e$ which says that a *snd* event has no cause, ensuring idempotency.

To show that our models are expressible in EPR-F, we need to show that all unary functions introduced during this conversion meet the required closure properties. Intuitively, all function introduced by us track the causality of network events. Our decidability criterion imply that every network event has a finite causal chain. This combined with the axiom that $cause(e)$ is idempotent implies that the functions meet the closure properties. However, for Z3 to guarantee termination, explicit axioms guaranteeing closure must be provided. Generating these axioms from a network is left to future work. In our experience, Z3 terminates on networks meeting our criterion even in the absence of closure axioms.