Abstract Interpretation of Stateful Networks

Kalev Alpernas[1], Roman Manevich[2], Aurojit Panda[3], Mooly Sagiv[1], Scott Shenker[4], Sharon Shoham[1], and Yaron Velner[5]

[1] Tel Aviv University
[2] Ben-Gurion University of the Negev
[3] NYU
[4] UC Berkeley
[5] Hebrew University of Jerusalem

Abstract. Modern networks achieve robustness and scalability by maintaining states on their nodes. These nodes are referred to as middleboxes and are essential for network functionality. However, the presence of middleboxes drastically complicates the task of network verification. Previous work showed that the problem is undecidable in general and EXPSPACE-complete when abstracting away the order of packet arrival.

We describe a new algorithm for conservatively checking isolation properties of stateful networks. The asymptotic complexity of the algorithm is polynomial in the size of the network, albeit being exponential in the maximal number of queries of the local state that a middlebox can do, which is often small.

Our algorithm is sound, i.e., it can never miss a violation of safety but may fail to verify some properties. The algorithm performs on-the fly abstract interpretation by (1) abstracting away the order of packet processing and the number of times each packet arrives, (2) abstracting away correlations between states of different middleboxes and channel contents, and (3) representing middlebox states by their effect on each packet separately, rather than taking into account the entire state space. We show that the abstractions do not lose precision when middleboxes may reset in any state. This is encouraging since many real middleboxes reset, e.g., after some session timeout is reached or due to hardware failure.

1 Introduction

Modern computer networks are extremely complex, leading to many bugs and vulnerabilities that affect our daily life. Therefore, network verification is an increasingly important topic addressed by the programming languages and networking communities [16,4,14,15,13,29,22,11]. Previous network verification tools leverage a simple network forwarding model, which renders the datapath *immutable*. That is, normal packets going through the network do not change its forwarding behaviour, and the control plane explicitly alters the forwarding state at relatively slow time scales.

While the notion of an immutable datapath supported by an assemblage of routers makes verification tractable, it does not reflect reality. *Middleboxes* are widespread in modern enterprise networks [30]. A simple example of a middlebox is a stateful firewall which permits traffic from untrusted hosts only after they have received a packet from a trusted host. Middleboxes, such as firewalls, WAN optimizers, transcoders, proxies,

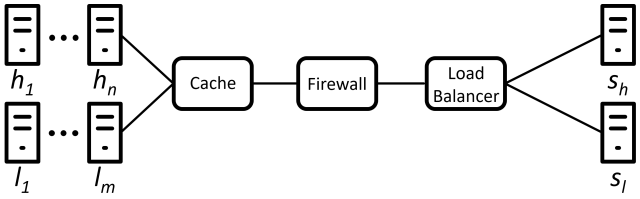


Fig. 1: A middlebox chain with a buggy topology.

load-balancers and the like, are the most common way to insert new functionality in the network datapath, and are commonly used to improve network performance and security. Middleboxes maintain a state and may change their state and forwarding behavior in response to packet arrivals. While useful, middleboxes are a common source of errors in the network [26].

As a simple example, consider the middlebox chain described in Fig. 1. In this network, a firewall is used to ensure that low security hosts ($l_1, \ldots, l_m$) do not receive packets from the S_h server, and a cache and load balancer are used to improve performance. Unfortunately, the configuration of the network is incorrect since the cache may respond with a stored packet, bypassing the security policy enforced by the firewall. Swapping the order of the cache and the firewall results in a correct configuration.

Safety of Stateful Networks. We address the problem of verifying safety of networks with middleboxes, referred to as *stateful networks*. We target verification of *isolation* properties, namely, that packets sent from one host (or class of hosts) can never reach another host (or class of hosts). Yet, our approach is sound for any safety property. For example, it detects the safety violation described in Fig. 1, and verifies the safety of the correct configuration of this network.

Our focus is on verifying the configuration of stateful networks, i.e., addressing errors that arise from the interactions between middleboxes, and not from the complexity of individual middleboxes. Hence, we follow [34] and use an abstraction of middleboxes as finite-state programs. Previous work [34,31] has shown that many kinds of middleboxes, including proxy, cache proxy, NAT, and various kinds of load-balancers can be modeled in this way, sometimes using non-determinism to over-approximate the behaviour, e.g. to model timers, counters, etc. Since we are interested in safety properties, such an abstraction (overapproximation) is suitable.

As shown in [34], it is undecidable to check safety properties in general and isolation in particular, even for middleboxes with a finite state space, and even when the order of packets pending for each middlebox is abstracted away the complexity is quite high (EXPSPACE-complete). Therefore, in this paper we develop additional abstractions for scaling up the verification.

Our approach. This paper makes a first attempt to apply abstract interpretation [6] to automatically prove the safety of stateful networks. Our approach combines sound network-level abstractions and middlebox-level abstractions that, together, make the verification task tractable. Roughly speaking, we apply (i) order abstraction [34], abstracting away the order of packets on channels, (ii) counter abstraction [25], abstracting

away their cardinality, (iii) network-level Cartesian abstraction [6,10,12], abstracting away the correlation between the states of different middleboxes and different channel contents, and (iv) middlebox-level Cartesian abstraction, abstracting away the correlation between states of different packets within each middlebox.

The network-level abstractions, (i)-(iii), lead to a chaotic iteration algorithm that is polynomial in the state space of the individual middleboxes and packets. However, the number of middlebox states can be exponential in the size of the network. For example, a firewall may record the set of trusted hosts and thus its states are subsets of hosts. Therefore, the resulting analysis is exponential in the number of hosts[6].

The middlebox-level Cartesian abstraction, (iv), is the key to reducing the complexity to polynomial. The crux of this abstraction is the observation that the abstraction of middleboxes as reactive processes that query and update their state in a restricted way (e.g., [34]) allows to represent a middlebox state as a product of loosely-coupled *packet states*, one per potential packet. This lets us define a novel, non-standard, semantics of middlebox programs that we call *packet effect semantics*. The packet effect semantics is equivalent (bisimilar) to the natural semantics. However, while the natural semantics is monolithic, the packet effect semantics decomposes a single middlebox state into the parts that determine the forwarding behavior of different packets, and therefore facilitates the use of Cartesian abstraction to further reduce the complexity.

One of the main challenges for abstract interpretation is evaluating its precision. To address this challenge, we provide sufficient conditions that ensure precision of our analysis. Namely, we show that if the network is safe in the presence of packet reordering and middlebox reverts, where a middelbox may revert to its initial state at any moment, then our analysis is guaranteed to be precise, and will never report false alarms. This is, to a great extent, due to the packet effect semantics, which allows to use a middlebox-level Cartesian abstraction without incurring additional precision loss for such networks. Notice that middlebox reverts enable modelling arbitrary hardware failures, which have not been addressed by previous work on stateful network verification (e.g., in [34]). Surprisingly, verification becomes easier under the assumption that middleboxes may reset at any time. (Recall that for arbitrary unordered networks safety checking is EXPSPACE-complete.)

In summary, the main contributions of this paper are

- We introduce the first abstract interpretation algorithm for verifying safety of stateful networks, whose time complexity is polynomial in the size of the network, albeit exponential in the maximal number of queries of the local state that a middlebox can do, which is often small even for complex middelboxes (up to 5 in our examples).
- We develop *packet effect semantics*, a non-standard semantics of middelbox programs that facilitates middlebox-level Cartesian abstraction, reducing the complexity of the abstract interpretation algorithm from exponential in the size of the network to polynomial without incurring any additional precision loss for unordered reverting networks.

[6] Unfortunately, if the set of hosts is not fixed, the safety problem becomes undecidable (even under the unordered abstraction) (Appendix F). This means that, in general, it is not possible to alleviate the dependency of the complexity on the hosts.

- We provide sufficient conditions for precision of the analysis that have a natural interpretation in the domain of stateful networks: ignoring the order of packet processing and letting middleboxes revert to their initial states at any time.
- We prove lower bounds on the complexity of safety verification in the presence of packet reordering and/or middlebox reverts, showing that our algorithm is essentially optimal.
- We implement our analysis and show that it scales well with the number of hosts and middelboxes in the network.

We defer proofs of key claims to App. B .

2 Expressing Middlebox Effects

This section defines our programming language for modeling the abstract behavior of middleboxes in the network. Our modeling language is independent of the particular network topology, which is defined in Sec. 3. The proposed language, AMDL (**A**bstract **M**iddlebox **D**efinition **L**anguage), is a restricted form of OCCAM [28], similar to the languages of [34,31].

We first define the syntax and informal semantics of AMDL (Sec. 2.1); we then define a formal "standard" *relation effect semantics* (Sec. 2.2); we continue by defining an alternative *packet effect* semantics (Sec. 2.3), which is bisimilar to the relation effect semantics (Sec. 2.4); and finally we present a localized version of the packet effect semantics (Sec. 2.5), which is suitable for Cartesian abstraction.

Packets. Middlebox behavior in our model is defined with respect to packets that consist of a fixed, finite, number of packet fields, ranging over finite domains. As such, a packet $p \in P$ in our formalism is a tuple of packet fields over predefined finite sorts. In our examples, a packet is a tuple $\langle s, d, t \rangle$, where s, d are the source and destination hosts, respectively, taken from a finite set of hosts H, and t is a packet tag (or type) that ranges over a finite domain T. In this case, $|P|$ is polynomial in $|H|$. (Our approach is also applicable when additional fields are added, e.g., for modeling the packet's payload via an abstract finite domain.)

2.1 Syntax and Informal Semantics

Fig. 3 describes the syntax of the AMDL language[7]. Middleboxes are implemented as reactive processes, with events triggered by the arrival of packets. If multiple packets are pending, the AMDL process non-deterministically reads a packet from one of the incoming channels of the process. The packet processing code is a loop-free block of guarded-commands, which may update relations and forward potentially modified packets to some of the output ports. AMDL uses *relations* over finite domains to store the middlebox state. These are the only data structures allowed in AMDL. The only relation operations allowed are inserting a value to a relation, removing a value from a relation,

[7] In the code examples, we write `p` for the triple `(src,dst,type)` and use access path notation to refer to the fields, e.g., `p.src`.

```
sfirewall = do
  internal_port ? p =>
    if
      p.dst in trusted => external_port ! p
      □
      p.type = 0 => // request packet
        external_port ! p;
        requested(p.dst) := true
    fi
  □
   external_port ? p =>
     if
       p.src in trusted => internal_port ! p
       □
       p.type = 1 and p.src in requested =>
         // response packet with a request
         trusted(p.src) := true
     fi
  od
```

Fig. 2: AMDL code for session firewall.

and *membership queries* — checking whether a value is in a relation. For a membership query of the form $\overline{a}$ **in** r, we denote the relation, r, used in the query by $\mathit{rel}(q)$ and denote the tuple of atoms $\overline{a}$ by $\mathit{atoms}(q)$. For example, the code for a session firewall is depicted in Fig. 2.

Middleboxes may enforce safety properties using the **abort** command. For example, an isolation middlebox would abort when a forbidden packet is received.

2.2 Middlebox Relation Effect Semantics

We now sketch the semantics of AMDL. The definitions below supply a part of the full network semantics, which is given in Sec. 3.

Middlebox States. Each middlebox $m \in M$ maintains its own local state as a set of relations. The domain of a relation r defined over sorts $s_{1..k}$ is $D(r) \stackrel{\text{def}}{=} D(s_1) \times \ldots \times D(s_k)$, where $D(s_i)$ is the domain of sort s_i. We use $\mathit{rels}(m)$ to denote the set of relations in m, and $D(m)$ to denote the union of $D(r)$ over $r \in \mathit{rels}(m)$.

The *middlebox state* of m is then a function $s \in \Sigma^{\mathrm{R}}[m] \stackrel{\text{def}}{=} \mathit{rels}(m) \to \wp(D(m))$, mapping each $r \in \mathit{rels}(m)$ to $v \subseteq D(r)$. In addition, we introduce a unique *error* middlebox state, denoted *err*. We assume that $\mathit{err} \in \Sigma^{\mathrm{R}}[m]$ for every middlebox m.

Middlebox Transitions. Middlebox transitions have the form

$$\xrightarrow{(p,c)/(p_i,c_i)_{i=1..k}}_{\mathrm{R}} \subseteq \Sigma^{\mathrm{R}}[m] \times \Sigma^{\mathrm{R}}[m]$$

$\langle mbox \rangle$ ::= $m =$ **do** $\langle pblock \rangle$ [□ $\langle pblock \rangle$]* **od**
$\langle pblock \rangle$ ::= c **?** $\overline{pfld}$ ⇒ $\langle gc \rangle$
$\langle gc \rangle$ ::= $\langle cond \rangle$ ⇒ $\langle action \rangle$ | **if** $\langle gc \rangle$ [□ $\langle gc \rangle$]* **fi**
$\langle action \rangle$ ::= $\langle action \rangle$ **;** $\langle action \rangle$ | c ! $\overline{\langle atom \rangle}$ | $r(\overline{\langle atom \rangle})$ **:=** $\langle cond \rangle$ | **abort**
$\langle cond \rangle$ ::= **true** | $\langle cond \rangle$ **and** $\langle cond \rangle$ | **not** $\langle cond \rangle$ | $\langle atom \rangle = \langle atom \rangle$ | $\overline{\langle atom \rangle}$ **in** r
$\langle atom \rangle$ ::= *pfld* | *const*

Fig. 3: AMDL syntax. $\overline{e}$ denotes a comma-separated list of elements drawn from the domain e. **abort** imposes a safety condition. c ? p reads p from a channel c and c ! p writes p into c. We write m for a middlebox name, r for a relation name, and c for a channel name. We write *const* for a constant symbol and *pfld* for identifiers used to match fields in packets, e.g., `src`. Non-deterministic choice is denoted by □.

where (p, c) denotes packet-channel at the input, and $(p_i, c_i)_{i=1..k}$ is the sequence of packet-channel pairs that the middlebox outputs.

For example, for $s \stackrel{\text{def}}{=} [\texttt{requested} \mapsto \emptyset, \texttt{trusted} \mapsto \emptyset]$, the guarded command corresponding to the internal port of the firewall middlebox (Fig. 2) induces a transition $s \xrightarrow{((h_1,h_2,0),\vec{c_{in}})/((h_1,h_2,0),\vec{c_{out}})}_{\text{R}} s'$ where $s' \stackrel{\text{def}}{=} [\texttt{requested} \mapsto \{h_2\}, \texttt{trusted} \mapsto \emptyset]$.

abort commands induce transitions to the *err* state.

The formal definition of the middlebox transitions appears in App. C.

2.3 Middlebox Packet Effect Semantics

We now present a semantics that is equivalent to the relation effect semantics. The semantics is based on an alternative (yet isomorphic) representation of middlebox states that reveals a loose coupling between the parts of the state that are relevant for different packets. This loose coupling then facilitates a Cartesian abstraction that abstracts away correlations between packets in the same state.

Packet Effect Representation of Middlebox State Recall that in Sec. 2.1 we restrict the values that can be used in a middlebox program to either constants or the values of fields of the currently processed packet. We do not allow extracting tuples from the relation (e.g., by having a `get` command, or by iterating over the contents of the relation). Instead, we limit the interaction with the relation to checking whether a tuple (that consists of packet fields or constants) exists in the relation. Consequently, instead of storing the contents of all relations, the state of the middlebox can be represented by mapping all potential packets in the network to their effect on the middlebox. Specifically, we map each packet and membership query in the program to whether that membership query will be evaluated to True when the program is executed on that packet.

For every middlebox m, we denote by $Q(m)$ the set of membership queries in m's program. (We need not distinguish between different instances of the same query.) For example, in Fig. 2, $Q(fw) = \{$`p.dst in trusted`, `p.src in trusted`, `p.src in requested`$\}$.

The *packet effect state* of a middlebox m is a function $s \in \Sigma^{\mathrm{P}}[m] \stackrel{\text{def}}{=} P \to Q(m) \to \{\mathsf{True}, \mathsf{False}\}$, mapping each packet $p \in P$ to the evaluation of all queries of m when p is the input packet, thus capturing the way in which p traverses m's program. We refer to $s(p) \in Q(m) \to \{\mathsf{True}, \mathsf{False}\}$ as the *packet state* of packet p in middlebox state s. We extend $\Sigma^{\mathrm{P}}[m]$ with an error state $\lambda p \in P.\ \mathit{err}$, which is also denoted *err*.

Middlebox Transition Relation in the Packet Space The semantics of middlebox m in the packet space is defined via a transition relation $\xrightarrow{(p,c)/(p_i,c_i)_{i=1..k}}_{\mathrm{P},m} \subseteq \Sigma^{\mathrm{P}}[m] \times \Sigma^{\mathrm{P}}[m]$. When m is clear, we omit it from the notation. A transition $\tilde{s} \xrightarrow{(p,c)/(p_i,c_i)_{i=1..k}}_{\mathrm{P}} \tilde{s}'$ exists if (one of) the sequence of operations applied on $\tilde{s}$ when packet p arrives on channel c outputs $(p_i, c_i)_{i=1..k}$ and leads to $\tilde{s}'$.

The semantics of operations is defined similarly to the "standard" relation effect semantics. The semantics of error and output actions (that do not change the middlebox state) is straightforward. Next, we explain the semantics of the operations that depend on or change the middlebox state — membership queries and relation updates.

Consider a membership query q. Let $\tilde{s}$ be the middlebox state before evaluating q, i.e., $\tilde{s}$ is the state that results from executing all previous relation updates, and let p be the packet that invoked the middlebox transition. Then q is evaluated to $\tilde{s}(p)(q)$.

Next, consider a relation update. A relation update $\mathrm{r}(\overline{a})$ **:=***cond* updates the packet states of all packets that are affected by the operation. This is done as follows. As before, let $\tilde{s}$ be the intermediate state of m right before executing the operation, and let p be the packet that the middlebox program is operating on. Consider the case where *cond* evaluates to True in $\tilde{s}$, corresponding to addition of a value. (Removal of a value is symmetric.) We denote by $\overline{a}(p)$ the result of substituting each field name in $\overline{a}$ by its value in p. That is, $\overline{a}(p) \in D(\mathrm{r})$ is the value being added to r. This addition may affect the value of membership queries $q \in Q(m)$ with $\mathit{rel}(q) = \mathrm{r}$ (querying the same relation r) for other packets $\tilde{p}$ as well, in case that $\mathit{atoms}(q)(\tilde{p})$, i.e., the value being queried on $\tilde{p}$, is the same as the value $\overline{a}(p)$ being added to r. Therefore, the intermediate state obtained after the relation update operation has been applied is

$$\tilde{s}' = \lambda \tilde{p} \in P.\ \lambda q \in Q(m).\ \begin{cases} \mathsf{True}, & \text{if } \mathit{rel}(q) = \mathrm{r} \wedge \mathit{atoms}(q)(\tilde{p}) = \overline{a}(p). \\ \tilde{s}(\tilde{p})(q), & \text{otherwise.} \end{cases}$$

Namely, the operation updates to True the value of queries that coincide with the tuple of elements inserted to the relation.

Example 1. Consider the packet effect state $\tilde{s} \stackrel{\text{def}}{=} \lambda p.\ \lambda q.\mathsf{False} \in \Sigma^{\mathrm{P}}[\mathit{fw}]$ of the firewall (Fig. 2), where q ranges over the three membership queries in the code. Upon reading the packet $(h_1, h_2, 0)$ from an internal port, the middlebox performs a sequence of internal transitions which includes evaluating the expression "`p.type=0`" to True, outputting the packet $(h_1, h_2, 0)$ to the output port, and executing the command `requested(p.dst) := true`, which results in updating the state to:

$$\tilde{s}' \stackrel{\text{def}}{=} \lambda \tilde{p}.\ \lambda q.\ \begin{cases} \mathsf{True}, & \text{if } \mathit{rel}(q) = \texttt{requested} \wedge \mathit{atoms}(q)(\tilde{p}) = h_2 \\ \mathsf{False}, & \text{otherwise.} \end{cases}$$

That is, $\tilde{s}'((h_2,*,*))(\texttt{p.src in requested}) = \mathsf{True}$ and all the other values in $\tilde{s}'$ remain False as before. Therefore, $\tilde{s} \xrightarrow{((h_1,h_2,0),\vec{c_{in}})/((h_1,h_2,0),\vec{c_{out}})}_{\mathrm{P}} \tilde{s}'$. □

2.4 Bisimulation of Packet Effect Semantics and Relation Effect Semantics

We continue by showing that the transition systems defining the semantics of middleboxes in the packet effect and in the relation effect representations are bisimilar.

To do so, we first define a mapping $ps\colon \Sigma^{\mathrm{R}}[m] \to \Sigma^{\mathrm{P}}[m]$ from the relation state representation to the packet effect state representation. Recall that the relation state representation of middlebox states is $s \in \Sigma^{\mathrm{R}}[m] \stackrel{\text{def}}{=} \mathit{rels}(m) \to \wp(D(m))$. Given a state $s \in \Sigma^{\mathrm{R}}[m]$, *ps* maps it to the packet effect state s^{P} defined as follows:

$$s^{\mathrm{P}} \stackrel{\text{def}}{=} \lambda\tilde{p} \in P.\ \lambda q \in Q(m).\ \mathit{atoms}(q)(\tilde{p}) \in s(\mathit{rel}(q)).$$

That is, for every input packet $\tilde{p}$, the value in s^{P} of the query $q \in Q(m)$ is equal to the evaluation of the same query in s based on an input packet $\tilde{p}$.

Definition 1 (Bisimulation Relation). *For a middlebox m, we define the relation $\sim_m \subseteq \Sigma^R[m] \times \Sigma^P[m]$ as the set of all pairs (s, s^p) such that $s = s^p = \mathit{err}$ or $ps(s) = s^p$.*

Lemma 1. *Let $s \in \Sigma^R[m]$ and $\tilde{s} \in \Sigma^P[m]$ and $s \sim_m \tilde{s}$. Then the following holds:*

- *For every state $s' \in \Sigma^R[m]$, if $s \xrightarrow{(p,c)/o}_R s'$ then there exists a state $\tilde{s}' \in \Sigma^P[m]$ s.t. $\tilde{s} \xrightarrow{(p,c)/o}_P \tilde{s}'$ and $s' \sim_m \tilde{s}'$, and*
- *For every state $\tilde{s}' \in \Sigma^P[m]$ if $\hat{s} \xrightarrow{(p,c)/o}_P \tilde{s}'$ then there exists a state $s' \in \Sigma^R[m]$ s.t. $s \xrightarrow{(p,c)/o}_R s'$ and $s' \sim_m \tilde{s}'$.*

2.5 Locality of Packet-Effect Middlebox Transitions

In this section we present a locality property of the packet effect semantics that will allow us to efficiently compute an abstract transformer when applying a Cartesian abstraction. Namely, we observe that an execution of an operation $\mathrm{r}(\overline{a})$ **:=***cond*, in the context of processing an input packet p, potentially updates the packet states of all packets. However, for each packet $\tilde{p}$, the updated packet state $\tilde{s}'(\tilde{p})$ depends only on its pre-state $\tilde{s}(\tilde{p})$, the input channel c, the input packet p, and $\tilde{s}(p)$, which determines the value of queries; it is completely independent of the packet states of all other packets. Since, in addition, the execution path of the middlebox when processing input packet p depends only on the packet state of p, this form of *locality*, which we formalize next, extends to entire middlebox programs.

Definition 2 (Substate). *Let $\tilde{s} \in P \to Q(m) \to \{\mathsf{True}, \mathsf{False}\}$ be a packet effect state. We denote by $\tilde{s}|_{\{p,\tilde{p}\}} \in \{p, \tilde{p}\} \to Q(m) \to \{\mathsf{True}, \mathsf{False}\}$ the* substate *obtained from $\tilde{s}$ by dropping all packet states other than those of p and $\tilde{p}$. Let $\Sigma^P[m, p, \tilde{p}] \stackrel{\text{def}}{=} \{p, \tilde{p}\} \to Q(m) \to \{\mathsf{True}, \mathsf{False}\}$ denote the set of substates for p and $\tilde{p}$.*

Definition 3 (Substate transition relation). *We define the* substate transition relation $\xrightarrow{(p,c)/(p_i,c_i)_{i=1..k}}_{P[p,\tilde{p}]}$: $\Sigma^P[m,p,\tilde{p}] \times \Sigma^P[m,p,\tilde{p}]$ *as follows. A* substate transition $\tilde{s}[p,\tilde{p}] \xrightarrow{(p,c)/(p_i,c_i)_{i=1..k}}_{P[p,\tilde{p}]} \tilde{s}[p,\tilde{p}]'$ *holds if there exist* $\tilde{s}$ *and* $\tilde{s}'$ *such that* $\tilde{s}|_{[p,\tilde{p}]} = \tilde{s}[p,\tilde{p}]$, $\tilde{s}'|_{[p,\tilde{p}]} = \tilde{s}[p,\tilde{p}]'$ *and* $\tilde{s} \xrightarrow{(p,c)/(p_i,c_i)_{i=1..k}}_{P} \tilde{s}'$.

The locality of AMDL programs manifests itself in the ability to compute the substate transition relation, $\xrightarrow{(p,c)/(p_i,c_i)_{i=1..k}}_{\mathsf{P}[p,\tilde{p}]}$, directly from the code (without first computing the transition relation and then using projection). This property will be important later to efficiently compute a network-level abstract transformer (Sec. 4.1):

Lemma 2 (2-Locality). *Given* $\tilde{s}[p,\tilde{p}]$ *and* $\tilde{s}[p,\tilde{p}]'$*, checking whether*

$$\tilde{s}[p,\tilde{p}] \xrightarrow{(p,c)/(p_i,c_i)_{i=1..k}}_{P[p,\tilde{p}]} \tilde{s}[p,\tilde{p}]'$$

can be done in time linear in the size of the middlebox program.

3 Network Semantics

This section defines the semantics of stateful networks by defining the semantics of packet traversal over communication channels in the network, and the transitions between network configurations. We first define a concrete semantics, followed by two relaxations: unordered semantics and reverting semantics. These relaxations provide sufficient conditions for completeness of the abstract interpretation performed in Sec. 4.

Fig. 12 provides a high-level view of the different network semantics.

Network Topology. A *network* N is a finite bidirected[8] graph of *hosts* and *middleboxes*, equipped with a *packet domain*. Formally, $\mathsf{N} = (H \cup M, E, P)$, where:

- P is a set of packets.
- H is a finite set of *hosts*. A *host* $h \in H$ consists of a unique identifier and a set of packets $P_h \subseteq P$ that it can send.
- M is a finite set of *middleboxes*. A middlebox $m \in M$ is associated with a set of communication channels C_m.
- $E \subseteq \{\langle h, c_m, m\rangle, \langle m, c_m, h\rangle \mid h \in H, m \in M, c_m \in C_m\} \cup \{\langle m_1, c_{m_1}, c_{m_2}, m_2\rangle \mid m_1, m_2 \in M, c_{m_1} \in C_{m_1}, c_{m_2} \in C_{m_2}\}$ is the set of directed communication channels in the network, each connecting a communication channel $c_{m_1} \in C_{m_1}$ of middlebox m_1 either to a host, or to a communication channel $c_{m_2} \in C_{m_2}$ of middlebox m_2. For e of the form $\langle m, c_m, h\rangle$ or $\langle m, c_m, c_{m_2}, m_2\rangle$, we say that e is an *egress* channel of middlebox m connected to channel c_m and an *ingress* channel of host h, respectively middlebox m_2, connected to channel c_{m_2}.

The network semantics is parametric in the middlebox semantics. It considers the semantics of a middlebox $m \in M$ to be a transition system with a finite set of states $\Sigma[m]$, an initial state $\sigma_I(m) \in \Sigma[m]$ and a set of transitions $\xrightarrow{(p,c)/(p_i,c_i)_{i=1..k}} \subseteq \Sigma[m] \times \Sigma[m]$. This can be realized with either the relation effect semantics or the packet effect semantics defined in Sec. 2.2 and Sec. 2.3, respectively.

[8] A *bidirected graph* is a directed graph in which every edge has a matching edge in the opposite direction. i.e., $(u,v) \in E \iff (v,u) \in E$.

3.1 Concrete (Ordered) Network Configurations

All variants of the network semantics defined in this section are defined over the same set of configurations. Let $\Sigma[M] \stackrel{\text{def}}{=} \bigcup_{m \in M} \Sigma[m]$ denote the set of middlebox states of all middleboxes in a network. An *ordered network configuration* $(\sigma, \pi) \in \Sigma = (M \to \Sigma[M]) \times (E \to P^*)$ assigns middleboxes to their (local) middlebox states and communication channels to sequences of packets. The sequence of packets on each channel represents all packets sent from the source and not yet processed by the destination.

Initial Configuration. We denote the ordered initial configuration by $(\sigma_I, \lambda\, e \in E\,.\,\epsilon)$, where $\sigma_I \colon M \to \Sigma[M]$ denotes the initial state of all middleboxes.

Error Configurations. We say that a configuration is an *error configuration* if any of its middleboxes is in the error state. We denote all error configurations by *err*.

3.2 Concrete (FIFO) Network Semantics

We first consider the First-In-First-Out (FIFO) network semantics, under which communication channels retain the order in which packets were sent.

Ordered Network Transitions. The network semantics is defined via *middlebox transitions* and *host transitions*.

A middlebox transition is $(\sigma, \pi) \xrightarrow{p,e,m}_{\text{o}} (\sigma', \pi')$ where the following holds: (i) p is the *first* packet on the channel $e \in E$, (ii) the channel e is an ingress channel of middlebox m connected to channel $c \in C_m$, (iii) $\sigma(m) \xrightarrow{(p,c)/(p_i,c_i)_{i=1..k}} \sigma'(m)$, meaning that $\sigma'(m)$ is the result of updating $\sigma(m)$ according to the middlebox semantics, (iv) the channels e_i are egress channels of middlebox m connected to the channels $c_i \in C_m$, (v) π' is the result of removing packet p from (the head of) channel e and appending p_i to the tails of the appropriate channels e_i, and (vi) the states of all other middleboxes equal their states in σ.

A host transition is $(\sigma, \pi) \xrightarrow{h,e,p}_{\text{o}} (\sigma, \pi')$ where one of the following holds:

Packet Production (i) the channel e is an egress channel of host h, (ii) $p \in P_h$ is a packet sent by h, and (iii) π' is the result of appending p to the tail of e; or

Packet Consumption (i) the channel e is an ingress channel of host h, (ii) p is the first packet on the channel e, and (iii) π' is the result of removing p from the head of e.

We denote the ordered transition relation obtained by the union of all middlebox and host transitions by $\Rightarrow_{\text{o}}$. It is naturally lifted to a concrete transformer $\mathcal{T}^o \colon \wp(\Sigma) \to \wp(\Sigma)$ defined as:

$$\mathcal{T}^o(X) \stackrel{\text{def}}{=} \{(\sigma', \pi') \mid (\sigma, \pi) \in X \wedge (\sigma, \pi) \Rightarrow_{\text{o}} (\sigma', \pi')\} \ .$$

Collecting Semantics. The ordered collecting semantics of a network N is the set of configurations reachable from the initial configuration.

$$[\![\mathsf{N}]\!]^o \stackrel{\text{def}}{=} \mathit{LeastFixpoint}(\mathcal{T}^o)(\sigma_I, \lambda\, e \in E\,.\,\epsilon) = \bigcup_{i=1}^{\infty} (\mathcal{T}^o)^i(\sigma_I, \lambda\, e \in E\,.\,\epsilon) \ .$$

Definition 4 (Safety Verification Problem). *For a network* N *and initial state* σ_I *for the middleboxes, the safety verification problem is to determine whether an error configuration is reachable from the initial configuration. That is, whether* $err \in [\![\mathsf{N}]\!]^o$.

Theorem 1. *[34] The safety verification problem for ordered networks is undecidable.*

In this work, we tackle the undecidability of verification by developing a sound abstract interpretation that can be used to check the safety of networks. Before doing so, we present two relaxed network semantics that motivate the abstractions we employ, and also provide sufficient conditions for their completeness.

3.3 Unordered and Reverting Network Semantics

The "unordered" semantics allows channels to not preserve the packet transmission order. Namely, packets in the same channel may be processed in a different order than the order in which they were received. The "reverting" semantics allows middleboxes to revert to their initial state after every transition. Formally, these relaxed semantics extend the set of network transitions (and consequently, the transformer and the collecting semantics) with reordering transitions and reverting transitions, respectively.

A *reordering transition* has the form $(\sigma, \pi) \overset{e}{\Rightarrow} (\sigma, \pi')$ where for the channel $e \in E$, $\pi'(e)$ is a permutation of $\pi(e)$ and for all other channels $e' \neq e$, $\pi'(e') = \pi(e')$.

A *reverting transition* has the form $(\sigma, \pi) \overset{m}{\Rightarrow} (\sigma', \pi)$ where for the middlebox $m \in M$, $\sigma'(m) = \sigma_I(m)$ and for all other middleboxes $m' \neq m$, $\sigma'(m) = \sigma(m)$.

The *unordered network transitions* consist of the ordered transitions as well as the reordering transitions; the *ordered reverting transitions* consist of the ordered transitions and the reverting transitions; and the *unordered reverting transitions* consist of all of the above. We denote the corresponding collecting semantics by $[\![\mathsf{N}]\!]^u$, $[\![\mathsf{N}]\!]^{or}$ and $[\![\mathsf{N}]\!]^{ur}$, respectively. Clearly,

$$[\![\mathsf{N}]\!]^o \subseteq [\![\mathsf{N}]\!]^u \subseteq [\![\mathsf{N}]\!]^{ur} \text{ and } [\![\mathsf{N}]\!]^o \subseteq [\![\mathsf{N}]\!]^{or} \subseteq [\![\mathsf{N}]\!]^{ur}$$

By plugging-in the two representations of middleboxes in the definition of the network semantics, we obtain two variants of the network semantics for each of the four variants considered so far. In the sequel, we use a pa subscript to refer to the packet effect semantics, and no subscript to refer to the relation effect semantics. The bisimulation between middlebox representations is lifted to a bisimulation between each relation state network semantics and the corresponding packet state network semantics. Therefore, the following holds:

Lemma 3. *For every semantic identifier* $i \in \{o, u, or, ur\}$, $err \in [\![\mathsf{N}]\!]^i$ *iff* $err \in [\![\mathsf{N}]\!]^i_{pa}$.

The safety verification problem is adapted for the different variants of the network semantics. The following theorem summarizes the complexity of the obtained problems. (We do not distinguish the packet effect semantics from the relation effect semantics, since due to Lem. 3 they induce the same safety verification problem.)

Theorem 2. *The safety verification problem is*

(i) EXPSACE-complete for unordered networks [34].
(ii) undecidable for ordered reverting networks (App. B).
(iii) coNP-hard for unordered reverting networks (App. B).

Thm. 2(ii) justifies the need for the unordered abstraction even in reverting networks. Thm. 2(iii) implies that our abstract interpretation algorithm, presented in Sec. 4, which is both sound and complete for the unordered reverting semantics, is essentially optimal since it essentially meets the lower bound stated in the theorem (it is exponential in the number of state queries of any middlebox and polynomial in the number of middleboxes, hosts and packets).

Sticky Properties. Unordered reverting networks have a useful property of *sticky packets*, meaning that if a packet is pending for a middlebox in some run of the network then any run has an extension in which the packet is pending again with multiplicity $> n$, for any $n \in \mathbb{N}$. This property implies a stronger property:

Lemma 4 (Sticky Packet States Property). *For every channel e, packets $p, \tilde{p}$, middlebox m and packet state $\tilde{v}$ of $\tilde{p}$ in m: If, in some reachable configuration, channel e contains p and in some (possibly other) reachable configuration the packet state of $\tilde{p}$ in m is $\tilde{v}$, then there exists a reachable configuration where simultaneously e contains p and the packet state of $\tilde{p}$ in m is $\tilde{v}$.*

Intuitively, Lem. 4 follows from the fact that all middleboxes can revert to their initial state and the unordered semantics enables a scenario where the particular state and packets are reconstructed. It ensures that ignoring the correlation between the packet states of a middlebox for different packets, the packet states across different middleboxes, and the occurrence (and cardinality) of packets on channels does not incur any precision loss w.r.t. safety. This makes the network-level abstraction defined in Sec. 4, which treats channels as sets of packets and ignores correlations between packet states and channels, precise.

4 Abstract Interpretation for Stateful Networks

In this section, we present our algorithm for safety verification of stateful networks based on abstract interpretation of the semantics $[\![\mathsf{N}]\!]^o_{pa}$, and discuss its guarantees.

4.1 Abstract Interpretation for Packet Space

We apply sound abstractions to different components of the concrete packet state network domain. Due to space constraints, we do not describe the intermediate steps in the construction of the abstract domain, and only present the final domain used by the analysis. Roughly speaking, the obtained domain abstracts away (i) the order and cardinality of packets on channels; (ii) the correlation between the states of different middleboxes and different channel contents; and (iii) the correlation between states of different packets within each middlebox.

Cartesian Packet Effect Abstract Domain. Let $Q \to \{T, F\}$ denote the union of $Q(m) \to \{T, F\}$ over all middleboxes $m \in M$, including the error state *err*. The

Cartesian abstract domain of the packet state of the network is given by the lattice $\mathcal{A} \stackrel{\text{def}}{=} (A, \bot, \sqsubseteq, \sqcup)$, where $A \stackrel{\text{def}}{=} (M \to P \to \wp(Q \to \{T, F\})) \times (E \to \wp(P))$. That is, an abstract element maps each packet in each middlebox to a set of possible valuations for the queries, and each channel to a set of packets. The bottom element is $\bot \stackrel{\text{def}}{=} (\lambda m.\ \lambda p.\ \emptyset, \lambda e.\ \emptyset)$, the partial order $a_1 \sqsubseteq a_2$ is defined by pointwise set inclusions per middlebox and channel, and join is defined by pointwise unions $(\omega_1, \omega_2) \sqcup (\omega_1', \omega_2') \stackrel{\text{def}}{=} (\lambda m.\ \lambda p.\ \omega_1(m)(p) \cup \omega_1'(m)(p), \lambda e.\ \omega_2(p) \cup \omega_2'(p))$.

Let $\mathcal{C} \stackrel{\text{def}}{=} (\wp(\Sigma^{\text{P}}), \subseteq)$ be the concrete network domain. We define the Galois connection $(\mathcal{C}, \gamma, \alpha, \mathcal{A})$ as follows. The abstraction function $\alpha : \wp(\Sigma^{\text{P}}) \to A$ for a set of packet state configurations $X \subseteq \Sigma^{\text{P}}$ is defined as $\alpha(X) = (\omega_{\mathit{mboxes}}, \omega_{\mathit{chans}})$ where

$$\omega_{\mathit{mboxes}} = \lambda m.\ \lambda p.\ \{\sigma(m)(p) \mid (\sigma, \pi) \in X\} \text{ and } \qquad \omega_{\mathit{chans}} = \lambda e.\ \bigcup_{(\sigma,\pi)\in X} \pi(e)\ .$$

The concretization function $\gamma : A \to \wp(\Sigma^{\text{P}})$ is induced by α and $\sqsubseteq$. We denote the initial abstract element as $a_I = \alpha(\{(\sigma_I, \lambda\, e \in E\,.\,\emptyset)\})$.

Abstract Transformer. Next, we define the abstract transformer $\mathcal{T}^\sharp : A \to A$, which soundly abstracts the concrete transformer $\mathcal{T}^o$ and show that it is efficient, due to the locality property of middlebox transitions. We use the predicate $\mathit{in}(c, e, m)$ to denote that the network channel e is an ingress channel of middlebox m, connected to its c channel. Similarly, $\mathit{out}(c, e, m)$ means that e is an egress channel of m connected to its c channel. Further, let $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ denote a mapping from each x_i to y_i for $i = 1..n$ and $f[x \mapsto y]$ denote the function f updated by (re-)mapping x to y.

Definition 5. *Let $(\omega_1, \omega_2) \in (M \to P \to \wp(Q \to \{T, F\})) \times (E \to \wp(P))$ be an abstract element. Then $\mathcal{T}^\sharp(\omega_1, \omega_2) \stackrel{\text{def}}{=}$*

$$\bigsqcup \left\{ \begin{array}{l} (\omega_1[m \mapsto \tilde{ps}], \\ \ \omega_2[e_i \mapsto \omega_2(e_i) \cup \{p_i\}]) \end{array} \middle| \begin{array}{l} (1)\ m \in M, \\ (2)\ p \in \omega_2(e), \mathit{in}(c, e, m), \\ (3)\ \tilde{s} \in \omega_1(m),\ \tilde{p} \in P, \\ \quad \tilde{s}[p, \tilde{p}] = [p \mapsto \tilde{s}(p), \tilde{p} \mapsto \tilde{s}(\tilde{p})], \\ (4)\ \tilde{s}[p, \tilde{p}] \xrightarrow{(p,c)/(p_i,c_i)_{i=1..k}}_{P[p,\tilde{p}]} \tilde{s}[p, \tilde{p}]', \\ (5)\ \tilde{ps} = \tilde{s}[\tilde{p} \mapsto \{\ \tilde{s}[p, \tilde{p}]'(\tilde{p})\ \}], \\ (6)\ \mathit{out}(c_i, e_i, m), i = 1..k \end{array} \right\}\ .$$

Intuitively, the transformer updates the abstract state by joining the individual effects obtained by: (1) considering each middlebox, (2) considering each input packet to the middlebox, (3) considering every possible substate for the input packet p and every other packet $\tilde{p}$, (4) considering every possible substate transition, (5) adding the new packet state for $\tilde{p}$ to the relevant set, and (6) adding each output packet to the corresponding edge.

Proposition 1. *The running time of $\mathcal{T}^\sharp$ is $O((|M|+|E|)\cdot|P|^2 \cdot 2^{2|Q_{max}|})$, where Q_{max} denotes the maximal set of queries $Q(m)$ over all middleboxes $m \in M$.*

Our algorithm for safety verification computes $\mu^\sharp \stackrel{\text{def}}{=} \mathit{LeastFixpoint}(\mathcal{T}^\sharp)(a_I) = \bigsqcup_{i=1}^{\infty} \mathcal{T}^{\sharp i}(a_I)$ and checks whether $\mathit{err} \in \mu^\sharp$.

Complexity of Least Fixpoint Computation. The height of the abstract domain lattice is determined by the number of packets that can be added to the channels of the network—($|P| \cdot |E|$), multiplied by the number of state changes that can occur in any of the middleboxes—$O(|M| \cdot |P| \cdot 2^{|Q|})$. The time complexity of the abstract interpretation is bounded by the height of the abstract domain lattice multiplied by the time complexity of the abstract transformer:

$$O(|P|^4 \cdot |E| \cdot |M| \cdot 2^{3|Q_{max}|} \cdot (|M| + |E|)) \ .$$

4.2 Soundness and Completeness

Our algorithm is sound in the sense that it never misses an error state. This follows from the use of a sound abstract interpretation:

Theorem 3 (Soundness). $[\![\mathsf{N}]\!]^{o}_{pa} \subseteq [\![\mathsf{N}]\!]^{ur}_{pa} \subseteq \gamma(\mu^\sharp)$.

Our algorithm is also complete relative to the reverting unordered semantics.

Theorem 4 (Completeness). $\mu^\sharp \sqsubseteq \alpha([\![\mathsf{N}]\!]^{ur}_{pa})$.

The proof of Thm. 4 relies on the sticky property formalized by Lem. 4. The theorem states that for reverting unordered networks $\mu^\sharp$ is at least as precise as applying the abstraction function on the concrete packet state network semantics. In particular, this implies that if $\mu^\sharp$ is an abstract error element then *err* $\in [\![\mathsf{N}]\!]^{ur}_{pa}$. As a result, for such networks our algorithm is a decision procedure. For other networks it may produce false alarms, if safety is not maintained by an unordered reverting abstraction.

Properties. Recall that we express safety properties via middleboxes in the network. Therefore, in unordered reverting networks, the possibility to revert applies to the safety property as well, and may introduce false alarms due to addition of behaviors leading to error. However, for safety properties such as isolation which are suffix-closed (i.e., all the suffixes of a safe run are themselves safe runs), this cannot happen (Appendix A).

5 Implementation and Initial Evaluation

In this section, we describe our implementation of the analysis described in Sec. 4, and report our initial experience running the algorithm on a few example networks.

Implementation. We have developed a compiler, `amdlc`, which takes as input a network topology and its initial state (given in `json` format) and AMDL programs for the middleboxes that appear in the topology. The compiler outputs a Datalog program, which can then be efficiently solved by a Datalog solver. Specifically, we use LogicBlox [2].

The generated Datalog programs include three relations: (i) `packetsSeen`, which stores the packets sent over the network channels; (ii) `middleboxState`, which stores the packet state of individual packets in each middlebox (i.e., the possible valuation of each middlebox program's queries for each individual packet); and (iii) `abort`, which stores the middleboxes that have reached an *err* state.

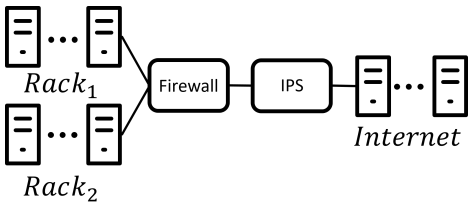


Fig. 4: Topology of the datacenter example.

We encode the packets that hosts can send to their neighboring middleboxes and the initial state of the middleboxes as Datalog *facts* (edb), and the effects of the middlebox programs, i.e. relation update actions and packet output actions, as Datalog *rules* (idb).

We then use the datalog engine to compute the fixed point of the datalog program. That fixed point is exactly the least fixed point $\mu^\sharp \stackrel{\text{def}}{=} \mathit{LeastFixpoint}(\mathcal{T}^\sharp)(a_I) = \bigsqcup_{i=1}^{\infty} \mathcal{T}^{\sharp^i}(a_I)$

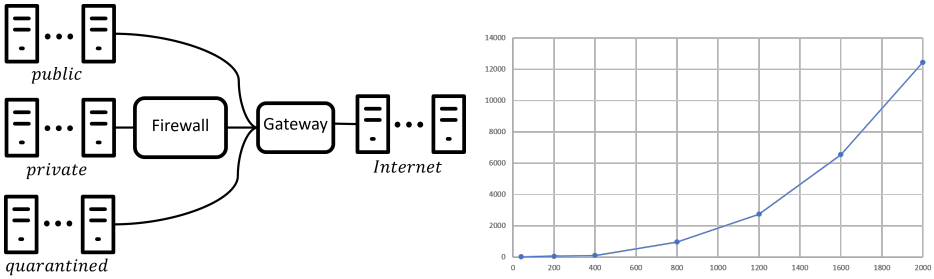
Evaluation. The main challenge in acquiring realistic benchmarks is that middlebox configuration and network topology are considered security sensitive, and as a result enterprises and network operators do not release this information to the public. Consequently, we benchmarked our tool using the synthetic topologies and configurations described by [23].

Our benchmarks focus on datacenter networks and enterprise networks. The set of middleboxes we used in our datacenter benchmarks is based on information provided in [26], and on conversations with datacenter providers. We ran both a simple case where each tenant machine is protected by firewalls and an IPS (Intrusion Prevention System); and a more complex case where we use redundant servers and distribute traffic across them using a load balancer. Our enterprise topology is based on the standard topology used in a variety of university departments including UIUC (reported in [17]), UC Berkeley, Stanford, etc. which employ firewalls and an IP gateway.

We ran two scaling experiments, measuring how well our system scales when the number of hosts or the number of middleboxes in the network increases The experiments were run on Amazon EC2 r4.16 instances with 64-core CPUs and 488GiB RAM.

Multi Tenant Datacenter Network. Fig. 4 illustrates the topology of a multi tenant datacenter. Each rack hosts a different tenant, and the safety property we wish to verify is isolation between the hosts of the two racks. In this example the network also employs an IPS to prevent malicious traffic from reaching the datacenter. Actual IPS code is too complex to be accurately modeled in AMDL; instead we over-approximate the behaviour of an IPS by modeling it as a process that non-deterministically drops incoming packets.

Enterprise Network. Fig. 5a illustrates the topology of an enterprise network. The enterprise network consists of three subnets, each with a different security policy. The *public* subnet is allowed unrestricted access with the outside network. The *quarantined* subnet is not allowed any communication with the outside network. The *private* subnet



(a) Enterprise (b) Running time (seconds).

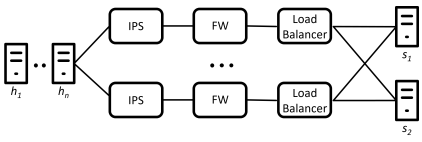
Fig. 5: Topology and running times of the host scalability test.

can initiate communication with a host in the outside network, but hosts in the outside network cannot initiate communication with the hosts in the *private* subnet.

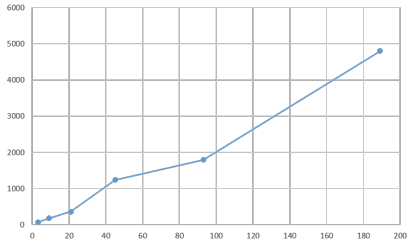
To evaluate the feasibility of our solution, we ran the analysis of Fig. 5a on networks with varying numbers of hosts ranging from 20 to 2,000. Our implementation successfully verified a network with 2,000 hosts in under four hours, suggesting that the implementation could be used to verify realistic networks. Fig. 5b shows the times of the analysis on an enterprise network with 20–2,000 hosts.

Datacenter Middlebox Pipeline. Fig. 6a describes a datacenter topology with a pipeline of middleboxes connecting servers to the Internet. The topology contains multiple middlebox pipelines for load-balancing purposes and to ensure resiliency. We use this topology to test the scalability of our approach w.r.t the size of the network, by adding additional middlebox pipelines and keeping the number of hosts constant.

Fig. 6b shows the running times of the analysis of a datacenter with 3–189 middleboxes (1–32 middlebox chains). All topologies contained 1000 hosts.



(a) Topology with multiple middlebox-pipelines



(b) Running time (seconds).

Fig. 6: Topology and running times of the network topology scalability test.

6 Concluding Remarks and Related Work

In this paper, we applied abstract interpretation for efficient verification of networks with stateful nodes. We now briefly survey closely related works in this area.

Topology Independent Network Verification. Early work in network verification focused on proving correctness of network protocols [5,27]. Subsequent work in the context of software define networking (SDN) including Flowlog [22] and VeriCon [3] looked at verifying the correctness of network applications (implemented as middleboxes or in network controllers) independent of the topology and configuration of the network where these were used. However, since this problem is undecidable, these methods use bounded model checking or user provided inductive invariants, which are hard to specify even in simple network topologies.

Verifying Immutable Network Configurations. Verifying networks with immutable states is an active line of research [17,13,15,4,14,32,29,1,11]. In the future, we hope to combine our abstraction with the techniques used in these papers. We hope to use similar techniques to Veriflow [15] to handle switches more efficiently, and leverage compact header representation described in NetKat [11].

Stateful Network Verification. Previous works provide useful tools for detecting errors in firewalls [19,18,21]. Buzz [8] and SymNet [33] have looked at how to use symbolic execution and packet generation for testing and verifying the behavior of stateful networks. These works implement testing techniques rather than verifying network behavior and are hence complementary to our approach.

Velner et al. [34] show that checking safety in stateful networks is undecidable, necessitating the use of overapproximations. They provide a general algorithm for checking safety using Petri nets. This algorithm has high complexity and scales poorly. They also provide an efficient algorithm for checking safety in a limited class of networks.

Exploring Network Symmetry. Recent work explored the use of bisimulation to leverage the extensive symmetry found in real network topologies [20] to accelerate stateless [24] and stateful [23] network verification. Both approaches are not automatic. We are encouraged by the fact that our automatic approach achieves performance comparable to VMN [23] on the same examples without requiring human intervention. We attribute this improvement to modularity and to the use of packet state representation.

Extensible Semantics. Previous works have explored ideas similar to the reverting semantics, to obtain complexity and decidability results in different settings.

In [7] the authors analyze the complexity of verifying asynchronous shared-memory systems. They use *copycat* processes that mirror the behaviour of another process to show that executions are extensible, similarly to how our work uses the sticky packet states property (Lem. 4). In their model, when the processes are finite state machines, they obtain coNP-complete complexity for verification.

In [9] the authors explore a more general setting of well-structured transition system, and present the *home-state idea*, which allows the system to return to its initial state (essentially, revert). They obtain decidability results for well-structured transition systems with a home-state, but do not show any tighter complexity results.

Acknowledgments We thank our anonymous shepherd, and anonymous referees for insightful comments which improved this paper. We thank LogicBlox for providing us with an academic license for their software, and Todd J. Green and Martin Bravenboer for providing technical support and helping with optimization. This publication is part of projects that have received funding from the European Research Council (ERC) under the European Union's Seventh Framework Program (FP7/2007–2013) / ERC grant agreement no. [321174-VSSC], and Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). The research was supported in part by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University, and the Pazy Foundation. This material is based upon work supported by the United States-Israel Binational Science Foundation (BSF) grants No. 2016260 and 2012259. This research was also supported in part by NSF grants 1704941 and 1420064, and funding provided by Intel Corporation.

References

1. C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *POPL*, 2014.
2. M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logicblox system. In *ACM SIGMOD International Conference on Management of Data*, pages 1371–1382, 2015.
3. T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, page 31, 2014.
4. M. Canini, D. Venzano, P. Peres, D. Kostic, and J. Rexford. A nice way to test openflow applications. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, 2012.
5. E. M. Clarke, S. Jha, and W. R. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods (PROCOMET '98) 8-12 June 1998, Shelter Island, New York, USA*, pages 87–106, 1998.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979.
7. J. Esparza, P. Ganty, and R. Majumdar. Parameterized verification of asynchronous shared-memory systems. In *International Conference on Computer Aided Verification*, pages 124–140. Springer, 2013.
8. S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, V. Sekar, S. Vyas, and Cmu. Buzz: Testing context-dependent policies in stateful networks buzz: Testing context-dependent policies in stateful networks. In *NSDI*, 2016.
9. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.
10. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.
11. N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for netkat. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, pages 343–355, 2015.

12. J. Hoenicke, R. Majumdar, and A. Podelski. Thread modularity at many levels: a pearl in compositional verification. In *POPL*, pages 473–485, 2017.
13. P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, 2013.
14. P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, 2012.
15. A. Khurshid, W. Zhou, M. Caesar, and B. Godfrey. Veriflow: verifying network-wide invariants in real time. *Computer Communication Review*, 42(4):467–472, 2012.
16. M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A soft way for openflow switch interoperability testing. In *CoNEXT*, pages 265–276, 2012.
17. H. Mai, A. Khurshid, R. Agarwal, M. Caesar, B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *SIGCOMM*, 2011.
18. R. M. Marmorstein and P. Kearns. A tool for automated iptables firewall analysis. In *Usenix annual technical conference, Freenix Track*, pages 71–81, 2005.
19. A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 177–187. IEEE, 2000.
20. K. S. Namjoshi and R. J. Trefler. Uncovering symmetries in irregular process networks. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 496–514, 2013.
21. T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.
22. T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 519–531, 2014.
23. A. Panda, O. Lahav, K. J. Argyraki, M. Sagiv, and S. Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 699–718, 2017.
24. G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling network verification using symmetry and surgery. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 69–83, 2016.
25. A. Pnueli, J. Xu, and L. Zuck. Liveness with (0, 1, infinity)-counter abstraction. In *Computer Aided Verification*, pages 93–111. Springer, 2002.
26. R. Potharaju and N. Jain. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013*, pages 9–22, 2013.
27. R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Security and Privacy*, 2000.
28. A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60(2):177–229, 1988.
29. D. Sethi, S. Narayana, and S. Malik. Abstractions for model checking sdn controllers. In *FMCAD*, 2013.

30. J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *SIGCOMM*, 2012.
31. A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 15–28, 2016.
32. R. Skowyra, A. Lapets, A. Bestavros, and A. Kfoury. A verification platform for sdn-enabled applications. In *HiCoNS*, 2013.
33. R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Scalable symbolic execution for modern networks. In *SIGCOMM*, 2016.
34. Y. Velner, K. Alpernas, A. Panda, A. Rabinovich, M. Sagiv, S. Shenker, and S. Shoham. Some complexity results for stateful network verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 811–830. Springer, 2016.

A Reverting Safety Properties

Recall that we express safety properties via middleboxes in the network. Therefore, in unordered reverting networks, the possibility to revert applies to the safety property as well. As the reverting semantics adds transitions, this may increase the possible set of transitions of the safety middleboxes, and, in particular, may add transitions into an error state. For some temporal safety properties this is a source of imprecision as they cannot be precisely captured by the reverting semantics, thus introducing false alarms.

For example, if the safety property forbids a packet from host h_{ext} to host h_{in} before a packet from host h_{in} has been sent to h_{ext}, then in a reverting network, even if a packet from host h_{in} has been previously sent to h_{ext}, a revert transition allows the middlebox to return to its initial state, from which a packet from host h_{ext} to host h_{in} leads to an error state.

However, we identify a class of safety middleboxes that is guaranteed not to be a source of imprecision. This class includes any stateless safety middlebox, and in particular isolation middleboxes, More generally, we provide a sufficient condition for a safety property to be precisely expressible in a reverting network. To do so, we first decouple the enforcement of safety from the forwarding behavior of the network. For this decoupling, in the sequel we consider safety middleboxes with a single output port that forward any incoming packet (on any input port) to the output port without any modification. This ensures that safety middleboxes do not affect the forwarding behavior of the network. In particular, the forwarding behavior of safety middleboxes does not depend on their state. The state is only used to enforce safety. For such safety middleboxes we define:

Definition 6. *A safety middlebox m is* revert-robust *if for every sequence of input packets $in = (p_i, c_i)_{i=1..k}$, if no execution of m on in, starting from m's initial state, leads to err, then for every suffix in' of in, no execution of m on in' starting from m's initial state leads to err as well.*

Intuitively, revert-robustness means that the language of "safe" sequences of packets is suffix-closed. In particular, any stateless safety middlebox (such as an isolation middleboxes) is revert-robust. For example, if the safety middlebox forbids a packet from host h_{ext} to host h_{in} after a packet from host h_{in} has been sent to h_{ext}, then it is revert-robust. The reason is that, in this example, the "safe" input sequences are ones where no packet from host h_{ext} to host h_{in} has a preceding packet from host h_{in} to h_{ext}. Therefore any suffix of a safe input sequence is also safe. As a result, such a safety middlebox will not introduce false alarms in a reverting network, as reverting transitions will just make the middlebox "forget" the prefix of the sequence. (Note that it will also not make the network wrongfully safe, as safety requires that all executions, including the ones that do not use revert transitions, are safe.) Next, we claim that revert-robustness is a sufficient condition for not losing precision of the analysis (i.e., not introducing false alarms) due to the revert transitions of the safety middlebox. In order to formalize this claim, we need the following definitions. For a network N with a set of middleboxes M, a subset $S \subseteq M$, and a semantic identifier $i \in \{o, u, or, ur\}$, we denote by $[\![\mathsf{N}]\!]_{pa}^{i \setminus S}$ the corresponding network collecting semantics, with the exception that no reverting transitions are applied to the middleboxes in S (when applicable). We then have:

Lemma 5. *Let* N *be a network such that all of its safety middleboxes,* $S \subseteq M$*, are revert-robust. Then for every* $i \in \{o, u, or, ur\}$*,* $err \in [\![\mathsf{N}]\!]_{pa}^{i \setminus S}$ *if and only if* $err \in [\![\mathsf{N}]\!]_{pa}^{i}$*, where* $[\![\mathsf{N}]\!]_{pa}^{i \setminus S}$ *is the same as* $[\![\mathsf{N}]\!]_{pa}^{i}$*, except that no reverting transitions are applied to the middleboxes in* S*.*

This means that the network is safe (under any of the semantics) if and only if it is safe with the same semantics except that all safety middleboxes are non-reverting.

Proof. The direction from left to right is trivial, as the reverting semantics is a sound approximation, hence a computation leading to error when S is non-reverting also exists when S is reverting. In order to prove the converse direction we denote by N the network where all middleboxes including S may revert and by N' the network where S may not revert. We prove that if all the computations of N' are safe then so are the computations of N. The proof is straightforward. We observe that for every scenario s in N there is a corresponding scenario in N' which is identical to s other than the behavior of the safety middleboxes (this is because safety middleboxes do not affect forwarding of packets). Consider a safety middlebox m and an arbitrary step i in the scenario. Let $p_1, \ldots, p_\ell$ be the sequence of packets that m processed until step i and let $p_r, \ldots, p_\ell$ be the packets it processed since it was last reverted. Since N' is safe, it follows that in N' the middlebox m is not in *err*. As m is revert-robust and $p_r, \ldots, p_\ell$ is a suffix of $p_1, \ldots, p_\ell$, then m is also not in *err* state in N. Thus, we get that for every s, i and m, the middlebox m is not in *err* state. Hence, N is safe and the proof is complete.

B Proofs

In this section, we include proofs for some of the key claims made in the paper.

Proof (Proof of Thm. 2 (Undecidability)). It is well known that an automaton with an ordered channel of messages (also known as a *channel machine*) can simulate a Turing machine. The channel can trivially store the content of a Turing machine tape, and the automaton can simulate the transitions of the machine. This can be used to easily show that in the absence of reverting the isolation problem over ordered channels is undecidable even when there is only one host, and one middlebox with a self loop.

When reverting is possible, we add auxiliary packet type and middlebox states. Whenever in initial state, the middlebox sends a special packet over its self loop, and discards all arrived packets until it receives the special packet [9]. This empties the self loop from its content, which intuitively, resets the tape of the Turing machine. Hence, when the middlebox reverts, so does the Turing machine. Thus, the isolation property is violated if and only if the Turing machine reaches an accepting state, and the undecidability proof follows.

Proof (Proof of Thm. 2 (coNP-hardness)). We prove that if the number of queries in a middlebox is not a constant (i.e., it depends on other parameters of the problem), then the safety problem is coNP- hard even when the network consists of only one middlebox

[9] Note that for this step it is crucial that the channels are FIFO.

and one host. The proof is by reduction from the Boolean unsatisfiability problem of propositional formulas.

Given a formula ϕ with n variables $x_1, \ldots, x_n$ we construct a network with one host and one middlebox m, such that m has only one port, connected to h. The packet types are $x_1, \neg x_1, \ldots, x_n, \neg x_n$, i.e., there are $2n$ packet types, one for each literal. The middlebox has two nullary relations, O_i and V_i, for every $i \in \{1, \ldots, n\}$, where intuitively, O_i indicates whether a packet of type x_i or $\neg x_i$ already occurred and V_i indicates if the first such packet is positive (x_i) or negative ($\neg x_i$). That is, the O_i relations indicate which variables are assigned, while the V_i relations store the assignment. Initially all the relations are initialized to False (i.e., no variable is assigned). Upon receiving a packet of type x_i or $\neg x_i$, the middlebox updates the relation V_i only if O_i is False, in which case O_i is also updated to True. If the packet type is x_i, then V_i is updated to True. Otherwise it is updated to False. In addition, whenever the interpretation of O_i and V_i satisfies ϕ, the middlebox aborts. Clearly, the size of the code of m is polynomial and safety is violated if and only if ϕ is satisfiable. We note that possible resets do not affect the safety of the network.

Lemma 6 (Sticky Packets Property). *For every channel e and packet p: If in some reachable configuration e contains p, then every run can be extended such that e will eventually contain p. Moreover, every run can be extended such that e will eventually contain n copies of p (for every $n > 0$).*

Proof (Proof of Lem. 6). The proof relies on the reverting property and on the fact that the channels are unordered.

Let σ_0 be a reachable configuration in which p occurs in e, and let s_0 be the scenario that led to it, i.e., the sequence of events that took place. Consider an arbitrary run (scenario) π. One can extend π with the following scenario: First all the middleboxes return to their initial state. Second, scenario s_0 occur, i.e., only packets from scenario s_0 are processed, and the other packets are ignored. This extension is possible because the channels are unordered.

To construct a scenario in which e contains n copies of p, we just concatenate the above mentioned extension n time.

Lemma 7 (Sticky States Property). *For every channel e, packet p, middlebox m and state s of m: If, in some reachable configuration, channel e contains p and in some (possibly other) reachable configuration m is in state s, then there exists a reachable configuration where simultaneously e contains p and m is in state s.*

Proof (Proof of Lem. 7). Let $(p_1, \ldots, p_\ell)$ be the sequence of packets that m processed from the latest reset until it arrives to state σ_m in the given witness scenario.

Consider an arbitrary run. By Lem. 6 we can extend this run such that $p_1, \ldots, p_\ell$ are pending packets in the ingress channel of middlebox m and p is pending in e (if some of the packets occur more than once in the sequence, then by the same lemma we may assume that there are multiple copies of those packets).

We further extend the run with a reset event for middlebox m. Finally, we extend the scenario such that in the next ℓ steps m will process $p_1, \ldots, p_\ell$ reaching state σ_m.

Proof (Proof of Lem. 4). The proof follows directly from Lem. 1, 6 and 7.

Proof (Proof of Thm. 4). In order to prove completeness it is enough to show that every application of the best abstract transformer results in an abstract value that is less or equal than the result of applying the abstraction function on the concrete least fixed point (i.e., the reachable states of the network w.r.t unordered reverting packet state space semantic). The proof is by induction over n, the number of times we apply the transformer. The proof for $n = 0$ is trivial. For $n > 1$, let $p, \tilde{p}$ and m be packets and a middlebox. By the induction hypothesis for every packet state $v \in \omega_1(m)(\tilde{p})$ there is a concrete reachable middlebox state such that the state of m over packet $\tilde{p}$ is v' and for every packet $p \in \omega'_2(e)$ there is a reachable concrete configuration where p is in e. Hence, by Lem. 4, there exists a concrete reachable configuration in which p is in e and the state of m over packet $\tilde{p}$ is v. Therefore, by definition of ω'_1 and ω'_2, every new state in $\omega'_1(m)(\tilde{p}) \setminus \omega_1(m)(\tilde{p})$ has a corresponding concrete reachable state, and likewise for any new pending packet in $\omega'_2(e) \setminus \omega(2)$. The proof is complete.

Proof (proof of Lem. 5). The direction from left to right is trivial, as the reverting semantics is a sound approximation, hence a computation leading to error when S is non-reverting also exists when S is reverting. In order to prove the converse direction we assume that *err* $\notin [\![\mathsf{N}]\!]^{i\setminus S}_{pa}$ and prove that all the computations of $[\![\mathsf{N}]\!]^{i}_{pa}$ are safe. The proof is straightforward. We observe that for every computation s in $[\![\mathsf{N}]\!]^{i}_{pa}$ there is a corresponding computation in $[\![\mathsf{N}]\!]^{i\setminus S}_{pa}$ which is identical to s other than the behavior of the safety middleboxes (this is because safety middleboxes do not affect forwarding of packets). Consider a safety middlebox m and an arbitrary step k in the computation. Let $p_1, \ldots, p_\ell$ be the sequence of packets that m processed until step i and let $p_r, \ldots, p_\ell$ be the packets it processed since it last reverted. Since *err* $\notin [\![\mathsf{N}]\!]^{i\setminus S}_{pa}$ it follows that in particular the middlebox m is not in *err* state. As m is revert-robust and $p_r, \ldots, p_\ell$ is a suffix of $p_1, \ldots, p_\ell$, then m is also not in *err* state in $[\![\mathsf{N}]\!]^{i}_{pa}$ (where it may revert). Thus, we get that for every s, k and m, the middlebox m is not in *err* state. Hence, *err* $\notin [\![\mathsf{N}]\!]^{i}_{pa}$ and the proof is completed.

C The Semantics of AMDL

In this section, we define two semantics for middleboxes—the one based on relation states and the one based packet states. We then prove that both semantics are bisimilar.

A Note on Field Binding.. A *pblock* construct binds the atoms in a packet received on a channel to field names before executing a guarded commands. We will assume that there is at most one *pblock* construct per incoming channel. This assumption does not impose a restriction, since two *pblock* constructs *ch* **?** $(f_1, \ldots, f_k) \Rightarrow gc_1$ and *ch* **?** $(g_1, \ldots, g_k) \Rightarrow gc_2$ over the same channel *ch* can be automatically merged into a single *pblock* construct via the source-to-source transformation

$$ch \text{ ? } (f_1, \ldots, f_k) \Rightarrow \textbf{if } gc_1 \Box gc_2[f_1/g_1, \ldots, f_k/g_k] \textbf{ fi}$$

where the field names of the second *pblock* construct are substituted appropriately for the field names of the first *pblock* construct. (Technically, the transformation first extends the sequence of atoms of the *pblock* construct with fewer number of atoms by

adding dummy atoms.) This assumption allows us to access the atom a_i of the incoming packet by indexing into the sequence of fields, as f_i.

C.1 Relation State Semantics

We start by defining a big-step semantics for relation states.

Let m be a fixed middlebox.

For simplicity of the presentation, we consider the case where $P \stackrel{\text{def}}{=} (H \times H \times T)$ denotes the set of all packets. (The adaptation to other definitions of the packets space is straightforward.) Let C_m denote the set of channels of m. We define the sequence of pairs of packets and channels to be sent following a transition of the middlebox m on every channel as *Cont* $\stackrel{\text{def}}{=} (P \times C_m)^*$. The semantics of guarded commands, actions, conditions, and atoms is given in the context of a middlebox state $s \in \Sigma[m] = \mathit{rels}(m) \to \wp(D(m))$ and a packet p.

We start by defining in Fig. 7 semantic evaluation functions for atoms and conditions:

$$\mathsf{R}[\![\cdot]\!] : \langle \mathit{atom} \rangle \to P \to (T \cup H)$$
$$\mathsf{R}[\![\cdot]\!] : \langle \mathit{cond} \rangle \to (\Sigma[m] \times P) \to \{\mathsf{True}, \mathsf{False}\}$$

$$\begin{array}{|rl|}
\hline
\mathsf{R}[\![f_i]\!]p \stackrel{\text{def}}{=} a_i & p = (a_1, \ldots, a_k) \\
\mathsf{R}[\![(f_{j_1}, \ldots, f_{j_k})]\!]p \stackrel{\text{def}}{=} (a_{j_1}, \ldots, a_{j_k}) & p = (a_1, \ldots, a_k) \\
\mathsf{R}[\![h]\!]p \stackrel{\text{def}}{=} h & h \in H \\
\mathsf{R}[\![t]\!]p \stackrel{\text{def}}{=} t & t \in T \\
\hline
\mathsf{R}[\![\textbf{true}]\!](s,p) \stackrel{\text{def}}{=} \mathsf{True} & \\
\mathsf{R}[\![\textbf{false}]\!](s,p) \stackrel{\text{def}}{=} \mathsf{False} & \\
\mathsf{R}[\![c_1 \textbf{ and } c_2]\!](s,p) \stackrel{\text{def}}{=} \begin{cases} \mathsf{True}, & \mathsf{R}[\![c_1]\!](s,p) = \mathsf{True} \text{ and } \mathsf{R}[\![c_2]\!](s,p) = \mathsf{True}; \\ \mathsf{False}, & \text{otherwise.} \end{cases} & \\
\mathsf{R}[\![\textbf{not } c]\!](s,p) \stackrel{\text{def}}{=} \begin{cases} \mathsf{False}, & \mathsf{R}[\![c]\!](s,p) = \mathsf{True}; \\ \mathsf{True}, & \text{otherwise.} \end{cases} & \\
\mathsf{R}[\![a_1 = a_2]\!](s,p) \stackrel{\text{def}}{=} \begin{cases} \mathsf{True}, & \mathsf{R}[\![a_1]\!]p = \mathsf{R}[\![a_2]\!]p; \\ \mathsf{False}, & \text{otherwise.} \end{cases} & \\
\mathsf{R}[\![\overline{a} \textbf{ in } r]\!](s,p) \stackrel{\text{def}}{=} \begin{cases} \mathsf{False}, & s = \mathit{err}; \\ \mathsf{True}, & \mathsf{R}[\![\overline{a}]\!]p \in s(r); \\ \mathsf{False}, & \text{otherwise.} \end{cases} & \\
\hline
\end{array}$$

Fig. 7: Semantic evaluation of atoms and conditions.

Fig. 8 defines transition relations for guarded commands, blocks, and middleboxes:

$$\begin{array}{r}
\mathsf{R}[\![\cdot]\!] : \langle \mathit{action} \rangle \to (\Sigma[m] \times P \times \mathit{Cont}) \times (\Sigma[m] \times P \times \mathit{Cont}) \\
\mathsf{R}[\![\cdot]\!] : \langle \mathit{gc} \rangle \to (\Sigma[m] \times P \times \mathit{Cont}) \times (\Sigma[m] \times P \times \mathit{Cont}) \\
\mathsf{R}[\![\cdot]\!] : \langle \mathit{pblock} \rangle \to (\Sigma[m] \times (P \times C_m)) \times (\Sigma[m] \times \mathit{Cont}) \\
\mathsf{R}[\![\cdot]\!] : \langle \mathit{mbox} \rangle \to (\Sigma[m] \times (P \times C_m)) \times (\Sigma[m] \times \mathit{Cont}) \ .
\end{array}$$

A guarded command accepts a middlebox state, an assignment of fields to values, and a mapping from output channels to their output content (i.e., the sequences of packets that should be delivered to them). It returns the updated state, the (same) assignment of fields to values, and the new mapping from channels to content.

A block accepts a middlebox state and a packet on a specified input channel and returns the updated state and the output sent to the output channels. A middlebox nondeterministically chooses between its blocks.

$$\langle ch\,!\,\overline{a}, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} (s, p, send) \qquad s = err$$
$$\langle ch\,!\,\overline{a}, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} (s, p, send \cdot (\mathsf{R}[\![\overline{a}]\!]p, ch)) \qquad s \neq err$$
$$\langle r(\overline{a}) := c, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} (s, p, send) \qquad s = err$$
$$\langle r(\overline{a}) := c, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} (s[r \mapsto s(r) \cup \{\mathsf{R}[\![\overline{a}]\!]p\}], p, send) \quad \mathsf{R}[\![c]\!](s, p) = \mathsf{True}$$
$$\langle r(\overline{a}) := c, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} (s[r \mapsto s(r) \setminus \{\mathsf{R}[\![\overline{a}]\!]p\}], p, send) \quad \mathsf{R}[\![c]\!](s, p) = \mathsf{False}$$
$$\langle \mathbf{abort}, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} (err, p, send)$$

$$\frac{\langle ac_1, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} (s', p, send') \qquad \langle ac_2, (s', p, send')\rangle \longrightarrow_{\mathsf{R}} (s'', p, send'')}{\langle ac_1; ac_2, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} (s'', p, send'')}$$

$$\langle c \Rightarrow ac, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} \mathsf{R}[\![ac]\!](s, p, send) \text{ if } \mathsf{R}[\![c]\!](s, p) = \mathsf{True}$$
$$\langle c \Rightarrow ac, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} (s, p, send) \qquad \text{if } s = err$$

$$\frac{\langle g_i, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} (s', p, send')}{\langle \mathbf{if}\ g_1 \Box \ldots \Box g_n\ \mathbf{fi}, (s, p, send)\rangle \longrightarrow_{\mathsf{R}} (s', p, send')} \quad i \in \{1, \ldots, n\}$$

$$\frac{\langle g, (s, p, \emptyset)\rangle \longrightarrow_{\mathsf{R}} (s', p, send)}{\langle ch\ \mathbf{?}\ (f_1, \ldots, f_k)\ \Rightarrow\ g, (s, (p, ch))\rangle \longrightarrow_{\mathsf{R}} (s', send)} \quad p = (a_1, \ldots, a_k)$$

$$\frac{\langle p_j, (s, (p, ch))\rangle \longrightarrow_{\mathsf{R}} (s', send)}{\langle m = \mathbf{do}\ p_1 \Box \ldots \Box p_n\ \mathbf{od}, (s, (p, ch))\rangle \longrightarrow_{\mathsf{R}} (s', send)} \quad j \in \{1, \ldots, n\}$$

Fig. 8: Derivation rules for atomic actions, guarded commands, blocks, and middleboxes.

C.2 Packet State Semantics

The packet state semantics is defined via the evaluation functions

$$\mathsf{P}[\![\cdot]\!] : \langle atom\rangle \to P \to (T \cup H)$$
$$\mathsf{P}[\![\cdot]\!] : \langle cond\rangle \to (\Sigma^{\mathrm{P}}[m] \times P) \to \{\mathsf{True}, \mathsf{False}\}$$

and the transition relations

$$\begin{aligned}
\mathsf{P}[\![\cdot]\!] &: \langle \mathit{action} \rangle \to (\Sigma^{\mathrm{P}}[m] \times P \times \mathit{Cont}) \times (\Sigma^{\mathrm{P}}[m] \times P \times \mathit{Cont}) \\
\mathsf{P}[\![\cdot]\!] &: \langle \mathit{gc} \rangle \to (\Sigma^{\mathrm{P}}[m] \times P \times \mathit{Cont}) \times (\Sigma^{\mathrm{P}}[m] \times P \times \mathit{Cont}) \\
\mathsf{P}[\![\cdot]\!] &: \langle \mathit{pblock} \rangle \to (\Sigma^{\mathrm{P}}[m] \times (P \times C_m)) \times (\Sigma^{\mathrm{P}}[m] \times \mathit{Cont}) \\
\mathsf{P}[\![\cdot]\!] &: \langle \mathit{mbox} \rangle \to (\Sigma^{\mathrm{P}}[m] \times (P \times C_m)) \times (\Sigma^{\mathrm{P}}[m] \times \mathit{Cont}) \ .
\end{aligned}$$

We define the helper function

$$\mathit{update} : (\Sigma^{\mathrm{P}}[m] \times \mathit{rels}(m) \times \mathit{atoms}^* \times \{\mathsf{True}, \mathsf{False}\}) \to \Sigma^{\mathrm{P}}[m] \ ,$$

which updates a given packet state by adding or removing a given tuple from a given relation, depending on the Boolean value b.

$$\begin{aligned}
&\mathit{update}(s, r, \overline{a}, b) \stackrel{\text{def}}{=} \\
&\lambda \tilde{p} \in P.\ \lambda q \in Q(m).\ \begin{cases} b, & \text{if } \mathit{rel}(q) = \mathrm{r} \wedge \\ & \mathit{atoms}(q)(\tilde{p}) = \overline{a}(p). \\ \tilde{s}(\tilde{p})(q), & \text{otherwise.} \end{cases}
\end{aligned}$$

Fig. 9 shows the evaluation of queries and the derivation rules for updating relations. The rest of the evaluation functions and derivation rules have the same shape as those in Fig. 7 and Fig. 8, replacing $\longrightarrow_{\mathsf{R}}$ with $\longrightarrow_{\mathsf{P}}$ and $\mathsf{R}[\![\cdot]\!]$ with $\mathsf{P}[\![\cdot]\!]$.

$$\mathsf{P}[\![\overline{a}\ \mathbf{in}\ r]\!](s, p) \stackrel{\text{def}}{=} \begin{cases} \mathsf{False}, & s = \mathit{err}; \\ s(p)(\overline{a}\ \mathbf{in}\ r), & \text{otherwise.} \end{cases}$$

$$\begin{aligned}
&\langle r(\overline{a})\ \mathbf{:=}\ c, (s, p, \mathit{send}) \rangle &&\longrightarrow_{\mathsf{P}} (s, p, \mathit{send}) && s = \mathit{err} \\
&\langle r(\overline{a})\ \mathbf{:=}\ c, (s, p, \mathit{send}) \rangle &&\longrightarrow_{\mathsf{P}} (\mathit{update}(s, r, \overline{a}, b), p, \mathit{send}) && b = \mathsf{P}[\![c]\!](s, p)
\end{aligned}$$

Fig. 9: Query evaluation and relation update derivation rule for the packet state semantics.

C.3 Proving Lem. 1

To prove bisimulation, we use induction on the derivation trees. Since the shape of all rules, except the ones shown in Fig. 9, is exactly the same, we only need to demonstrate bisimilarity for them.

Notice that the semantics is strict in *err*—the derivation rules for *err* propagate *err* and query evaluations return False. We therefore, focus only on the cases where the states are different from *err*.

$$
\begin{aligned}
&\mathit{ps}(s[r \mapsto s(r) \cup \{\mathsf{R}[\![\overline{a}]\!]p\}]) \\
&= \lambda\tilde{p} \in P.\ \lambda q \in Q(m).\ \mathit{atoms}(q)(\tilde{p}) \in s[r \mapsto s(r) \cup \{\mathsf{R}[\![\overline{a}]\!]p\}](\mathit{rel}(q)) \\
&= \lambda\tilde{p} \in P.\ \lambda q \in Q(m).\ \mathit{atoms}(q)(\tilde{p}) \in \begin{cases} \{\mathsf{R}[\![\overline{a}]\!]p\}, & \mathit{rel}(q) = r; \\ s(\mathit{rel}(q)), & \text{otherwise.} \end{cases} \\
&= \lambda\tilde{p} \in P.\ \lambda q \in Q(m).\ \begin{cases} \mathit{atoms}(q)(\tilde{p}) \in \{\mathsf{R}[\![\overline{a}]\!]p\}, & \mathit{rel}(q) = r; \\ \mathit{atoms}(q)(\tilde{p}) \in s(\mathit{rel}(q)), & \text{otherwise.} \end{cases} \\
&= \lambda\tilde{p} \in P.\ \lambda q \in Q(m).\ \begin{cases} \mathit{atoms}(q)(\tilde{p}) = \mathsf{R}[\![\overline{a}]\!]p, & \mathit{rel}(q) = r; \\ \mathit{atoms}(q)(\tilde{p}) \in s(\mathit{rel}(q)), & \text{otherwise.} \end{cases} \\
&= \lambda\tilde{p} \in P.\ \lambda q \in Q(m).\ \begin{cases} \mathsf{True}, & \mathit{rel}(q) = r \wedge \mathit{atoms}(q)(\tilde{p}) = \overline{a}(p); \\ \mathit{atoms}(q)(\tilde{p}) \in s(\mathit{rel}(q)), & \text{otherwise.} \end{cases} \\
&= \lambda\tilde{p} \in P.\ \lambda q \in Q(m).\ \begin{cases} \mathsf{True}, & \mathit{rel}(q) = r \wedge \mathit{atoms}(q)(\tilde{p}) = \overline{a}(p); \\ \tilde{s}(\tilde{p})(q), & \text{otherwise.} \quad (\text{using } 1) \end{cases} \\
&= \mathit{update}(\tilde{s}, r, \overline{a}, \mathsf{True})
\end{aligned}
$$

Fig. 10: Detailed proof steps.

Bisimilarity of Query Evaluation

Lemma 8. *If $\tilde{s} \sim_m s$ and $s \neq \mathit{err}$ then the following holds:*

$$\mathsf{P}[\![\overline{a}\ \textbf{\textit{in}}\ r]\!](\tilde{s}, p) = \mathsf{R}[\![\overline{a}\ \textbf{\textit{in}}\ r]\!](s, p)\ .$$

Proof. Recall that $\tilde{s} \sim_m s$ is defined as:

$$\tilde{s} = \lambda\tilde{p} \in P.\ \lambda q \in Q(m).\ \mathit{atoms}(q)(\tilde{p}) \in s(\mathit{rel}(q))\ .$$

Assume $p = (a_1, \ldots, a_k)$ and $\overline{a} = (f_1, \ldots, f_k)$.
Then the following holds:

$$
\begin{aligned}
&\mathsf{P}[\![\overline{a}\ \textbf{in}\ r]\!](\tilde{s}, p) \\
&= \tilde{s}(p)(\overline{a}\ \textbf{in}\ r) \\
&= (\lambda\tilde{p} \in P.\ \lambda q \in Q(m).\ \mathit{atoms}(q)(\tilde{p}) \in s(\mathit{rel}(q)))(p)(\overline{a}\ \textbf{in}\ r) \\
&= (\lambda q \in Q(m).\ \mathit{atoms}(q)(p) \in s(\mathit{rel}(q)))(\overline{a}\ \textbf{in}\ r) \\
&= (a_1, \ldots, a_k) \in s(r) \\
&= \mathsf{R}[\![(f_1, \ldots, f_k)]\!]p \in s(r) \\
&= \mathsf{R}[\![\overline{a}\ \textbf{in}\ r]\!](s, p)\ .
\end{aligned}
$$

Bisimilarity of Relation Updates Assume that $\tilde{s} \sim_m s$ and that $s \neq \mathit{err}$. By the induction hypothesis, we have that $b = \mathsf{P}[\![c]\!](\tilde{s}, p) = \mathsf{R}[\![c]\!](s, p)$ holds.

Assume that $b = \mathsf{True}$. Therefore, the following derivations apply:

$$
\begin{aligned}
&\langle r(\overline{a})\ \textbf{:=}\ c, (s, p, \mathit{send})\rangle \longrightarrow_{\mathsf{R}} (s[r \mapsto s(r) \cup \{\mathsf{R}[\![\overline{a}]\!]p\}], p, \mathit{send}) \\
&\langle r(\overline{a})\ \textbf{:=}\ c, (\tilde{s}, p, \mathit{send})\rangle \longrightarrow_{\mathsf{P}} (\mathit{update}(\tilde{s}, r, \overline{a}, \mathsf{True}), p, \mathit{send})\ .
\end{aligned}
$$

We will use the following identity, which we obtain from the definition of $\tilde{s}$:

$$
\begin{aligned}
&\tilde{s}(p)(q) \\
&= (\lambda\tilde{p} \in P.\ \lambda\tilde{q} \in Q(m).\ \mathit{atoms}(\tilde{q})(\tilde{p}) \in s(\mathit{rel}(\tilde{q})))(p)(q) \qquad (1) \\
&= \mathit{atoms}(q)(p) \in s(\mathit{rel}(q))\ .
\end{aligned}
$$

$$
\begin{aligned}
&ps(s[r \mapsto s(r) \setminus \{\mathsf{R}[\![\overline{a}]\!]p\}]) \\
&= \lambda \tilde{p} \in P.\ \lambda q \in Q(m).\ \mathit{atoms}(q)(\tilde{p}) \in s[r \mapsto s(r) \setminus \{\mathsf{R}[\![\overline{a}]\!]p\}](\mathit{rel}(q)) \\
&= \lambda \tilde{p} \in P.\ \lambda q \in Q(m).\ \begin{cases} \mathit{atoms}(q)(\tilde{p}) \notin \{\mathsf{R}[\![\overline{a}]\!]p\}, & \mathit{rel}(q) = r; \\ \mathit{atoms}(q)(\tilde{p}) \in s(\mathit{rel}(q)), & \text{otherwise.} \end{cases} \\
&= \lambda \tilde{p} \in P.\ \lambda q \in Q(m).\ \begin{cases} \mathit{atoms}(q)(\tilde{p}) \neq \mathsf{R}[\![\overline{a}]\!]p, & \mathit{rel}(q) = r; \\ \mathit{atoms}(q)(\tilde{p}) \in s(\mathit{rel}(q)), & \text{otherwise.} \end{cases} \\
&= \lambda \tilde{p} \in P.\ \lambda q \in Q(m).\ \begin{cases} \mathsf{False}, & \mathit{rel}(q) = r \wedge \mathit{atoms}(q)(\tilde{p}) = \overline{a}(p); \\ \mathit{atoms}(q)(\tilde{p}) \in s(\mathit{rel}(q)), & \text{otherwise.} \end{cases} \\
&= \lambda \tilde{p} \in P.\ \lambda q \in Q(m).\ \begin{cases} \mathsf{False}, & \mathit{rel}(q) = r \wedge \mathit{atoms}(q)(\tilde{p}) = \overline{a}(p); \\ \tilde{s}(\tilde{p})(q), & \text{otherwise.} \quad (\text{using } 1) \end{cases} \\
&= \mathit{update}(\tilde{s}, r, \overline{a}, \mathsf{False})
\end{aligned}
$$

Fig. 11: Detailed proof steps.

We have to show that the following relation holds in Fig. 10:

$$s[r \mapsto s(r) \cup \{\mathsf{R}[\![\overline{a}]\!]p\}] \sim_m \mathit{update}(\tilde{s}, r, \overline{a}, \mathsf{True}) \ .$$

Assume that $b = \mathsf{False}$. Therefore, the following derivations apply:

$$
\begin{aligned}
&\langle r(\overline{a}) := c, (s, p, \mathit{send})\rangle \longrightarrow_{\mathsf{R}} (s[r \mapsto s(r) \setminus \{\mathsf{R}[\![\overline{a}]\!]p\}], p, \mathit{send}) \\
&\langle r(\overline{a}) := c, (\tilde{s}, p, \mathit{send})\rangle \longrightarrow_{\mathsf{P}} (\mathit{update}(\tilde{s}, r, \overline{a}, \mathsf{False}), p, \mathit{send}) \ .
\end{aligned}
$$

We show that the following relation holds in Fig. 11:

$$s[r \mapsto s(r) \setminus \{\mathsf{R}[\![\overline{a}]\!]p\}] \sim_m \mathit{update}(\tilde{s}, r, \overline{a}, \mathsf{False}) \ .$$

D Hierarchy of Abstract Domains

Fig. 12 provides a high-level view of the different network semantics.

E Example

Fig. 13a shows a simple network where two stateful firewalls are connected in a row to prevent traffic between nodes h_2 to h_1. This is an artificial example meant to illustrate the verification process. More realistic examples are presented in Sec. 5. It is assumed that hosts h_1 and h_2 can send and receive arbitrary packets on channels e_1 and e_4, respectively. The example is implemented using three middleboxes: two middleboxes, fw_1 and fw_2, running firewalls that restrict traffic from left to right and from right to left, respectively, and one middlebox, *is*, checking whether isolation between h_2 and h_1 is preserved. In fw_1, e_2 is connected to the "internal" port and e_3 is connected to the "external" port, thus limiting traffic from right to left. In fw_2, e_4 is connected to the "internal" port and e_3 is connected to the "external" port, thus limiting traffic from left

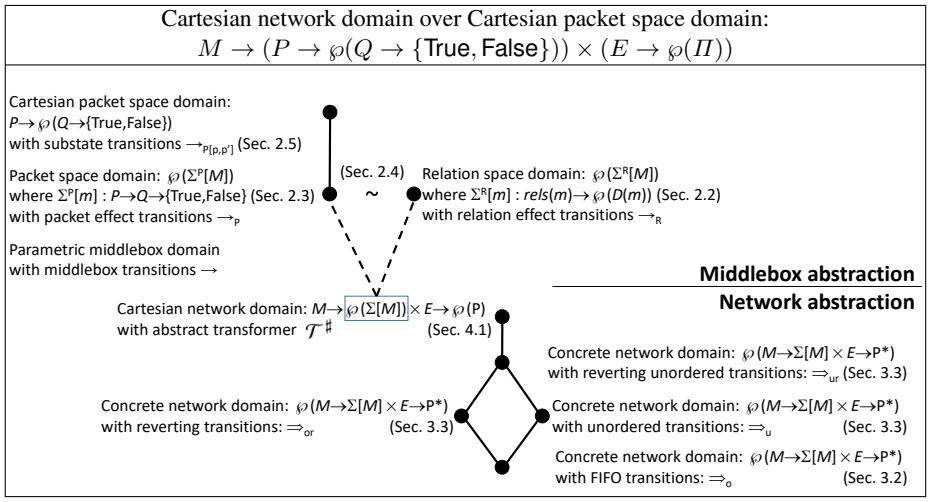


Fig. 12: Hierarchy of abstractions. Solid edges stand for abstraction (either by relaxing the transition relation or by abstracting the configurations). Dashed edges stand for instantiation of the middlebox (local) semantics.

to right. In *is*, e_1 is connected to the "internal" port and e_2 is connected to the "external" port.

Fig. 2 describes the code running in either of the session firewalls, fw_1 and fw_2. We use CSP/OCCAM-like syntax where (messages) packets are sent/received asynchronously. The middlebox non-deterministically operates on a packet from the "internal" port or the "external" port. When reading a packet from the "internal" port, the program distinguishes between two cases. In the first case, a session had been previously established, and the packet is simply forwarded to the "external" port. In the second case the type of the packet is a "request" packet (`type`=0), and the program adds the destination host to the set of `requested` hosts and forwards the "request". The `requested` set is used to store the hosts to which the middlebox sent a "request" packet, to avoid the case where a session is established with a host that the middlebox did not send a "request" to. Packets that do not fall into any of these two cases are discarded with no further processing.

When the middlebox reads a packet from the "external" port again it distinguishes between two cases — in one case a session had previously been established, and is similar to its "internal" counterpart. In the second case, the processed packet is a "response" packet (`type`=1) from a host that is in the `requested` set, and the program marks the source of the packet as `trusted`, thus establishing a session. Other packets are discarded.

A "data" packet (`type`=2) is implicitly handled by checking whether the source/destination of the packet is in the `trusted` set, and if so, allowing the packet to propagate on.

(a) A network topology.

```
is = do
  external_port ? p =>
    if
      p.src = forbidden => abort
      □
      true => internal_port ! p
    fi
  □
  internal_port ? p =>
    true => external_port ! p
od
```

(b) AMDL code for *is*.

Fig. 13: Network topology and AMDL code for the running example.

Fig. 13b describes the code running in a special middlebox, *is*, which intercepts packets before they arrive to host h_1 — the middlebox non- deterministically reads a packet from the "external" port and aborts if the source of the packet is the host `forbidden` $= h_2$, and otherwise forwards to h_1 on the "internal" port. On the other direction, it simply forwards packets from the "internal" port to the "external" port. In this example, *is* models the safety property.

E.1 Analysis Using Network Level Abstractions

Tab. 1 shows the run of our analysis, when restricted to the network-level abstractions, on the running example. Each row corresponds to a step in the least fixpoint computation of the (abstract) reachable network states. Each column at the table represents the abstract content of a channel (as a set of packets) or the abstract state of an individual middlebox (as the contents of its set-valued variables). For each channel e, $\overrightarrow{e}$ denotes channels connecting traffic from left to the right, while $\overleftarrow{e}$ denotes channels connecting traffic from right to the left. For example, $\overrightarrow{e_1}$ contains packets sent from h_1 to *is*.

Channel abstract states are sets of packets.

For the firewall middleboxes, a (concrete) state is a pair of values for the `requested` and `trusted` sets. An abstract state is a set of such (concrete) states. The isolation middlebox is stateless.

At the initial configuration, the states of fw_1 and fw_2 are pairs of empty sets; the states of channels $\overrightarrow{e_1}$ and $\overleftarrow{e_4}$ are all the packets that hosts h_1 and h_2 can send, respectively.

The analysis ignores the correlations between different columns. At each step, the analysis chooses an input channel and a middlebox state and computes the next state.

$\overrightarrow{\mathbf{e_1}}$	$\overleftarrow{\mathbf{e_1}}$	$\overrightarrow{\mathbf{e_2}}$	$\overleftarrow{\mathbf{e_2}}$	fw_1	$\overrightarrow{\mathbf{e_3}}$	$\overleftarrow{\mathbf{e_3}}$	fw_2	$\overrightarrow{\mathbf{e_4}}$	$\overleftarrow{\mathbf{e_4}}$	**action**
$p_{(1,2,0)}$ $p_{(1,2,1)}$ $p_{(1,2,2)}$	$\emptyset$	$\emptyset$	$\emptyset$	$(\emptyset, \emptyset)$	$\emptyset$	$\emptyset$	$(\emptyset, \emptyset)$	$\emptyset$	$p_{(2,1,0)}$ $p_{(2,1,1)}$ $p_{(2,1,2)}$	initial state
		$p_{(1,2,0)}$								*is* reads $p_{(1,2,0)}$
		$p_{(1,2,0)}$ $p_{(1,2,1)}$								*is* reads $p_{(1,2,1)}$
		$p_{(1,2,0)}$ $p_{(1,2,1)}$ $p_{(1,2,2)}$								*is* reads $p_{(1,2,2)}$
				$(\emptyset, \emptyset)$ $(\{h_2\}, \emptyset)$	$p_{(1,2,0)}$					fw_1 reads $p_{(1,2,0)}$
										fw_1 reads $p_{(1,2,1)}$
										fw_1 reads $p_{(1,2,2)}$
						$p_{(2,1,0)}$	$(\emptyset, \emptyset)$ $(\{h_1\}, \emptyset)$			fw_2 reads $p_{(2,1,0)}$
										fw_2 reads $p_{(2,1,1)}$
										fw_2 reads $p_{(2,1,2)}$
										fw_1 reads $p_{(2,1,0)}$
										fw_2 reads $p_{(1,2,0)}$

Table 1: Modular analysis of the running example with explicit state representation. Only changed values are shown. The abstract states of channels are sets. The abstract states of firewalls are sets of pairs for the values of the `requested` and `trusted` sets. Each cell in the table represent a set of the elements described within, except for empty sets in the initial state. The notation $p_{(i,j,k)}$ stands for the packet from h_i to h_j with type k.

The analysis stops when no more new middlebox states or channel states are discovered and reports potential violation of the safety property if the `abort` command is executed.

In the first action, the code of *is* executes and reads $(h_1, h_2, 0)$ from $\overrightarrow{e_1}$. Notice that this does not change the (abstract) content of this channel. The packet is forwarded to $\overrightarrow{e_2}$. Thus, our analysis only accumulates packets, ignoring their order. The reachable states of the middleboxes are explicitly maintained. For example, when fw_1 reads $(h_1, h_2, 0)$ from $\overrightarrow{e_2}$, it forwards it to $\overrightarrow{e_3}$ and reaches a new state with `requested` $= \{h_2\}$ and `trusted` $= \emptyset$.

Notice that in this example, the analysis proved that the `abort` command can ever be executed on arbitrary packet propagation scenarios. Specifically, no packets ever reaches channel $\overleftarrow{e_2}$, so the safety middlebox *is* never reads a packet that will result in the execution of an `abort` command. Thus, the analysis succeeded in proving isolation.

This example illustrates that, although our analysis employs Cartesian abstraction, it is able to prove a network-wide property. Specifically, proving isolation requires rea-

soning about the states of both firewalls. We note that removing either of the firewalls violates the safety property.

E.2 Analysis Using Network Level and Middlebox Level Abstractions

Tab. 2 shows the verification process with packet states in the running example. Instead of storing the contents of relations `trusted` and `requested` in each middlebox state, we store, for each packet, whether each of the expressions "`p.dst in trusted`", "`p.src in trusted`", and "`p.src in requested`", evaluates to True (T) or False (F), respectively.

Since both relations are empty in the initial state, the packet states for both firewalls map each packet to (F, F, F).

Recall that when fw_1 reads $(h_1, h_2, 0)$ from $\vec{e_2}$, it forwards it to $\vec{e_3}$ and reaches a new state with `requested` $= \{h_2\}$ and `trusted` $= \emptyset$. Therefore, any future evaluation of the expression "`p.src in requested`" (for any value of `type`) should result in True. Under the packet state representation, this would result in adding to the abstract state of fw_1 a packet state similar to that of the initial state where each of the packets $p_{(2,1,0)}$, $p_{(2,1,1)}$, and $p_{(2,1,2)}$ is re- mapped from (F, F, F) to (F, F, T). Our middlebox-level Cartesian abstraction allows us to instead accumulate these mappings (separated by a horizontal line from the initial mappings) in a single abstract state, *without affecting the overall precision of the abstract interpretation*.

A similar change to the packet state of fw_2 occurs upon reading the packet $(h_2, h_1, 0)$ from $\overleftarrow{e_4}$.

F Networks with unbounded number of hosts

In this section, we prove the lack of small model to stateful networks, w.r.t number of network hosts. This property holds even for reverting networks with only a single middlebox and packets of the type (s, d, t) where s and d are hosts i.e., $s, d \in H$, and t, the packet type, is taken from a bounded type set T.

Small model property. For simplicity, we consider only a network with a single middlebox m that never output packets. The small model property is a bound $b(m)$, such that any network with the above topology is safe if and only if any network with the above topology and at most $b(m)$ hosts is safe. And if for certain number of hosts the network is not safe, we define $b(m) = \infty$.

Theorem 5. *The function $b(m)$ is not a computable function. In particular, the problem of deciding whether $b(m) < \infty$ is undecidable.*

We prove the above theorem by a reduction to the halting problem. We show that giving a Turing machine M, we can construct a middlebox $m(M)$ such that $b(m(M)) = \infty$ if and only if M is never halts and is using unbounded space on its run when then initial input is empty (which is known to be undecidable).

$\overrightarrow{\mathbf{c_1}}$	$\overleftarrow{\mathbf{c_1}}$	$\overrightarrow{\mathbf{c_2}}$	$\overleftarrow{\mathbf{c_2}}$	fw_1	$\overrightarrow{\mathbf{c_3}}$	$\overleftarrow{\mathbf{c_3}}$	fw_2	$\overrightarrow{\mathbf{c_4}}$	$\overleftarrow{\mathbf{c_4}}$	**action**
$p_{(1,2,0)}$ $p_{(1,2,1)}$ $p_{(1,2,2)}$	$\emptyset$	$\emptyset$	$\emptyset$	$p_{(1,2,0)} \mapsto (F,F,F)$ $p_{(1,2,1)} \mapsto (F,F,F)$ $p_{(1,2,2)} \mapsto (F,F,F)$ $p_{(2,1,0)} \mapsto (F,F,F)$ $p_{(2,1,1)} \mapsto (F,F,F)$ $p_{(2,1,2)} \mapsto (F,F,F)$	$\emptyset$	$\emptyset$	$p_{(1,2,0)} \mapsto (F,F,F)$ $p_{(1,2,1)} \mapsto (F,F,F)$ $p_{(1,2,2)} \mapsto (F,F,F)$ $p_{(2,1,0)} \mapsto (F,F,F)$ $p_{(2,1,1)} \mapsto (F,F,F)$ $p_{(2,1,2)} \mapsto (F,F,F)$	$\emptyset$	$p_{(2,1,0)}$ $p_{(2,1,1)}$ $p_{(2,1,2)}$	initial state
		$p_{(1,2,0)}$								*is* reads $p_{(1,2,0)}$
		$p_{(1,2,0)}$ $p_{(1,2,1)}$								*is* reads $p_{(1,2,1)}$
		$p_{(1,2,0)}$ $p_{(1,2,1)}$ $p_{(1,2,2)}$								*is* reads $p_{(1,2,2)}$
				$p_{(1,2,0)} \mapsto (F,F,F)$ $p_{(1,2,1)} \mapsto (F,F,F)$ $p_{(1,2,2)} \mapsto (F,F,F)$ $p_{(2,1,0)} \mapsto (F,F,F)$ $p_{(2,1,1)} \mapsto (F,F,F)$ $\underline{p_{(2,1,2)} \mapsto (F,F,F)}$ $p_{(2,1,0)} \mapsto (F,F,T)$ $p_{(2,1,1)} \mapsto (F,F,T)$ $p_{(2,1,2)} \mapsto (F,F,T)$	$p_{(1,2,0)}$					fw_1 reads $p_{(1,2,0)}$
										fw_1 reads $p_{(1,2,1)}$
										fw_1 reads $p_{(1,2,2)}$
						$p_{(2,1,0)}$	$p_{(1,2,0)} \mapsto (F,F,F)$ $p_{(1,2,1)} \mapsto (F,F,F)$ $p_{(1,2,2)} \mapsto (F,F,F)$ $p_{(2,1,0)} \mapsto (F,F,F)$ $p_{(2,1,1)} \mapsto (F,F,F)$ $\underline{p_{(2,1,2)} \mapsto (F,F,F)}$ $p_{(1,2,0)} \mapsto (F,F,T)$ $p_{(1,2,1)} \mapsto (F,F,T)$ $p_{(1,2,2)} \mapsto (F,F,T)$			fw_2 reads $p_{(2,1,0)}$
										fw_2 reads $p_{(2,1,1)}$
										fw_2 reads $p_{(2,1,2)}$
										fw_1 reads $p_{(2,1,0)}$
										fw_2 reads $p_{(1,2,0)}$

Table 2: Packet state enumeration for the running example. The abstract states of channels are sets of packets. The abstract states of middleboxes are relations over packets and query valuations; each entry in the table is denoted by $\mapsto$. Each cell in the table represent a set of the elements described within, except for empty sets in the initial state. The horizontal lines in the fw_1 and fw_2 columns appear to emphasize the changes. As before, $p_{(i,j,k)}$ stands for the packet from h_i to h_j with type k.

Proof overview Given a Turing machine M over alphabet σ we construct a network with a single middlebox m and a host set H and packet space $P = H \times H \times T$ such that N is safe if and only if M does not halts for any run that requires at least $|H|$ space.

Informally, we construct m such that initially m encodes a successor relation over H, and later it uses the relation to simulate the run of the Turing machine m for $|H|$ cells in the turing machine tape. If in using at most $|H|$ space the Turing machine halts, then m goes to an abort state. Hence, N is safe iff M does not halt using at most $|H|$ space.

Detailed proof sketch We assume a constant symbol h_0 (the first host). For the successor construction, the middlebox m has the next relations:

- $R_{\text{successor}}(h_1, h_2)$. Intuitively, $R_{successor}(h_1, h_2) = True$ stands for $h_1 = h_2 + 1$. Initially, the relation returns false to all pairs.
- $R_{\text{max host}}(h)$. Intuitively, $R_{\text{max host}}(h) = True$, if h was the last host that was assigned as a successor. Initially, only $R_{\text{max host}}(h_0) = True$.
- $R_{\text{already in order}}(h)$. Intuitively, $R_{\text{already in order}}(h) = True$ if h was already assigned as a successor. Initially only $R_{\text{already in order}}(h_0) = True$.

In the successor construction phase, m construct an order, given an input packet (s, d, t) as follows: If $R_{\text{max host}}(s)$ is false or $R_{\text{already in order}}(d)$ is true, it goes to a sink state. Otherwise it set $R_{\text{max host}}(s) = False$, $R_{\text{already in order}}(d) = True$, $R_{\text{max host}}(d) = True$ and $R_{successor}(s, d) = True$. A special packet type $t = 1$ indicates that m should leave the successor construction phase and go to simulation phase.

To describe the simulation phase, we first recall that a Turing machine has a finite set of states Q and a finite input/output alphabet Σ. In every step, the machine reads an input from the head, write a new symbol to head, and moves the head one step to the right or to the left (w.l.o.g, we assume that head position is changed in every step). At this phase, hosts represent turing machine head position. For the Turing machine simulation phase the middlebox has the next relations:

- For every $\sigma \in \Sigma$: $R_{\text{symbol}_\sigma}(h)$. Intuitively, it is true if and only if the symbol on the $h - th$ position is σ. Initially, it is false for all pairs.
- $R_{\text{expected position}}(h)$. Intuitively, it is true if and only if the head is expected to be in position h. Initially, only $R_{\text{expected position}}(h_0)$ is true.
- For every $q \in Q$: $R_{\text{state}_q}()$ is true iff the machine is at state q. Initially, only $R_{\text{state}_{q_0}}()$ is true.

In this state, m simulates the machine as follows: given a packet (s, d, t):

- Check head position: If $R_{\text{expected position}}(s) = False$ go to sink state.
- Query head symbol: go over all $R_{\text{symbol}_\sigma}(s)$ and extract current head symbol σ (if it is false for all symbols, then the cell is empty, i.e., $\sigma = \epsilon$).
- Query current state: go over all $R_{\text{state}_q}()$ and extract current state q.

- Update head symbol and current state: set $R_{\text{symbol}_\sigma}(s) = False$ and $R_{\text{symbol}'_\sigma}(s) = True$ where σ' is the output symbol (according to the turing machine). Similarly update the current state relation.
- Update expected head position: If at state q and input σ the head moves left, then if $d \neq s - 1$ (according to the successor relation) then go to sink state. Otherwise set $R_{\text{expected position}}(s) = False$, and $R_{\text{expected position}}(d) = True$. If the head moves right, check if $d = s + 1$ and act in the same way.
- if q is a final state, then abort.

Lemma 9. *The network is safe if and only if M does not halt using at most $|H|$ space.*

Proof. If M halts using at most $|H|$ space, then a sequence of packets which construct the order and simulate the run without going to a sink state leads to an abort state. If M does not halt with at most $|H|$ space, then any sequence of packets must end in a sink state.

Additional observations

- The program is only using the inputs s, d and t and a single constant h_0. In the construction it is enough to have $t \in \{0, 1\}$.
- Same proof holds for reverting middlebox. Indeed, whenever the middlebox reverts, the state of the turing machine and the relation order are reset, and the run starts from scratch. This is thanks to the fact that m does not output any packets.