NetSMC: A Custom Symbolic Model Checker for Stateful Network Verification

Yifei Yuan[1] *Soo-Jin Moon*[2] *Sahil Uppal*[2] *Limin Jia*[2] *Vyas Sekar*[2]
[1]*Intentionet* [2]*Carnegie Mellon University*

Abstract

Modern networks enforce rich and dynamic policies (e.g., dynamic service chaining and path pinning) over a number of complex and stateful NFs (e.g., stateful firewall and load balancer). Verifying if those policies are correctly implemented is important to ensure the network's availability, safety, and security. Unfortunately, theoretical results suggest that verifying even simple policies (e.g., A cannot talk to B) in stateful networks is undecidable. Consequently, any approach for stateful network verification has to fundamentally make some relaxations; e.g., either on policies supported, or the network behaviors it can capture, or in terms of the soundness/completeness guarantees. In this paper, we identify practical opportunities for relaxations in order to develop an efficient verification tool. First, we identify key domain-specific insights to develop a more compact network semantic model which is equivalent to a general semantic model for checking a wide range of policies under practical conditions. Second, we identify a restrictive-yet-expressive policy language to support a wide range of policies including dynamic service chaining and path pinning while enable efficient verification. Third, we develop customized symbolic model checking algorithms as our model and policy specification allows us to succinctly encode network states using existential first-order logic, which enables efficient checking algorithms. We prove the correctness of our approach for a subset of policies and show that our tool, NetSMC, achieves orders of magnitude speedup compared to existing approaches.

1 Introduction

Today's computer networks deploy a large number and variety of complex *stateful* functions [44], ranging from stateful firewalls, NATs, to proxies, and load balancers. Network operators configure those network functions (NFs) to enforce *rich and dynamic policies*, such as dynamic service chaining (e.g., all packets should traverse IPS, while only malicious packets detected by IPS should be sent to FW) [18, 19] and path pinning (e.g., if packets from A to B traverse NF f_1 and then f_2, the reverse packets from B to A should traverse f_2 and then f_1). Formally verifying if the network correctly implements the policies is critical to ensure the availability, security, and safety of the network.

Checking whether policies are correctly enforced, however, is challenging on stateful networks (networks with stateful NFs). Even checking simple policies such as isolation policies (packets from A cannot be delivered to B), an efficiently solvable problem on stateless networks [23, 25–27, 31], is undecidable on stateful networks [47]. In practice, policies enforced on stateful networks will be more complex (see §2).

As such, making practical progress requires non-trivial trade-offs on the supported network behavior, expressiveness of policies, and the soundness/completeness guarantees. For example, recent work VMN [41] simplifies the behavior of a stateful network by assuming that each NF can buffer multiple packets in an out-of-order way, which makes the problem decidable for checking a restrictive set of policies. To verify policies, VMN encodes the network and the policy using first-order logical formulas which are solved by a general-purpose SMT solver. However, it is inefficient to even check the isolation policy due to its high complexity (i.e., EXPSPACE-complete [47]).

In this work, we revisit the stateful network verification problem and explore a different set of relaxation trade-offs in order to achieve more efficient verification for practical scenarios based on the following domain-specific insights:

One-packet at a time network model: Instead of dealing with multiple packets, we adopt a simpler model where only one packet exists in every network state. This model is motivated by the fact that packets inducing conflict behavior on a network are often processed by the network in an *order-preserving* way. For example, connection-based NFs often process packets in a connection in order. Thus, each packet would be processed by the network exactly in the same way when traversing the network with other packets as when traversing alone. Therefore, we can consider only one packet at a time in each network state. While this model simplifies the behavior of a network (e.g., we cannot find violations appearing only under packet interleaving. See §8), we show that the verification result of a wide range of policies (e.g., isolation) based on this model is correct w.r.t. the more complex model in previous work [41] for order-preserving networks (details in §4).

Customized policy and verification algorithm: We design a restricted-yet-expressive policy language based on a subset of linear temporal logic and verification algorithms based on symbolic model checking (SMC) to achieve further speedup of the verification. While our model of network behavior reduces the state space of the problem, checking simple policies (e.g., isolation) efficiently is still hard. Naive approaches based on reducing the problem to constraint solving using general-purpose solvers is not particularly efficient since it would not benefit from the simpler model (see §7).

There are two key challenges to apply the SMC framework to stateful networks: 1) how to succinctly encode a large number of network states and 2) how to efficiently support the computation required in SMC. To this end, we leverage the customized policy structure to use simple existential first order logic (EFO) formulas to succinctly encode a large number of network states. Furthermore, we develop efficient algorithms required in the SMC framework by leveraging the simple network model and extending classic algorithms in other domains (e.g., query containment in database theory).

Based on the key insights discussed above, we implement NetSMC, a symbolic model checker for stateful networks. We prove the correctness of our algorithms w.r.t. a general network semantic model for a subset of policies (e.g., isolation properties). For other policies requiring reasoning about packet interleaving, NetSMC is a sound-but-incomplete bug finding tool that is very efficient. We show the effectiveness of NetSMC via several use cases using real-world NFs such as pfSense [3] and HAProxy [2] running in Cloudlab [43]. We evaluate NetSMC on various network topologies and policies and show that NetSMC scales to networks with hundreds of stateful NFs and is $>$ 200X faster than the state-of-the-art stateful network verification tool VMN [41] on typical fattree-topology networks.

2 Motivation

We motivate the stateful network verification problem by describing several practical policies, followed by the key challenges of the verification problem.

2.1 Stateful Network Verification Examples

Isolation. Consider a network (Fig. 1a) with a stateful firewall to protect the Department from the Internet. Network operators may enforce the isolation policy: traffic from untrusted hosts in the Internet cannot be sent to the Department.

Conditional reachability. Continuing the example above, the network operators may additionally enforce the policy to allow all traffic from the Department and to allow traffic from those trusted hosts in the Internet that have a connection already established from a host in Department.

Flow affinity. Consider a load balancer that distributes traffic among n servers as shown in Fig. 1b. To keep the service provisioning undisrupted, the network operator wants to enforce the following flow affinity policy: if a packet from a host Client is load balanced to a server, then all future packets in the same flow should always be sent to the same server.

Dynamic service chaining. Fig. 1c shows a multi-stage intrusion prevention system (IPS) consisting of a light IPS and a heavy IPS. Each device in the network is configured such that all traffic from the Department is sent to the light IPS, which performs basic detection such as counting the number of bad connections for each host. If a host is detected suspicious by light IPS (e.g. issuing more than 10 bad connections), all future packets from the host should be directed to the heavy IPS for further processing; otherwise its traffic is directly sent to the Internet.

Path pinning. Often a network needs to deploy multiple instances of the same middlebox function for better throughput.

	Buzz & Symnet	VMN	**NetSMC**
Model	One-packet	Out-of-order	One-packet
Policy lang.	Assertion	LTL-based	LTL-based
Correctness	Sound	Sound, Complete	Sound, Cond. complete

Table 1: Comparison with network verification tools.

Consider the network shown in Fig. 1d which is configured to forward traffic between the Department and the Internet to one of the firewalls. An interesting path pinning policy is: if a packet from H1 in the Department to H2 in the Internet goes through the i-th firewall, then all future packets from H2 to H1 should traverse the same firewall.

2.2 Challenges

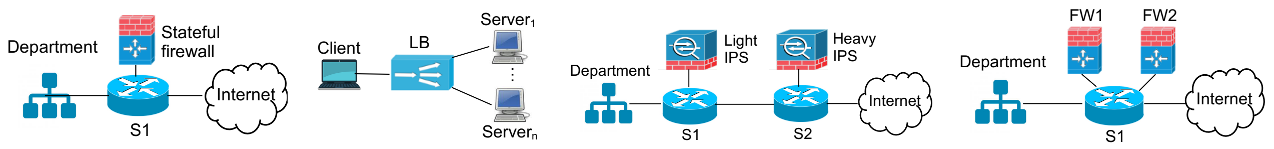
Stateful network verification is more challenging compared to stateless verification. In general, this problem has been shown to be undecidable even for simple isolation policies (see Theorem 1 in [47]). As such, any practical progress needs to make practical relaxations on at least one of the following dimensions: the supported network behavior, the expressiveness of policies, and the correctness guarantees.

As an example, VMN [41] recovers the decidability of the problem by assuming that an NF buffers packets in an out-of-order fashion instead of FIFO. VMN reduces the verification problem to encoding network configurations and policies as first-order logical constraints which can be solved by an SMT solver. Since solving general first-order constraints is undecidable, VMN targets policies in the form of "if packet p reaches node B then p must not satisfy some property P in the past", so that the encoded constraints fall in a decidable fragment. Even with the above simplification the verification problem induces high complexity (i.e., EXPSPACE-hard [47]). Thus, VMN is not scalable to large-size networks even for checking simple isolation policies as shown in §7.

3 Overview

The undecidability result [47] means that *any* approach in this space has to seek some relaxations or tradeoffs in order to make the problem tractable. One of our contributions is identifying and exploring a different point in this space of practical relaxation choices in order to develop an efficient verification tool in practical network scenarios. Table 1 summarizes key difference of our trade-offs compared with existing work.

More specifically, our trade-off is based on the following two key domain-specific insights: First, the key challenge to stateful network verification is to handle interleaving among multiple packets in the network (e.g., packet p_1 is processed by NF A before packet p_2, but p_2 is processed by NF B before p_1). In practice, however, networks often exhibit intrinsic *order-preserving* property in several scenarios, where packets that induce conflict network behavior are processed in the same order. For example, a syn packet is often processed by the network before a synack packet is sent into and processed by the network. Motivated by this observation, we



(a) Stateful firewall. (b) Load balancing. (c) Multi-stage IPS. (d) Multiple stateful firewalls.

Figure 1: Examples

adopt a semantic model that only allows one packet being considered at each network state. In effect, we consider a sequential execution model of networks, similar to the ones considered in stateful network testing tools [18,45]. Second, even with this simple model verifying simple isolation policies is still hard (i.e. PSAPCE-hard) and naively employing a general-purpose tool is not efficient for large-size networks (see §7). Motivated by recent success in customized stateless network verification tools (e.g., HSA [26], VeriFlow [27]), we design a customized policy language and verification technique for stateful networks based on the symbolic model checking (SMC) framework. We are able to identify efficient symbolic representations for network states and develop efficient SMC algorithms.

Next, we discuss our key design choices before delving into more detailed algorithmic designs in the following sections.

Network model (§4). To model the behavior of a stateful network, we need two key components, an NF model that can capture various NF behavior and the modeling of packets traversing the network as we discussed above.

We need an expressive yet restrictive model that can model various NF behavior while supporting efficient verification. On one end of the spectrum, we could use a general purpose language (e.g., C in Buzz [18]), but that leads to highly inefficient verification. Motivated by existing NF models [7,52], a stateful NF can be modeled as: 1) a set of state tables indexed by packet header fields, and 2) a set of rules that modify those state tables based on packet matching and table testing results of the incoming packet. Such restricted formalization enables efficient checking algorithms in SMC as we show in §6.

Policy specification language (§5). As shown in §2, stateful network policies have temporal properties, so a temporal specification language such as linear temporal logic (LTL) [42] is a natural choice for specifying such policies. While it is expressive enough for a wide set of network policies, the full set of LTL is computationally difficult to handle.

Our insight is that the set of network policies of interest fall into the intersection of LTL and computational tree logic (CTL) which has more efficient verification algorithms [13]. Therefore, we identify a subset of LTL that is intuitive and expressive to specify a wide range of policies while using efficient verification algorithms based on CTL. As we show in §6, reasoning about policies in this set of policies also enables us to use succinct symbolic encoding of network states and efficient checking algorithms required in SMC.

Efficient verification algorithm (§6). We design customized verification techniques based on classic SMC framework for efficiency. Algorithms for the symbolic model checking framework are well known. However, there are a set of challenges to instantiate the framework in the context of stateful network verification. First, we need a succinct symbolic representation for a large set of network states, particularly for the internal state (e.g., connection state tables) of NFs. Second, the symbolic model checking algorithm requires computing the pre-image of a symbolic state (i.e., the set of states that can transition to this state in one step), and the termination of the symbolic model checking framework requires efficient computation to check containment of two sets of states in the symbolic representation. How to efficiently support the computations remains another challenge. While classic data structures such as BDD are widely used in other contexts, it is not suitable in the context of stateful networks due to the large state space. For example, a stateful firewall maintains state for each flow and thus the number of states is as large as 2^F where F, the maximal number of flows a firewall tracks, can be in the order of thousands. Our design of the network model and policies allows us to address those two challenges. First, we succinctly encode a large set of states into a fragment of existential first-order logic (EFO). As an example for a stateful firewall, we may use $\exists x, y. \mathit{Trust}[x, y] = 1$ to represent all network states where there is a legitimate flow (a src-dst pair for simplicity) recorded by the connection table *Trust*. Our choice of the policy language ensures that any symbolic state emerged during the computation of SMC can be encoded in EFO. Second, our simple NF model allows us to efficient compute the pre-image of a symbolic state . Moreover, using EFO, we identify the connection between containment checking of sets of states in EFO and conjunctive query containment problem in the database community [11,29]. We adapt the query containment checking algorithm to efficiently check the containment of two sets of network states.

4 Stateful Network Model

We present our stateful network model, including the NFs and semantic rules. We illustrate the expressiveness of our NF model via example encodings of common network functions.

4.1 NF Model

We summarize the syntax in Fig. 2. Inspired by prior work [7,52], each NF includes local state which are key-value maps and a set of rules for processing packets and updating local state. NFs' computations are restricted to state checking, simple counting, and non-deterministic value choice. This model is more expressive than the one used by the verification

Field Name	f	$\in$	{`srcip`, `dstip`, `srcport`... }
Value	v	$\in$	$\text{Int} \cup \text{IP} \cup ...$
Packet	pkt	$::=$	$\{\overrightarrow{f_i = v_i}\}$
Location	l	$\in$	Loc
Located Packet	lp	$::=$	(l, pkt)
State Table	T	$\in$	TableNames
Expression	e	$::=$	$v \mid f \mid$ **pickFrom**$(D) \mid T[\overrightarrow{e_i}]$
Atomic Test	at	$::=$	True $\mid$ `loc`$\in D \mid f \in D \mid T[\overrightarrow{e_i}] \in D$
Test	t	$::=$	$at \mid \neg at \mid t, t$
Update	u	$::=$	$T[\overrightarrow{e_i}] := e \mid$ **inc**$(T[\overrightarrow{e_i}], v) \mid$ **dec**$(T[\overrightarrow{e_i}], v)$
Action	a	$::=$	**fwd**$(e) \mid$ **drop** $\mid$ **modify**(f, e)
Command	c	$::=$	$u \mid a \mid c; c$
Rule	r	$::=$	$t \Rightarrow c$
NF Config	R	$::=$	$\cdot \mid r; R$
NF	NF	$::=$	$(\overrightarrow{l}, \overrightarrow{T}, R)$
Network Topo	$topo$	$\in$	$\text{Loc} \to \text{Loc}$
Network Config	N	$::=$	$(topo, [NF_1, .., NF_k])$
Table Valuation	Δ	$\in$	$\text{TableNames} \to \delta_T$
Network State	s	$::=$	(lp, Δ)

Figure 2: Syntax of stateful network model.

tool VMN [41] and is efficient (§6).

Basics. We write pkt to denote packets, which are records of packet fields (notation $\{\overrightarrow{t_i}\}$ is a shorthand for $\{t_1, \cdots, t_n\}$). Packet field names, denoted f, are drawn from a set of predefined names, including common field names such as `srcip`, `dstip`, `srcport` and user-defined application specific field names. We use Loc to denote the set of all locations (e.g., interfaces at a switch) in the network, including two special ones: Drop (denoting that packets are dropped) and Exit (denoting that packets exit the network). A located packet, denoted lp, is a pair of a location and a packet.

NF. We model all the network devices as *network functions*, denoted NF, which is a tuple consisting of a set of locations $\overrightarrow{l}$ (i.e. interfaces), a set of tables $\overrightarrow{T}$ for storing internal state (e.g. a stateful firewall may use state tables to store connection state), and a list of rules R that process packets and update its state. Stateless devices' state tables are empty.

A rule r consists of a list of tests on packet fields and state tables, denoted t, and a sequence of commands, denoted c, for updating the state and generating the outgoing packet. For instance, a stateful firewall may drop or forward the packet (captured by c) depending on the result of testing the packet headers and the internal state (captured by t). A rule r is fired, i.e., its commands are executed, when the current packet and state tables pass the tests in r.

We allow the following atomic tests: trivial tests that return true; tests that check the current location of the incoming packet; tests that check whether a field value or the value of a state table entry is in a specified finite domain D (e.g., an interval). Common features such as longest prefix matching for IP addresses can be modeled using $f \in D$. A command c is a sequence of updates to state tables, denoted u and actions applied to packets, denoted a.

We write e to denote expressions that can be used by rules of NFs, which include constants, packet field values indexed by field names, values picked (nondeterministically) from a domain D (**pickFrom**(D)), and values stored in state tables. Each state table is a finite key-value map, where we write $T[\overrightarrow{e_i}]$ to denote the value in an entry indexed by the key $\overrightarrow{e_i}$.

A state table can be updated. The update $T[\overrightarrow{e_i}] := e$ updates the entry indexed by the key $\overrightarrow{e_i}$ to the value of e. We allow simple counting operations to model IDS/IPS. **inc**$(T[\overrightarrow{e_i}]$, $v)$ increments the value in the table entry by a constant v; **dec**$(T[\overrightarrow{e_i}], v)$ performs decrementing similarly. We consider the following actions for incoming packets: forwarding, dropping, and modifying the value of a packet field. We do not model multicasting or broadcasting in this paper.

Upon receiving a packet pkt at a location l, an NF attempts to match the located packet $lp = (l, pkt)$ with all of its rules. The matching succeeds if all atomic tests in the rule are true given lp and the current state tables. For an atomic test that involves a field name f (e.g., $f \in D$), that field name evaluates to the value of the field f in the packet pkt. As an example, an atomic test `Trust[`**`dst,src`**`]=1` first evaluates **`src`** and **`dst`** to be the source and destination addresses of the incoming packet pkt, then uses the concrete values as the key to look up the entry in the table `Trust`, and finally checks if the corresponding entry is `1`. If the matching succeeds, all actions and updates of the rule are applied sequentially. Without loss of generality, we assume that exactly one rule can match an incoming packet. It is straightforward to translate other models such as the one based on first-match into this model.

4.2 NF Examples

To demonstrate the expressiveness of our model, we show example encodings of several stateful network functions. Writing a NF model is a one-time effort, and can be automated (c.f. [48]), which is out of the scope of this work.

Stateful firewall. A stateful firewall protects an internal network by restricting accesses from external hosts. Fig. 3 shows the code snippet of a stateful firewall. Here, we assume that the internal network is connected to location 0 of the stateful firewall, and the outside network is connected to location 1. The stateful firewall uses a state table `Trust` to keep track of the flows that are established by the internal network. Initially, all entries in `Trust` have value 0. When a packet comes from the internal network, the firewall forwards it directly to the outside, and updates the state table entry for that flow to 1 (the 1st rule). When a packet comes from outside (location 1), the firewall first checks the state table to see whether a packet in the reverse direction has been seen (i.e., the table entry is 1); if so, the packet is forwarded (the 2nd rule); otherwise the packet is dropped (the 3rd rule).

Load balancer. A load balancer forwards packets destined for a virtual destination of a service (e.g., online searching) to one of the backend servers that implement the service. Fig. 4 shows a load balancer for a service with virtual IP address

```
loc=0 => Trust[src,dst]:= 1, fwd(1);
loc=1, Trust[dst,src]=1 => fwd(0);
loc=1, Trust[dst,src]=0 => drop;
```

Figure 3: Stateful firewall.

VIP, where we assume that servers are connected to location 1 and clients are connected to location 0. The load balancer maintains two state tables, `Connected` for storing whether a client has been assigned to a server and `Server` for storing the address of the server assigned to each client. Initially all table entries have value 0, indicating that no server has been assigned to any client. The first rule corresponds to the case where a client was assigned to a server (i.e. `Connect[src]=1`), and the load balancer needs to modify the destination address of the packet to the address of the assigned server as stored in the `Server` table. Similarly, the second rule accounts for the case where the client has not been assigned to any servers (i.e., `Connect[src]=0`). In this case, the load balancer picks a server from all the backend servers `D`, updates the state tables, and modifies the packet destination accordingly. Note that the use of `pickFrom(D)` abstracts away the concrete mechanism of choosing the server for a client. For packets not destined to the service, the load balancer may drop the packets as shown in the third rule. Last, for traffic going from servers to clients, the load balancer simply modify the source address of the packet to the virtual address, as indicated by the last rule.

```
loc=0, dst=VIP, Connected[src]=1 =>
  modify(dst, Server[src]), fwd(1);
loc=0, dst=VIP, Connected[src]=0 =>
  Server[src]:=pickFrom(D), Connected[src]:=1,
  modify(dst, Server[src]), fwd(1);
loc=0, dst!=VIP => drop;
loc=1 => modify(src, VIP), fwd(0);
```

Figure 4: Load balancer.

4.3 Network Semantic Model

Network configuration. We consider a network configuration N as a set of links, denoted *topo*, together with a set of NFs in the network. We model the execution of a stateful network as a state transition system, where each state corresponds to a snapshot of the network (referred to network state) and a transition between two network states denotes an atomic step of feasible network execution. In each network state, we need to consider the valuation of state tables of each NF as well as the packets in the network. We use a function Δ, which maps each state table T to a function δ_T, to denote the valuation of all NF state tables.

Packet processing in the network. To model packets in the network, a general approach is to associate each NF interface with *an infinite* FIFO queue buffering packets to be processed. As mentioned in §1, unfortunately, policy checking in such models is undecidable. Recent progress [41] relaxes this model by assuming packets are buffered in an out-of-order way. While this out-of-order model recovers decidability for some policies, it still incurs high computational complexity.

For efficient verification, we consider a model where only one packet exists in any network state. We model a network state as a pair (lp, Δ), where lp is a located packet being processed and Δ is a table valuation. Our model has three types of transitions: (1) At a network state (lp, Δ), the packet is received by a NF and NF modifies the located packet to lp' and updates the state tables to Δ', and the network evolves to state (lp', Δ'); (2) When a packet pkt is moved from one end l of a link to the other end l', a network state $((l, pkt), \Delta)$ can evolve to $((l', pkt), \Delta)$; (3) When the current packet pkt is dropped or exits the network, a new packet at an ingress location is brought into the network.That is, $((O, pkt), \Delta)$ can evolve to $((I, pkt'), \Delta)$, where O is Drop or Exit, I denotes an ingress location, and pkt' is an arbitrary packet. We write $E = s_0 \xrightarrow{lp_0/lp_1} \cdots \xrightarrow{lp_{n-1}/lp_n} s_n$ to denote execution traces, where s_i is a network state, lp_{i-1}/lp_i denotes the processed located packet and the resulting located packet in step i respectively. We assume no indefinite loops for any packet; transient loops are allowed. Detailed rules are in Appendix A.

Connecting to packet-interleaving model. Our one-packet model excludes packet interleaving behaviors, and thus cannot find all policy violations. To answer the question: *when can packet interleaving be safely ignored*, we first formalize when do the two models agree and what do they agree on.

Let us write $\mathsf{E}^{\infty}(N)$ to denote the set of all closed (i.e., both the initial and final states have only packets at locations Drop or Exit) finite execution traces of the network N under the packet interleaving semantics; similarly, we write $\mathsf{E}^{\mathsf{one}}(N)$ to denote the set of all finite closed network execution traces under the one-packet model. We assume that each packet has an unique ID, that is not modified by any NFs. Given a network execution trace $E = s_0 \xrightarrow{lp_0/lp_1} \cdots \xrightarrow{lp_{n-1}/lp_n} s_n \in \mathsf{E}^{\mathsf{one}}$ of a network N and a packet ID id, we define *per-packet trace* for a packet with ID id, denoted $E|_{id}$, as the sequence $[lp_{i_1}/lp_{i_1+1} \ldots lp_{i_k}/lp_{i_k+1}]$ obtained by keeping only those lp_i/lp_{i+1} pairs whose packet ID is id. Per-packet trace for $E \in \mathsf{E}^{\infty}$ can be defined similarly.

It turns out that most network policies are checkable on per-packet traces. For example, checking the policy "packets from N1 cannot reach N2" can be achieved by examining packet-traces for every packet in the network. Then, when are the per-packet traces of the one-packet model the same as the per-packet traces of the packet interleaving model?

Our key insight is that a wide range of network scenarios are intrinsically *order-preserving*, where packets whose interleaving matters (i.e., swapping their process orders induces divergent network behavior) are processed in the same order by all NFs in the network. For example, the state update of connection-based stateful NFs is triggered by control packets of a connection, which are often sent to and processed by the network in order (e.g., a syn packet is processed before a synack packet). Thus, a network with connection-based stateful NFs is order-preserving. As another example, modern network devices often integrate a pipeline of NFs where packets traverse them in order. Networks with such stateful devices on the edge of the network such that any packet only traverses one of them is also an order-preserving network. The formal definitions are in Appendix C. We can show that for order-preserving networks, the packet interleaving model and the one-packet model agree on per-packet traces. Formally:

Lemma 1 *Given an order-preserving network N, $\forall E \in \mathsf{E}^{\infty}(N)$, $\exists E' \in \mathsf{E}^{\mathsf{one}}(N)$, s.t. $\forall id$, $E|_{id} = E'|_{id}$.*

The above lemma is the building block for proving the conditional soundness and completeness results of our algorithm.

5 Network Policies

To strike a reasonable balance between efficiency and expressiveness, we use a subset of the linear temporal logic (LTL) as our specification language. This language is expressive enough to specify a wide range of policies as we show in §7 and also is more efficient for policy checking compared to the full set of LTL formulas. Policies in this language can be translated to equivalent ones in the computational tree logic (CTL), thus allowing more efficient checking algorithms of CTL to be used [14]. Further, a simple fragment of first order logic can be used to reason about network states (details in §6). We envision tools could be used to build policy templates (e.g., [41]) or GUI (e.g., [18]), to ease the policy specification process. We provide a high-level overview of our policy specification language with details in Appendix B. We show an example policy specification from §2. We end by a theorem stating that the one-packet and packet interleaving model agree on checking several practical policies.

Syntax. The syntax of our specification language is shown below. Predicates, denoted θ, include equality checks between a packet field value, the current location of the packet, and a state table value and a variable (a symbolic value).

$$\begin{array}{llcl}
\text{Predicate} & \theta & ::= & f = x \,|\, \texttt{loc} = x \,|\, T[\overrightarrow{e_i}] = x \\
\text{Basic formula} & \gamma & ::= & \theta \,|\, \neg\gamma \,|\, \gamma_1 \wedge \gamma_2 \,|\, \gamma_1 \vee \gamma_2 \\
\text{Temporal formula} & \rho & ::= & \gamma \,|\, \texttt{F}\,\gamma \,|\, \texttt{f}\,\gamma \,|\, \texttt{X}\,(\gamma \to \rho) \\
 & & & |\, \texttt{G}\,(\gamma \to \rho) \,|\, \texttt{g}\,(\gamma \to \rho) \\
\text{Policy} & P & ::= & \forall \overrightarrow{x_i \in D_i}.\rho
\end{array}$$

We write γ to denote basic formulas, which include predicates and propositional connectives. A temporal formula is denoted ρ, whose semantics is defined on an execution trace E, assuming the first state of E is the current state. An execution trace E satisfies $\texttt{F}\gamma$ if γ is true on some future network state in E. We do not have the $\texttt{U}$ operator in LTL but introduce two special operators $\texttt{f}$ and $\texttt{g}$ to specify properties that should hold during the traversal of the current packet in the network. More concretely, $\texttt{f}\,\gamma$ is the short-hand for $(\texttt{loc} \neq \mathsf{Drop} \wedge \texttt{loc} \neq \mathsf{Exit})\,\texttt{U}\gamma$. E satisfies $\texttt{f}\,\gamma$ if γ holds on some future network state in the current packet's traversal. $\texttt{g}\gamma$ is the short-hand for $\gamma\texttt{U}\,(\texttt{loc} = \mathsf{Drop} \vee \texttt{loc} = \mathsf{Exit})$. It is true on E if γ is true on every network state in the current packet's traverse. Nested temporal formulas is only allowed in the restricted forms: $\texttt{X}(\gamma \to \rho)$, $\texttt{G}(\gamma \to \rho)$ and $\texttt{g}(\gamma \to \rho)$, where $\texttt{X}(\gamma \to \rho)$ states that starting on the next state γ is true entails ρ is true, and $\texttt{G}(\gamma \to \rho)$ ($\texttt{g}(\gamma \to \rho)$, resp.) intuitively asserts that whenever γ holds (during the traversal of the current packet, resp.) ρ should also hold. A network policy is a closed temporal formula, universally quantified at the outermost layer. A network N satisfies a policy P, denoted $N \models P$ iff for all execution E starting from an initial network state of N, E satisfies P. Formally definitions are in Appendix B.

The predicates here are customized to one-packet model in that a packet field name uniquely identifies the packet in the network state. By prefixing each packet field with a packet ID (i.e., $id.f$), we can express (equivalent) policies for networks with packet interleaving.

An example. We show the specification of the dynamic service chaining policy in §2. More examples can be found in §7. For space constraint, we consider a sub-policy of the dynamic service chaining policy: if a host is detected suspicious by Light IPS then all of its future packets should be directed to Heavy IPS. Suppose Light IPS keeps an internal state table named *susp* which counts bad connection numbers from each host, and a host is suspicious when the count is larger than 10. The top-level structure of the policy is $\forall x.\texttt{G}(\gamma(x) \to \rho(x))$, which specifies that for all host x whenever $\gamma(x)$ holds $\rho(x)$ should also hold. Here, $\gamma(x) = (\texttt{src} = x \wedge susp[x] > 10)$ specifies that the *susp* count of x is larger than 10 and the current packet is from x; $\rho(x)$ specifies that whenever a packet is sent from x, it will eventually reach H (i.e. the location of Heavy IPS) before being dropped or exiting the network. We can specify this policy as follows.

$$\begin{aligned}
\forall x \in Dept. \quad & \texttt{G}(\texttt{src} = x \wedge susp[x] > 10 \to \\
& \quad \texttt{G}(\texttt{src} = x \to \texttt{f}(\texttt{loc} = H)))
\end{aligned}$$

Model equivalence. We write $N \models^{\infty} P$ to denote that a network N satisfies policy P in the packet interleaving model; we use $N \models^{one} P$ to denote that N satisfies P in the one-packet model. A policy P^{one} specified for the one-packet model can be translated to a policy for the packet-interleaving model, denoted P^{∞}. For example, the translated isolation polity in Appendix F is $\forall id.\texttt{G}(id.loc = A \to ((id.loc \neq B)\texttt{U}(id.loc = \mathsf{Drop} \vee id.loc = \mathsf{Exit}))$. We prove the following theorem stating that checking a number of policies is not affected by ignoring the packet interleaving (Appendix D).

Corollary 2 *For all order-preserving networks N, $N \models^{\infty} P^{\infty}$ if and only if $N \models^{one} P^{one}$, if P is the isolation, tag preservation, or tag-based isolation policy.*

6 NetSMC Checking Algorithms

We view the verification of policies on a network as a model checking problem. Our choice of the policy language in the

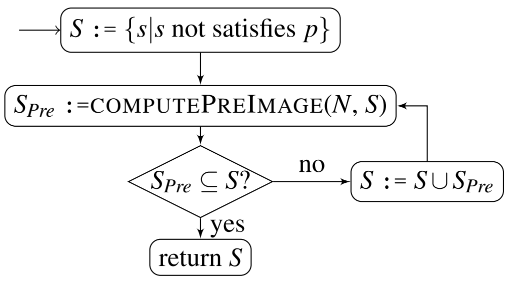

$S := \{s | s$ not satisfies $p\}$
S_{Pre} :=COMPUTEPREIMAGE(N, S)
$S_{Pre} \subseteq S$?
no
$S := S \cup S_{Pre}$
yes
return S

Figure 5: Generic SMC algorithm workflow to check $\mathtt{G}\, p$

restricted LTL form allows us to use the more efficient CTL model checking algorithm. In addition, to handle the large state space we adopt the symbolic model checking (SMC) framework for CTL. In this section, we first provide some background of SMC to highlight key challenges in implementing an SMC tool, and then discuss our approaches to address those challenges.

6.1 Background on SMC

Symbolic model checking is a verification technique that has been proven to be effective and efficient in many domains [14, 35]. Given a system model N and a policy P to be checked, a generic SMC framework computes the set of system states S that violates P (i.e. the set of states satisfying $\neg P$). Then the algorithm checks whether an initial state is in the set S. If so, a violation of the policy P is found; otherwise, P is verified.

To compute the set of states S satisfying $\neg P$, an SMC algorithm computes the set of states that satisfies each sub-formula of $\neg P$ bottom-up, following the structure of $\neg P$. As an example, Figure 5 shows the generic SMC algorithm to compute the set of states that violates $\mathtt{G}\, p$, i.e., from those states there exists an execution trace of the system that violates p.

Initially, the algorithm computes the set of states violating the sub-formula p. Then it repeatedly adds states that can reach some state violating p. In each iteration of the loop, the algorithm computes the set of states S_{Pre} that can transition to a state in S in one step (i.e., $S_{Pre} = \{s | \exists s' \in S. s \rightarrow s'\}$, called the pre-image of S). The algorithm converges when every state in S_{Pre} is contained in S, and then returns the desired set.

To enable efficient SMC, we need a succinct symbolic encoding for a large set of states (e.g. S and S_{Pre}) while supporting efficient computation over them as shown in pre-image computation and subset checking. In the following, we present key components of our algorithm to address these challenges.

6.2 Symbolic Network States in EFO

To illustrate the intuition of our symbolic encodings, consider the firewall example in Fig. 3, where we are interested in checking whether packets from external networks can reach the internal network. Based on the firewall model, a packet from the outside is only allowed to go through if the tests in the second rule return true. The tests return true for all network states where an entry with value 1 exists in the table `Trust`. That is, the network states of interest can be encoded as $\exists x, y. Trust[x, y] = 1$. Thanks to our policy specification, any set of states generated in the checking of (violation of) a policy can be encoded in such an existential form (see details later in this section). Therefore, we can use the following fragment of existential first-order logic (EFO) as our symbolic state encoding. The existential quantifications are only at the outermost level (we thus omit quantifiers) and we operate on the inner formulas without quantifiers.

Atomic Predicates	α	$::=$	$x \in D \mid x \neq y$
		$\mid$	$loc = x \mid f = x \mid T[\overrightarrow{x_i}] = y$
Clauses	β	$::=$	$\bigwedge_i \alpha_i$
State Formulas	ϕ	$::=$	$\bigvee_i \beta_i$

A set of network states is encoded using a state formula, written ϕ, in a DNF form. We say that a network state (lp, Δ) is encoded by ϕ, if there exists a substitution of concrete values for all free variables in ϕ such that (lp, Δ) satisfies ϕ under that substitution. We write $Sat(\phi)$ to denote the set of network states encoded by ϕ. The atomic predicates, α, include membership predicate, inequality check, and test for fields, location and state tables. For the firewall example above, the state formula ϕ is $(Trust[x, y] = 1)$, encoding all network states where the firewall has an entry in $Trust$ with value 1. As we show in the following, the encoding of EFO enables efficient computation of key components in SMC.

6.3 Computing Pre-Image

Next, we describe how to compute the state formula of the pre-image of a symbolic state S, denoted $Pre(S)$, (i.e. COMPUTEPREIMAGE). That is, given a state formula ϕ, we need to compute the state formula ϕ_{Pre}, such that $Sat(\phi_{Pre}) = Pre(Sat(\phi))$. We develop an algorithm that directly generates ϕ_{Pre} by transforming ϕ based on the network model.

Notation. Before explaining the algorithm, we define some auxiliary notations. Without loss of generality, we assume that for each clause β in ϕ, each field f appears at most once (any formula can be rewritten to this form). We write $var(f, \beta)$ to denote the variable being compared to f in β. That is, $var(f, \beta) = x$ if $f = x$ appears in β. If f does not appear in β, $var(f, \beta)$ returns a fresh variable. We write $\beta \backslash_\alpha$ to denote the formula resulted from removing the clause α from β.

Top-level algorithm. Our pre-image computing algorithm (shown in Alg. 1) considers all three types of network transitions (c.f. §4). The top-level function COMPUTEPREIMAGE takes as inputs the network model and the state formula ϕ and returns ϕ_{Pre}. The loop (lines 2-5) goes over every rule in every network function to generate a pre-image that could reach ϕ using that rule. Function COMPUTEPRERULE accounts for the network transitions under NF processing; Function COMPUTELINK on line 6 computes the pre-image for link traversal; Function COMPUTELASTPKT on line 7 computes the pre-image when ϕ represents the state where a new packet

Algorithm 1 Computing the pre-image of a state formula.

```
function COMPUTEPREIMAGE($N$, $\phi$)
    for all NF in the network do
        $(L, \overrightarrow{T}, R) \leftarrow$ NF
        $\phi_{\mathtt{NF}} := \bigvee_{l \in L, r \in R} \bigvee_{\beta \in \phi_r} ((\mathtt{loc} = l) \wedge \beta)$
        where $\phi_r :=$ COMPUTEPRERULE($r$, $\phi$)
    $\phi_{link} :=$ COMPUTELINK($\phi$)
    $\phi_{pkt} :=$ COMPUTELASTPKT($\phi$)
    return $\bigvee_{\mathtt{NF}} \phi_{\mathtt{NF}} \vee \phi_{pkt} \vee \phi_{link}$
function COMPUTEPRERULE($r$, $\phi$)
    $t \Rightarrow c \leftarrow r$
    $\phi_c :=$ COMPUTEPRECMD($c$, $\phi$)
    return $\bigvee_{\beta \in \phi_c} (\bigwedge_{at \in t} trans(at) \wedge \beta)$
function COMPUTEPRECMD($c$, $\phi$)
    match $c$ with
        | $a \Rightarrow$ return COMPUTEPREACTION($a$, $\phi$)
        | $u \Rightarrow$ return COMPUTEPREUPDATE($u$, $\phi$)
        | $c_1, c_2 \Rightarrow \phi_2 :=$ COMPUTEPRECMD($c_2$, $\phi$)
            return COMPUTEPRECMD($c_1$, $\phi_2$)
function COMPUTELASTPKT($\phi$)
    $\phi'$:=False
    for all $\beta$ in $\phi$ do
        $\beta' := \beta \backslash_{\mathtt{loc}=var(\mathtt{loc},\beta)}$
        for all $f = x$ in $\beta$ do
            $\beta' := \beta' \backslash_{f=x}$
        for all ingress location $l$ do
            $\beta_1 := (var(\mathtt{loc}, \beta) = l) \wedge \beta' \wedge (\mathtt{loc} = \mathsf{Drop})$
            $\beta_2 := (var(\mathtt{loc}, \beta) = l) \wedge \beta' \wedge (\mathtt{loc} = \mathsf{Exit})$
            $\phi' := \phi' \vee \beta_1 \vee \beta_2$
    return $\phi'$
```

enters the network. The algorithm returns the disjunction of all formulas for each possible transition. Note that the returned state formula is still in EFO, which enables us to only consider EFOs for state containment. Next, we describe two key functions. We omit the third as it is similar.

Packet transitions. Function COMPUTELASTPKT computes the pre-image when a new packet comes to the network. It computes the pre-image of each clause β in ϕ, and returns the disjunction of all computed pre-images. If a network state $((l', pkt'), \Delta)$ is the result of the transition, then the state before this transition has the same state tables but a different packet. Therefore, all constraints on packet fields and locations are removed from β (line 22-24) as they do not apply to the packet in the pre-image. Furthermore, the location of the packet in the pre-image must be either Drop or Exit, and l' must be an ingress location. Thus, constraints $\mathtt{loc} = \mathsf{Drop}$, $\mathtt{loc} = \mathsf{Exit}$ are added, the same for $var(loc, \beta) = l$ for each ingress location l (line 26, 27).

NF transitions. The function COMPUTEPRERULE iteratively computes the pre-image ϕ_c under the actions and updates in r

Algorithm 2 Sub-functions of computing the pre-image.

```
function COMPUTEPREACTION($a$, $\phi$)
    match $a$ with
        | fwd($e$) $\Rightarrow (g_e, x_e) := F(e)$
            return $\bigvee_{\beta \in \phi} \beta \backslash_{\mathtt{loc}} \wedge g_e \wedge (var(\mathtt{loc}, \beta) = x_e)$
        | drop $\Rightarrow$
            return $\bigvee_{\beta \in \phi} \beta \backslash_{\mathtt{loc}} \wedge (var(\mathtt{loc}, \beta) = \mathsf{Drop})$
        | modify($f$, $e$) $\Rightarrow (g_e, x_e) := F(e)$
            return $\bigvee_{\beta \in \phi} \beta \backslash_f \wedge g_e \wedge (var(f, \beta) = x_e)$
function COMPUTEPREUPDATE($u$, $\phi$)
    T[$\overrightarrow{e_i}$]:=$e \leftarrow u$
    $(g_{e_i}, x_{e_i}) := F(e_i)$ for all $e_i$
    $(g_e, x_e) := F(e)$
    $g := \bigwedge_i g_{e_i} \wedge g_e$
    for all $\beta_j$ in $\phi$ do
        $\phi_j :=$ COMPUTECLS($u$, $\beta_i$, $[(g_{e_i}, x_{e_i})]$, $(g_e, x_e)$)
    return $\bigvee_i \bigvee_{\beta \in \phi_i} g \wedge \beta$
function COMPUTECLS($u$, $\beta$, $[(g_{e_i}, x_{e_i})]$, $(g_e, x_e)$)
    let $tList$ be the list of state tests $T(\overrightarrow{x}) = y$ in $\beta$
    match $tList$ with
        | nil $\Rightarrow$ return $\beta$
        | $h$::$hs$ $\Rightarrow \phi_0 =$ COMPUTECLS($u$, $\beta \backslash_h$)
            T[$\overrightarrow{e_i}$]:=$e \leftarrow u$, $(T(\overrightarrow{x}) = y) \leftarrow h$
            $\beta_0 := (\overrightarrow{x} = \overrightarrow{x_{e_i}}) \wedge (y = x_e)$
            $\beta_j := h \wedge (x_j \neq x_{e_j})$ for $j = 1, .., m$
            return $\bigvee_{\beta' \in \phi_0} ((\beta_0 \wedge \beta') \vee \bigvee_j (\beta_j \wedge \beta'))$
```

(line 9 in Alg. 1). The pre-image under rule r is obtained by adding constraints of the tests t in r using the helper function *trans* (not shown due to space) which translates each atomic test into a clause. Key sub-routines are summarized in Alg. 2.

Function COMPUTEPREACTION computes the pre-image of an action a. Consider the case where a is **modify**(f, e). The semantics of **modify** require the value of field f be modified to the value of e. Thus, the algorithm first considers the value returned by the expression e using the helper function F, which returns a clause g_e together with a variable x_e given expression e. The intuitive meaning is that if g_e is satisfied, then the value of e is equal to x_e (F's formal definitions is omitted). As an example for $e = (T[\mathtt{src}, \mathtt{dst}])$, $g_e = (\mathtt{src} = y_1 \wedge \mathtt{dst} = y_2 \wedge T[y_1, y_2] = y_3)$, and $x_e = y_3$. Then the algorithm adds the constraint g_e and $var(f, \beta) = x_e$. Furthermore, since the value for f is modified, the constraints associated with f from β can be removed as they do not apply to the pre-image.

Function COMPUTEPREUPDATE computes the pre-image under an update $T[\overrightarrow{e_i}] := e$; **inc** and **dec** are similar. The function computes the pre-images for each clause β using the sub-procedure COMPUTECLS, and returns the union of them. Function COMPUTECLS recursively enumerates all possible effects of u to β to compute its pre-image. More concretely, the function considers two cases that u may impact a constraint $T(\overrightarrow{x}) = y$ in β: 1) $T(\overrightarrow{x}) = y$ is updated by u, and 2) $T(\overrightarrow{x}) = y$ is not updated by u. In the first case, it must be the cases that $\overrightarrow{x} = \overrightarrow{x_{e_i}}$ and $y = x_e$, where x_{e_i} denotes the value read from e_i (β_0 shown in line 23). In the second case, $x_i \neq x_{e_i}$ for at least one x_i (line 24). We obtain the pre-image of β as a disjunction of the two case shown in line 25.

6.4 Containment of Network States

As shown in Figure 5, we need an efficient approach to check the containment of two sets of network states. Given two state formulas ϕ_1 and ϕ_2, we need to check if $Sat(\phi_1) \subseteq Sat(\phi_2)$. This is equivalent to checking $\exists \overrightarrow{x}.\phi_1 \Rightarrow \exists \overrightarrow{y}.\phi_2$, where $\overrightarrow{x}$ ($\overrightarrow{y}$, resp.) denotes all free variables in ϕ_1 (ϕ_2, resp.). While this can be solved using a general-purpose SMT solver, as we show in the evaluation section, this is quite inefficient.

Instead, we observe that the state containment problem of EFO is a variant of the *query containment* problem well-studied in database theory [11]. In short, query containment aims to determine if the result of a database query q_1 is contained in that of q_2 for all database instances I. To make the connection clearer, consider the state formula $\phi_1 = (\texttt{src} = x \wedge \texttt{dst} = y \wedge Trust[y,x] = 1)$. This formula can be viewed as the following (conjunctive) query on a database with three tables: *src*, *dst* and *Trust*. $q_1(x,y) : -src(x), dst(y), Trust(y,x)$

Each concrete network state can be viewed as a database instance with the schema defined by packet fields and state tables. Furthermore, each state formula ϕ is a union of conjunctive queries, where each clause β in ϕ is a conjunctive query with inequalities between variables.

In database theory, to determine whether a conjunctive query q_1 is contained in another conjunctive query q_2, it is equivalent to checking whether there is a *homomorphism* from q_2 to q_1, i.e. a function h that maps variables in q_2 to variables and constants in q_1, such that for all $R(x_1, x_2, ..)$ in q_2, there is an $R(h(x_1), h(x_2), ..)$ in q_1 [11].

However, there are still a few challenges in applying the algorithm to our problem. First, as shown in [29], when there are inequalities, there may not exist a homomorphism even when ϕ_1 is contained in ϕ_2. For example, $\phi_1 = (x_1 \neq x_3 \wedge T[x_1,x_2] = 1 \wedge T[x_2,x_3] = 1)$ is contained in $\phi_2 = (y_1 \neq y_2 \wedge T[y_1,y_2] = 1)$, but there is not homomorphism from ϕ_2 to ϕ_1. Second, in query containment problem a variable ranges over a continuous domain (e.g. rational numbers) [29], while in network verification a variable can only take discrete values such as IP addresses. As a result again, there may not exist a homomorphism, even if a set of states encoded in ϕ_1 is contained in the set of states encoded in ϕ_2, E.g., $\phi_1 = (x \in \{0\} \wedge y \in \{0\} \wedge T_1[x] = 1 \wedge T_1[y] = 0)$ is contained in $\phi_2 = (T_2[z] = 0)$ since no states are encoded by ϕ_1, but no homomorphism exists.

To address the first challenge, we break each clause in ϕ_1 into *atomic clauses*, which has been shown to handle inequalities [29]. We call a clause β an atomic clause w.r.t. a state formula ϕ_2, if all variables in β are distinct, and for all variables x in β and y in ϕ_2, the domain of x is either contained in or disjointed with the domain of y. For the first example above, ϕ_1 can be break into the following three atomic clauses, namely, $\beta_1 = (x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_1 \neq x_3 \wedge T[x_1,x_2] = 1 \wedge T[x_2,x_3] = 1)$, $\beta_2 = (x_1 \neq x_2 \wedge T[x_1,x_2] = 1 \wedge T[x_2,x_2] = 1)$, and $\beta_3 = (x_1 \neq x_3 \wedge T[x_1,x_1] = 1 \wedge T[x_1,x_3] = 1)$. We see that there is a homomorphism from ϕ_2 to β_3. For the second challenge, we check for emptiness of clauses, and show that given a state formula ϕ_2 and an atomic clause β w.r.t. ϕ_2, $Sat(\beta) \subseteq Sat(\phi_2)$ if and only if there is a homomorphism from some $\beta' \in \phi_2$ to β, or $Sat(\beta)$ is empty. Now we can verify the containment in the second example above. We obtain our algorithm of checking containment by putting these two pieces together (Alg. 3).

Algorithm 3 Checking containment

```
1: function CHECKCMT($\phi_1$, $\phi_2$)
2:     for all $\beta$ in $\phi_1$ do
3:         let $[\beta_0, .., \beta_k]$ be the set of atomic clauses w.r.t. $\phi_2$ obtained from $\beta$
4:         for all $i = 0$ to $k$ do
5:             if ISEMPTY($\beta_i$) then continue
6:             if there is no homomorphism from $\beta'$ to $\beta_i$ for all $\beta' \in \phi_2$ then
7:                 return False
8:     return True
```

We prove the correctness of our algorithm w.r.t. the one-packet semantics: if NetSMC says policy verified, then all possible executions of the network satisfy the policy; and if NetSMC says policy violated, then there exists an execution that violates the policy. Formally:

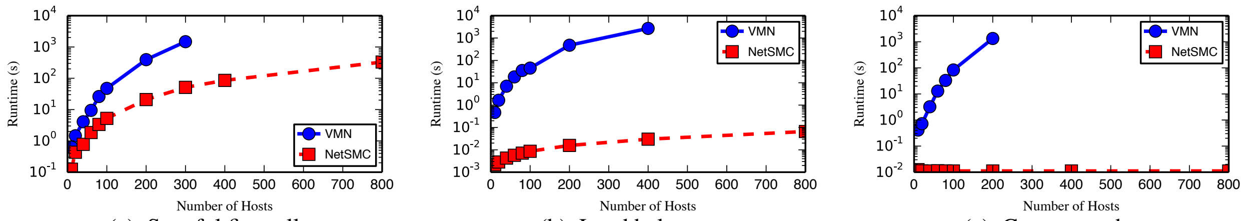
Theorem 3 (Correctness) *Given a stateful network N and a policy P, NetSMC returns True if and only if N satisfies the policy P under the one-packet model.*

The above theorem uses our one-packet semantics. Combined with theorems like Lemma 1 , the verification results of our tool on a large set of practical scenarios are correct w.r.t. the general packet interleaving semantic model as well.

7 Evaluation

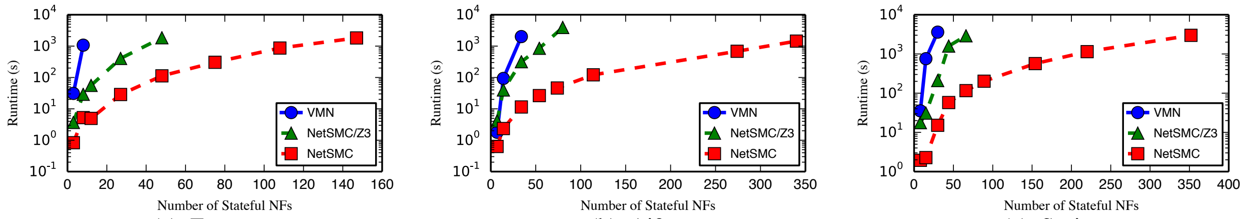
We implement a prototype tool NetSMC in Python based on the algorithms above. We evaluate NetSMC and show that:

- NetSMC can scale to large-size networks and is orders of magnitude more efficient than existing approaches (§7.1);
- Our custom algorithm on containment checking in NetSMC is effective and is 42 times more efficient than naive approaches based on general-purpose solvers (§7.1);
- NetSMC can check a wide range of network policies in various practical network scenarios, which can not be easily supported in alternative tools (§7.2).



(a) Stateful firewall (b) Load balancer (c) Content cache

Figure 6: Scalability with NF complexity



(a) Fattree (b) Ai3 (c) Sprint

Figure 7: Scalability with network complexity.

Evaluation setup: We ran NetSMC on a server with 20 cores (2.8GHz) and 128GB RAM. We first evaluate NetSMC's scalability by varying the complexity of NFs, topologies, and policies (§7.1). For comparison, we use the open-source implementation of VMN [41], a state-of-the-art stateful network verification tool. We also demonstrate the effectiveness and expressiveness in a range of network scenarios using real NFs based on emulation in Cloudlab [43] (§7.2). We use pf-Sense [3] as stateful firewalls and NATs, HAProxy [2] as load balancers. For the case study that required dynamic rule installation (e.g., path pinning), we used the Mininet-based emulation with POX [34] as the SDN controller. We ensure high-fidelity of NetSMC NF models using Alembic [38], which can automatically synthesize NF models from NF implementations. For NFs that are not readily available from Alembic (e.g., POX programs), we manually translate NF programs into equivalent NetSMC models.

7.1 Scalability

NF complexity. Stateful NFs may implement complex functionalities using multiple configuration rules. To evaluate the scalability of NetSMC w.r.t. the complexity of NFs, we consider three types of stateful NFs: (1) a stateful firewall, (2) a load balancer, and (3) a content cache. To create NF configurations with varying complexity, we connect n hosts and n servers to each NF, and for each pair of hosts and servers, we add a rule to the NF. For example, for the stateful firewall, we add rules to limit access from servers to hosts.

Fig. 6 shows the runtime on verifying isolation of a server to a host. NetSMC is orders of magnitude faster than VMN on all tested NFs. Particularly, for the stateful firewall experiment, VMN takes 1477 seconds to verify the policy with 300 hosts, while NetSMC only takes 51 seconds (28×). In the load balancer experiment, VMN takes 2693 seconds with 400 hosts while NetSMC only takes 0.03 seconds. We observe similar speedup in the cache experiment.

Topology complexity. We consider the fattree [4] topology and Ai3 and Sprint, from Topology Zoo [30]. For fattree, we create a range of topologies by varying the number of ports per switch. For Ai3 and Sprint, we systematically extend each switch with multiple switches to generate topologies with varying size. For each topology with n switches, we add additional $2n/3$ stateful NFs where each switch is attached to at most one stateful NF. We use each tool to verify the isolation policy of two hosts in each network. Since VMN critically relies on the slicing technique, we slice the flow-space of all tested networks before applying both tools.

Fig. 7 shows the runtime of verification tools w.r.t. the number of *stateful* NFs in the network. We make the following observations. First, NetSMC is at least two orders of magnitude faster than VMN. Specifically, VMN spends 1072 seconds on the fattree network with 8 stateful NFs, while NetSMC only uses 5 seconds (200× faster). Furthermore, VMN cannot scale to larger networks within 12 hours, while NetSMC can successfully verify the desired policy for networks with 147 stateful NFs in half an hour. For Ai3 and Sprint network, we see similar performance speedup. For example, VMN uses 2011 seconds on the Ai3 network with 34 stateful NFs while NetSMC only uses 11 seconds (175×).

Effectiveness of customized algorithm. To evaluate the benefit of our custom algorithms, we consider an alternative approach by using Z3 to solve the containment problem of network states in NetSMC (shown as NetSMC/Z3 in Fig. 7). We observe that our custom algorithm on containment checking significantly improves the scalability. When using Z3 to check containment, the tool uses 1844 seconds to verify the policy for the fattree network with 48 stateful NFs, which is 16× slower than our custom approach. On Ai3 and Sprint, we measured 42× and 25× speedup respectively.

Policy complexity. To evaluate the scalability of NetSMC w.r.t. the complexity of the policy to be checked, we use NetSMC to check a range of service chaining policies with varied number of NFs on the chain. Since VMN's implementation does not support this type of policy, we consider the variant of NetSMC that uses Z3 for containment checking for comparison. Fig. 8 plots the results. First, we observe that NetSMC can scale up to reasonably large policies. Particularly, NetSMC can check the service chaining policy with 20 NFs in 20 minutes. Second, we observe again that our custom algorithm on containment checking significantly improves the performance: on 12 NFs, our custom model checking algorithm is 23× faster than the Z3 variant.

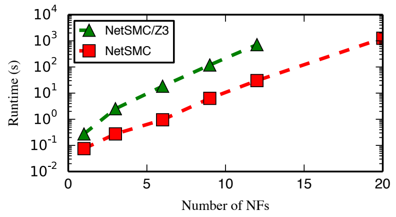


Figure 8: Scalability with policy complexity.

Comparison with general-purpose model checkers. To evaluate the benefit of our custom symbolic model checking algorithm, we further compare NetSMC with a classical BDD-based symbolic model checker NuSMV [12] and a SMT-based model checker Cubicle [15]. Since NuSMV cannot effectively model the state tables using small BDD structures, we model state tables with fixed sizes in NuSMV. We repeat the stateful firewall experiment as described above. With table size 16, NuSMV takes 1163 seconds on verify the reachability policy, while NetSMC only uses 0.015 seconds without size constraint on the state tables. NetSMC is 750X faster than Cubicle. The result confirms that our encoding of symbolic states and custom algorithms for stateful networks are more efficient than general-purpose encodings and tools.

7.2 Effectiveness and Expressiveness

Red-blue team exercise: To validate the effectiveness of NetSMC, we conduct a red-blue team exercise in a range of network scenarios using real NFs. In each scenario, the red team (Author 2 and Author 3) set up a network with intended policies in CloudLab and then delele/modify NF rules (which are kept secret from the blue team) to introduce misconfiguration. The blue team (Author 1) uses NetSMC to check intended policies on the network, so as to identify and fix the misconfiguration.

• **Blocking hosts behind NAT [19]:** The red team sets up a network with two subnetworks (N1 and N2) and two NFs using pfSense with the intended policy to block a host h1 in N1 from reaching another host h2 in N2. However, using NetSMC, the blue team identifies a violation that packets from h1 can still reach h2. The root cause is that the red team mistakenly adds a NAT rule in the first NF such that h1's address is translated and bypassing the firewall rule installed on the second NF blocking h1's address. The blue team fixes this misconfiguration by adding the firewall rule on the first NF and NetSMC verifies the policy.

• **Opposite rules in firewalls:** The red team sets up a network with two subnetworks (N1 and N2) and two stateful firewalls in each subnetwork (fw1 in N1 and fw2 in N2) to protect the subnetworks. The intended policy is to allow N1's packet to reach N2. The blue team uses NetSMC to check this policy and find a violation that packets from N1 is allowed by fw1 but denied at fw2. The blue team fixes the misconfiguration by removing the rule blocking N1 on fw2. Using the new model after the fix, NetSMC successfully verifies the policy.

• **Consistent load balancing:** The red team configures HAProxy to enforce the policy that packets from the same host should always be load balanced to the same server. Using NetSMC, the blue team finds a violation where one flow is sent to server 1 while another is sent to server 2. Checking the configuration, the blue team identifies that HAProxy is misconfigured in the "round robin" mode thus violating the policy. The blue team then reconfigures the LB in the "Source" mode. NetSMC successfully verifies the desired policy.

• **Path pinning:** The red team sets up the network scenario in Fig. 1d with two hosts for the Department and Internet. To enforce the path pinning policy, the red team uses a POX controller to program the forwarding rules on s1. The blue team uses NetSMC to check the path pinning policy, which finds a violation where the first packet from Department is sent to FW1 but the return packet is sent to FW2. The root cause is that the controller mistakenly installed a wrong rule on the switch for the return packets. The blue team fixes the problem by using consistent rules on the controller. Then, NetSMC then successfully verifies the policy.

Policy expressiveness. We show a wide range of policies that

	Time	
Policy & network scenario	Verification	Bug find
Conditional reach.: A stateful firewall with ACL rules.	0.06s	0.03s
Data isol. [41]: A content cache with a client and a server.	2.23s	0.0007s
Pipeline [41]: A stateful firewall with two hosts and servers.	0.001s	0.0006s
Flow affinity: As described in Fig. 1b.	0.19s	0.04s
Dynamic service chaining: As described in Fig. 1c.	0.1s	0.008s
Reachability: Two cascaded NATs [1] (Outside can reach the inside server).	0.001s	0.005s
Tag-based isol.: Network as in Fig. 1c. (A packet labeled by a specific tag should not pass a specific MB).	0.04s	0.94s
Tag preservation: Network as in Fig. 1c. (A packet's tag labeled by a MB should be not be modified).	0.03s	0.98s
NAT consistency: If a NAT modifies a packet's port then all future packets in the flow should have the same port.	0.09s	0.078s

Table 2: Example policies supported by NetSMC.

can be specified and checked using NetSMC, summarized in Table 2. For each case, we simulate networks in NetSMC as described in the table and introduce misconfiguration by deleting/modifying rules in NFs. NetSMC identifies the misconfiguration in all cases and can verify all policies after fixing the bugs. We report the time of verifying the policy or finding bugs. We can see that NetSMC significantly expands the scope of efficiently verifiable network policies. Today's stateless verification tools cannot model any network scenarios considered in the table, and existing stateful network verification tools [5,41] cannot specify some of the policies (summarized in Appendix D).

8 Limitation and Discussion

One-packet model: NetSMC is built on top of the one-packet model, and thus may not find violations caused by packet interleaving. For instance, in the second example of the red-blue team exercise, if fw1 drops all packets from N1 once receiving packets from N2, while fw2 allows packets from N1 to reach N2 after seeing packets from N2 to N1, then NetSMC declares packets from N1 cannot reach N2. However, the following violating trace is missed by the one-packet model: first, a packet p1 from N1 passes fw1, then fw2 processes packet p2 from N2 to N1. Next, fw2 allows p1 to reach N2.

NF Model: NetSMC only supports header matching, state table checking, and simple counting; more complex operations, such as computing average values, are not supported.

Policy: Our policy language cannot express policies needing arbitrary quantification, nesting of temporal operators, or generic until path formulas. For instance, the policy that "at some time in the future, a packet from h1 to h2 is delivered", $\texttt{F}(loc = h1 \rightarrow \texttt{f}(loc = h2))$, is beyond our scope.

Network failures: Currently NetSMC does not model network failures directly and can only check policies in the presence of failure by enumerating each failure scenario and run NetSMC in each case, which may not be efficient.

9 Related Work

Our stateful network model is motivated by existing work on network modeling and programming languages [6, 7, 21, 28, 36, 37, 52]. Our NF model shares key characteristics with the models in NetEgg [52] and SNAP [7].

There is a rich body of work for testing and verifying forwarding behaviors in stateless networks [23, 25–27, 31, 32, 46, 49, 50, 53, 53, 54]. While those work can efficiently check a number of policies such as reachability and loop freedom, it is nontrivial to extend those work to support stateful data planes, which are the target of our work.

For example, Header Space Analysis (HSA) [26] models each packet as a point in the high-dimension space of packet headers and each switch as a transfer function from a subspace into another. Based on symbolic reasoning of the transfer functions, HSA can efficiently verify policies such as reachability and loop freedom. Adapting HSA for stateful network verification would need to introduce some notion of state to the transfer function, which would require a complete redesign of the verification algorithm.

Veriflow [27] uses an alternative "trie" like encoding and focuses on checking policies incrementally when network configurations change. Whenever a rule change occurs on a switch, Veriflow computes the packet space that is influenced by the change, and only applies verification to the delta part. Again, adding state to the trie structure and its associated algorithms is non-trivial.

NoD [31] is based on a generic Datalog framework to check reachability policies, where both networks and policies are encoded in Datalog. NoD can potentially be extended to model stateful network functions. However, Datalog is limited in its expressive power in terms of network policies; temporal properties such as flow affinity and dynamic service chaining cannot be easily specified. It is also unclear if such a stateful extension (if exists) scales.

Our work is closely related to recent efforts on stateful data plane testing and verification. Buzz [18] and SymNet [45] generate test cases for stateful networks based on symbolic execution. VMN [41] verifies isolation properties based on SMT encodings. Alpernas et al. present abstractions to check isolation properties [5]. Those projects, except VMN, only support a subset of our policies. We cannot express generic policies involving past operations; while VMN cannot express some policies NetSMC supports. Our approach is also different in the network model and we build a highly custom symbolic model checker to improve the efficiency.

There are several complementary proposals on verifying control planes: Batfish [20], ERA [17], ARC [22] and Minesweeper [9] analyze routing control planes. NICE [10], VeriCon [8], SDNRacer [16], FlowLog [39] and Kuai [33] target SDN controllers. Work on verifying firewalls [24, 40, 51, 55] can handle statefulness in firewalls, but it is not clear whether those techniques can generalize to handle more expressive network functions and policies.

10 Conclusions

This paper explores a different design space in building efficient verification tools for stateful networks. We identify key domain-specific insights to define a compact model of stateful networks, customize policy specifications, and develop efficient custom symbolic model checking algorithms for verification. We implement NetSMC and show that it achieves orders of magnitude speedup compared to alternative approaches, while supporting a wide range of policies.

Acknowledgements: This work is supported in part by the National Science Foundation via grant CNS1513961 and CNS1552481, and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] What is Double NAT? https://kb.netgear.com/30186/What-is-Double-NAT.

[2] haproxy. https://www.haproxy.org/.

[3] pfSense. https://www.pfsense.org/.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, 2008.

[5] Kalev Alpernas, Roman Manevich, Aurojit Panda, Mooly Sagiv, Scott Shenker, Sharon Shoham, and Yaron Velner. Abstract Interpretation of Stateful Networks. *arXiv preprint arXiv:1708.05904*, 2017.

[6] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 2014.

[7] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, 2016.

[8] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 31. ACM, 2014.

[9] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, 2017.

[10] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, Jennifer Rexford, et al. A nice way to test openflow applications. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[11] Ashok K Chandra and Philip M Merlin. Optimal Implementation of Conjunctive Queries in Relational Data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*. ACM, 1977.

[12] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, 2002.

[13] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, Berlin, Heidelberg, 1982. Springer-Verlag.

[14] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[15] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems . In *Proceedgins of International Conference on Computer-Aided Verification (CAV)*, 2012.

[16] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. SDNRacer: concurrency analysis for software-defined networks. In *ACM SIGPLAN Notices*, volume 51, pages 402–415. ACM, 2016.

[17] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[18] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. Buzz: Testing context-dependent policies in stateful networks. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[19] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, 2014.

[20] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D Millstein. A General Approach to Network Configuration Analysis. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI*, 2015.

[21] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David

Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291. ACM, 2011.

[22] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[23] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time Network Verification Using Atoms. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[24] Alan Jeffrey and Taghrid Samak. Model checking firewall policy configurations. In *Proceedings of IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, 2009.

[25] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[26] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

[27] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proceedings of 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[28] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[29] Anthony Klug. On conjunctive queries containing inequalities. *Journal of the ACM (JACM)*, 35(1):146–160, 1988.

[30] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.

[31] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking Beliefs in Dynamic Networks. In *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[32] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, 2011.

[33] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for software-defined networks. In *Proceedgins of Formal Methods in Computer-Aided Design (FMCAD)*, 2014.

[34] J Mccauley. Pox: A python-based openflow controller, 2014.

[35] Kenneth L McMillan. Symbolic Model Checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[36] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. *ACM SIGPLAN Notices*, 47(1):217–230, 2012.

[37] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[38] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. Alembic: Automated model inference for stateful network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[39] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[40] Timothy Nelson, Christopher Barratt, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The Margrave Tool for Firewall Analysis. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, 2010.

[41] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying Reachability in Networks with Mutable Datapaths. In *Proceedings of 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[42] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977.

[43] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[44] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, 2012.

[45] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, 2016.

[46] Brendan Tschaen, Ying Zhang, Theo Benson, Sujata Benerjee, JK Lee, and Joon-Myung Kang. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *Proceedings of IEEE SDN-NFV Conference*, 2016.

[47] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. Some complexity results for stateful network verification. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2016.

[48] Wenfei Wu, Ying Zhang, and Sujata Banerjee. Automatic synthesis of nf models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, 2016.

[49] Geoffrey G Xie, Jibin Zhan, David A Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On Static Reachability Analysis of IP Networks. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2015.

[50] H. Yang and S. S. Lam. Real-Time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, April 2016.

[51] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and Prasant Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, 2006.

[52] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. Scenario-based Programming for SDN Policies. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, CoNEXT '15, 2015.

[53] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. *IEEE/ACM Trans. Netw.*, 22(2):554–566, April 2014.

[54] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[55] Shuyuan Zhang, Abdulrahman Mahmoud, Sharad Malik, and Sanjai Narain. Verification and synthesis of firewalls using SAT and QBF. In *Proceedings of 20th IEEE International Conference on Network Protocols (ICNP)*, 2012.

Appendix

A Semantics of Stateful Network Model

The top-level transition rules are of the form: $s \to s'$. We use a number of auxiliary transitions summarized below:

$[\![e]\!]_{lp;\Delta} = v$	Expression evaluation
$[\![at]\!]_{lp;\Delta} = b$	Atomic test evaluation
$[\![t]\!]_{lp;\Delta} = b$	Test evaluation
$c; lp; \Delta \to lp'; \Delta'$	Command evaluation
$r; lp; \Delta \to lp'; \Delta'$	Rule evaluation
$\texttt{NF}; lp; \Delta \to lp'; \Delta'$	NF evaluation

The semantic rules of our stateful network model are summarized in Figure 10, Figure 11, and Figure 9.

$\boxed{s \to s'}$

$$\begin{array}{l}\textsc{Net-Trans-NF}\\ \dfrac{\begin{array}{c} NF = (L,\_,\_) \qquad l \in L \\ lp = (l,pkt) \qquad NF;(l,pkt);\Delta \to (l',pkt');\Delta' \end{array}}{(lp,\Delta) \to ((l',pkt'),\Delta')}\end{array}$$

$$\frac{topo(l) = l'}{((l,pkt),\Delta) \to ((l',pkt),\Delta)}\ \textsc{Net-Link}$$

$$\frac{l = \mathsf{Drop/Exit} \quad l' \in IngressLocs}{((l,\_),\Delta) \to ((l',pkt),\Delta)}\ \textsc{Net-Packet}$$

Figure 9: One-packet semantics of network execution.

$$\boxed{[\![e]\!]_{lp;\Delta} = v}$$

$$\textsc{PickFrom}\quad \frac{v \in D}{[\![\textbf{pickFrom}(D)]\!]_{lp;\Delta} = v} \qquad \textsc{Field}\quad \frac{lp = (\_, pkt) \qquad pkt.f = v}{[\![f]\!]_{lp;\Delta} = v}$$

$$\textsc{State-Table}\quad \frac{\forall i, [\![e_i]\!]_{lp;\Delta} = v_i \qquad \delta_T = \Delta(T) \qquad \delta_T(\overrightarrow{v_i}) = v}{[\![T[\overrightarrow{i = e_i}]]\!]_{lp;\Delta} = v}$$

$$\boxed{[\![at]\!]_{lp;\Delta} = b}$$

$$\textsc{Test-Loc-True}\quad \frac{lp = (l, \_)}{[\![\texttt{loc} = l]\!]_{lp;\Delta} = \mathsf{True}} \qquad \textsc{Test-Loc-False}\quad \frac{lp = (l', \_) \qquad l \neq l'}{[\![\texttt{loc} = l]\!]_{lp;\Delta} = \mathsf{False}}$$

$$\textsc{Test-Field-True}\quad \frac{[\![f]\!]_{lp,\Delta} \in D}{[\![f \in D]\!]_{lp;\Delta} = \mathsf{True}} \qquad \textsc{Test-Field-False}\quad \frac{[\![f]\!]_{lp,\Delta} \notin D}{[\![f \in D]\!]_{lp;\Delta} = \mathsf{False}}$$

$$\textsc{Test-Table-True}\quad \frac{[\![T[\overrightarrow{i = e_i}]]\!]_{lp;\Delta} = v}{[\![T[\overrightarrow{i = e_i}] = v]\!]_{lp;\Delta} = \mathsf{True}}$$

$$\textsc{Test-Table-False}\quad \frac{[\![T[\overrightarrow{i = e_i}]]\!]_{lp;\Delta} = u \qquad u \neq v}{[\![T[\overrightarrow{i = e_i}] = v]\!]_{lp;\Delta} = \mathsf{False}} \qquad \textsc{Test-Neg}\quad \frac{}{[\![\neg at]\!]_{lp;\Delta} = \neg [\![at]\!]_{lp;\Delta}}$$

$$\boxed{[\![t]\!]_{lp;\Delta} = b}$$

$$\textsc{Test-Sequence-True}\quad \frac{[\![t_1]\!]_{lp;\Delta} = \mathsf{True}}{[\![t_1, t_2]\!]_{lp;\Delta} = [\![t_2]\!]_{lp;\Delta}} \qquad \textsc{Test-Sequence-True}\quad \frac{[\![t_1]\!]_{lp;\Delta} = \mathsf{False}}{[\![t_1, t_2]\!]_{lp;\Delta} = \mathsf{False}}$$

Figure 10: Semantics of network function.

B Policy Semantics

We define the semantics of open formulas ρ over a tuple $(\mathtt{V}, E, N)$ (Figure 2), where $\mathtt{V}$ is the valuation function of all free variables appearing in ρ, E is an infinite network execution trace (i.e., a sequence of network states), and N is the network configuration. We write E_i to denote the i-th state in E and $E[i..]$ as the suffix of E starting at the i-th state, Similar to the semantic rules, we omit the network configuration N for simplicity of presentation. $(\mathtt{V}, E)$ satisfying ρ, written $(\mathtt{V}, E), \models \rho$ is formally defined below.

- $(\mathtt{V}, E) \models \gamma$ iff $E_0 \models \gamma$ in the standard way.
- $(\mathtt{V}, E) \models \mathtt{G}\rho$ iff $(\mathtt{V}, E[i..]) \models \rho$ for all $i \geq 0$
- $(\mathtt{V}, E) \models \mathtt{g}\rho$ iff there is an $i \geq 0$ where $E_i = (lp, \Delta)$, s.t. 1) $lp = (\mathsf{Drop}, \_)$ or $lp = (\mathsf{Exit}, \_)$ and 2) for all $j < i$, $(\mathtt{V}, E[j..]) \models \rho$.

$$\boxed{c; lp; \Delta \to lp'; \Delta'}$$

$$\textsc{Update}\quad \frac{\begin{array}{c}[\![e]\!]_{lp;\Delta} = v \qquad \forall i, [\![e_i]\!]_{lp;\Delta} = v_i \\ \delta_T = \Delta(T) \qquad \delta'_T = \delta_T[\overrightarrow{v_i} \mapsto v] \qquad \Delta' = \Delta[T \mapsto \delta'_T]\end{array}}{T[\overrightarrow{i = e_i}] := e; lp; \Delta \to lp; \Delta'}$$

$$\textsc{Inc}\quad \frac{\begin{array}{c}\forall i, [\![e_i]\!]_{lp;\Delta} = v_i \qquad \delta_T = \Delta(T) \\ \delta'_T = \delta_T[\overrightarrow{v_i} \mapsto \delta_T(\overrightarrow{v_i}) + v] \qquad \Delta' = \Delta[T \mapsto \delta'_T]\end{array}}{\textbf{inc}(T[\overrightarrow{i = e_i}], v); lp; \Delta \to lp; \Delta'}$$

$$\textsc{Dec}\quad \frac{\begin{array}{c}\forall i, [\![e_i]\!]_{lp;\Delta} = v_i \qquad \delta_T = \Delta(T) \\ \delta'_T = \delta_T[\overrightarrow{v_i} \mapsto \delta_T(\overrightarrow{v_i}) - v] \qquad \Delta' = \Delta[T \mapsto \delta'_T]\end{array}}{\textbf{dec}(T[\overrightarrow{i = e_i}], v); lp; \Delta \to lp; \Delta'}$$

$$\textsc{Action-Forward}\quad \frac{lp = (l, pkt) \qquad [\![e]\!]_{lp;\Delta} = v \qquad lp' = (v, pkt)}{\textbf{fwd}(e); lp; \Delta \to lp'; \Delta}$$

$$\textsc{Action-Drop}\quad \frac{}{\textbf{drop}; (l, pkt); \Delta \to (\mathsf{Drop}, pkt); \Delta}$$

$$\textsc{Action-Modify}\quad \frac{lp = (l, pkt) \qquad [\![e]\!]_{lp;\Delta} = v \qquad pkt' = pkt[f \mapsto v]}{\textbf{modify}(f, e); lp; \Delta \to (l, pkt'); \Delta}$$

$$\textsc{Sequence}\quad \frac{c_1; lp; \Delta \to lp'; \Delta'}{(c_1; c_2); lp; \Delta \to c_2; lp'; \Delta'}$$

$$\boxed{r; lp; \Delta \to lp'; \Delta'}$$

$$\textsc{Rule}\quad \frac{[\![t]\!]_{lp;\Delta} = \mathsf{True} \qquad c; lp; \Delta \to lp'; \Delta'}{t \Rightarrow c; lp; \Delta \to lp'; \Delta'}$$

$$\boxed{\mathtt{NF}; lp; \Delta \to lp'; \Delta'}$$

$$\textsc{NF}\quad \frac{\begin{array}{c}lp = (l, pkt) \\ l \in L \qquad r_j \in R \qquad r_j; lp; \Delta \to lp'; \Delta'\end{array}}{(L, \overrightarrow{T_j}, R); lp; \Delta \to lp'; \Delta'}$$

Figure 11: Semantics of network function (cont.).

- $(\mathtt{V}, E) \models \mathtt{F}\gamma$ iff $(\mathtt{V}, E[i..]) \models \gamma$ for some i
- $(\mathtt{V}, E) \models \mathtt{f}\gamma$ iff there is an $i \geq 0$ s.t. 1) $E_i \models \gamma$ and 2) for all $j < i$ where $E[j] = (lp, \Delta)$, $lp \neq (\mathsf{Drop}, \_)$ and $lp \neq (\mathsf{Exit}, \_)$
- $(\mathtt{V}, E) \models \mathtt{X}\rho$ iff $(\mathtt{V}, E[1..]) \models \rho$.

C One packet vs. packet interleaving model

We identify sufficient conditions under which our one packet model is equivalent to the interleaving model considered in [41] w.r.t. a set of policies. First, we give the formal semantics of the interleaving model; then we show sufficient conditions under which our one packet model is equivalent to the interleaving model, followed by our proofs establishing the equivalence.

Executions in packet interleaving network model: A network state in the packet interleaving model is a pair $(\mathtt{Q},\Delta)$, where $\mathtt{Q}$ is the set of packets buffered at each network location and Δ is the valuation function of state tables as usual. The network state of the one-packet model is a special case where $|\mathtt{Q}| = 1$. We use $ID(p)$ to denote the ID of packet p and $ID(lp)$ has its natural meaning. The top-level transition rules of the packet interleaving model are given in Fig. 12, where the three rules generalized the corresponding rules in the one packet model. Each transition in the transition system of the packet-interleaving model is of the form of $s \xRightarrow{lp/lp'}_{NF} s'$, where lp/lp' denote the (located) packet that is processed by the transition and lp (lp', resp.) denote the packet before (after, resp.) the transition, NF denotes the NF that processes that packet or *null* (which we typically ignore) if the packet is transmitted by a link or injected into the network.

A network execution trace E is a sequence of transitions $s_0 \Rightarrow s_1 \Rightarrow \cdots \Rightarrow s_n$. A *closed network execution trace* is a finite network execution where the both initial state s_0 and the final state s_n contain no packets buffered at any location except for Drop and Exit. We use $\mathsf{E}^{\infty}(N)$ to denote the set of all closed network execution traces of a given network N under the packet interleaving semantics; similarly, we use $\mathsf{E}^{\mathsf{one}}(N)$ to denote the set of all closed network execution traces under the one-packet model. We assume that there is no indefinite loops for any packet traversal; transient loops are allowed to appear.

Formalization of processing-order preserving: We give necessary definitions first before we formalize the processing-order preserving condition.

Definition 4 (Non-conflict) *Given NF, lp_1, lp_2, we say that lp_1 is non-conflicting with lp_2 at NF if $\forall \Delta_0,\Delta_1,\Delta_2,lp'_1,lp'_2$ s.t. $NF;lp_1;\Delta_0 \to lp'_1;\Delta_1$ and $NF;lp_2;\Delta_1 \to lp'_2;\Delta_2$, $\exists \Delta'_1$ s.t. $NF;lp_2;\Delta_0 \to lp'_2;\Delta'_1$ and $NF;lp_1;\Delta'_1 \to lp'_1;\Delta_2$.*

We define processing-order preserving based on packet orders. Particularly, given a set of packets, a packet order $\prec$ is a strict total order on the IDs of the packets. Given two located packets lp_1 and lp_2, we denote $lp_1 \prec lp_2$ if $ID(lp_1) \prec ID(lp_2)$.

Definition 5 (Processing-order preserving trace) *Given a network N, a network execution trace $E = s_0 \Rightarrow \ldots \Rightarrow s_n$ (under the packet interleaving model) in $\mathsf{E}^{\infty}(N)$, a packet order $\prec$, E is* processing-order preserving *(or order-preserving in short) under $\prec$ if $\forall lp_1, lp_2, NF$ such that $lp_2 \prec lp_1$ and lp_1 is conflicting with lp_2 at NF, there do not exist transitions $s_j \xRightarrow{lp_1/lp'_1}_{NF} s_{j+1}$ and $s_k \xRightarrow{lp_2/lp'_2}_{NF} s_{k+1}$ in E where $k \geq j+1$.*

$$\boxed{(\mathtt{Q},\Delta) \xRightarrow{lp/lp'}_{NF} (\mathtt{Q},\Delta')}$$

NET-TRANS-NF

$$\frac{\begin{array}{c} NF = (L,\_,\_) \\ l \in L \qquad pkt \in \mathtt{Q}(l) \qquad NF;(l,pkt);\Delta \to (l',pkt');\Delta' \\ \mathtt{Q}' = \mathtt{Q}[l' \mapsto (\mathtt{Q}(l') \cup \{pkt'\})][l \mapsto (\mathtt{Q}(l) \setminus \{pkt\})] \end{array}}{(\mathtt{Q},\Delta) \xRightarrow{(l,pkt)/(l',pkt')}_{NF} (\mathtt{Q}',\Delta')}$$

NET-LINK

$$\frac{\begin{array}{c} topo(l) = l' \qquad pkt \in \mathtt{Q}(l) \\ \mathtt{Q}' = \mathtt{Q}[l' \mapsto (\mathtt{Q}(l') \cup \{pkt\})][l \mapsto (\mathtt{Q}(l) \setminus \{pkt\})] \end{array}}{(\mathtt{Q},\Delta) \xRightarrow{(l,pkt)/(l',pkt')}_{null} (\mathtt{Q}',\Delta)}$$

NET-TRANS-IN

$$\frac{l \in \mathsf{IngressLocs} \qquad \mathtt{Q}' = \mathtt{Q}[l \mapsto (\mathtt{Q}(l) \cup \{pkt\})]}{(\mathtt{Q},\Delta) \xRightarrow{-/(l,pkt)}_{null} (\mathtt{Q}',\Delta)}$$

Figure 12: Semantics of packet interleaving execution.

We call a network N *processing-order preserving* if there is a packet order $\prec$ such that for all $E \in \mathsf{E}^{\infty}(N)$, E is order preserving under $\prec$. Some example processing-order preserving networks includes: (1) a network with no stateful NFs; (2) a network where any single packet only traverses one stateful NF; (3) a network with connection-based NFs where packets in a connection are delivered in order.

Equivalence between one packet model and packet interleaving model: Given a closed network execution trace $E = s_0 \xRightarrow{lp_1/lp_2} \cdots \xRightarrow{lp_{n-1}/lp_n} s_n$ in $\mathsf{E}^{\infty}(N)$ of a network N and an ID id, we call the per-packet trace of id, denoted $E|_{id}$, as the sequence $[lp_{i_1}/lp_{i_1+1} \ldots lp_{i_k}/lp_{i_k+1}]$ obtained by projecting all lp_i/lp_{i+1} pairs (except for the first pair corresponding to the NET-TRANS-IN rule) with the ID id from the sequence $[lp_1/lp_2 \ldots lp_{n-1}/lp_n]$. We define the per-packet trace for executions in the one-packet model similarly.

Lemma 1 *Given an order-preserving network N, $\forall E \in \mathsf{E}^{\infty}(N)$, $\exists E' \in \mathsf{E}^{\mathsf{one}}(N)$, s.t. $\forall id$, $E|_{id} = E'|_{id}$.*

PROOF. Let $\prec$ be the packet order satisfying the order-preserving of N. Suppose $E = s_0 \xRightarrow{lp_0/lp'_0} s_1 \xRightarrow{lp_1/lp'_1}, \ldots, \xRightarrow{lp_{n-1}/lp'_{n-1}} s_n$. We define the out-of-order index I of E as number of disordered transitions w.r.t. $\prec$. Formally $I(E,\prec) = \sum_{i<n} |\{j | j < i, lp_i \prec lp_j\}|$.

We prove this lemma by induction over the out-of-order index of E.

Base case: Since $I(E,\prec) = 0$, we know that $lp_i \not\prec lp_{i-1}$ for all

$i \geq 1$, i.e., $ID(lp_{i-1}) = ID(lp_i)$ or $ID(lp_{i-1}) \prec ID(lp_i)$. Since E is closed, there is an execution trace $(lp_0, \Delta_0) \to (lp_1, \Delta_1) \to \ldots \to (lp_n, \Delta_n)$ (if some lp_i is empty, simply ignore that state) in the one-packet model where Δ_i is the table valuation in s_i, and the per-packet trace for all packets are the same.
Inductive case: We have the inductive hypothesis: For all $E \in \mathsf{E}^{\infty}(N)$ where $I(E, \prec) \leq k$, there is an execution trace $E' \in \mathsf{E}^{\mathsf{one}}(N)$ s.t. $E|_{id} = E'|_{id}$. Now consider the case where $I(E, \prec) = k+1$. Since $I(E, \prec) > 0$, there exists $i > 0$ such that $s_{i-1} \xrightarrow{lp_{i-1}/lp'_{i-1}} s_i \xrightarrow{lp_i/lp'_i} s_{i+1}$ and $lp_i \prec lp_{i-1}$. We claim that there must exists a network state s'_i such that $s_{i-1} \xrightarrow{lp_i/lp'_i} s'_i \xrightarrow{lp_{i-1}/lp'_{i-1}} s_{i+1}$. This is easy to see when the transition $s_i \xrightarrow{lp_i/lp'_i} s_{i+1}$ is obtained from rule NET-LINK or NET-TRANS-IN. Suppose the transition is from NET-TRANS-NF and the NF that processes lp_i on s_i is NF. If transition $s_{i-1} \xrightarrow{lp_{i-1}/lp'_{i-1}} s_i$ is not from NET-TRANS-NF or does not correspond to the processing NF NF, the claim is also obvious. When both transitions correspond to the process of the packets on NF, from the processing-order preserving definition, lp_i must be non-conflicting with lp_{i-1} at NF. Thus from Lemma 6 the claim is still true. By swapping the processing of lp_{i-1} and lp_i we obtain an order-preserving execution trace E'' from E such that $E''|_{id} = E|_{id}$ for all id and $I(E'', \prec) = k$. From the inductive hypothesis, there is an execution trace $E' \in \mathsf{E}^{\mathsf{one}}(N)$ s.t. $E'|_{id} = E''|_{id} = E|_{id}$ for all id. □

Lemma 6 *For all network N, located packets lp_1, lp'_1, lp_2, lp'_2, $NF \in N$, and network states s_1, s_2, s_3 of N, if $s_1 \xrightarrow{lp_1/lp'_1}_{NF} s_2 \xrightarrow{lp_2/lp'_2}_{NF} s_3$ and lp_1 is non-conflicting with lp_2 at NF, then $\exists s'_2$ s.t. $s_1 \xrightarrow{lp_2/lp_2}_{NF} s'_2 \xrightarrow{lp_1/lp'_1}_{NF} s_3$.*

PROOF. Immediate from definition of non-conflicting.

D Soundness and Completeness of Checking Per-Packet-Trace Policies

From Lemma 1, we can show that checking a range of policies involving per-packet traces is equivalent between the packet interleaving network model and the one packet model provided that network is processing-order preserving.

Per-packet-trace policies. We define per-packet-trace policies (for the one-packet model) as follows.

$$\varphi^{one} ::= \forall \overrightarrow{x_i \in D_i}.\mathtt{G}(\gamma_1 \to \mathtt{g}\gamma_2) \mid \forall \overrightarrow{x_i \in D_i}.\mathtt{G}(\gamma_1 \to \gamma_2)$$

A translation function, denoted $\langle\cdot\rangle$, turns a formula for the one-packet model to a corresponding formula for the packet-interleaving model. It is defined as follows, which essentially introduces a packet ID into the formula. We only show selected rules and the rest are inductively defined over the structure of the formula.

$$\begin{aligned}
&\langle \forall \overrightarrow{x_i \in D_i}.\mathtt{G}(\gamma_1 \to \mathtt{g}\gamma_2)\rangle = \\
&\quad \forall id.\forall \overrightarrow{x_i \in D_i}.\mathtt{G}(\langle\gamma_1\rangle_{id} \to \\
&\qquad \langle\gamma_2\rangle_{id}\,\mathtt{U}\,(id.\mathtt{loc} = \mathsf{Drop} \vee id.\mathtt{loc} = \mathsf{Exit})) \\
&\langle f = x\rangle_{id} = id.f = x \\
&\langle \mathtt{loc} = x\rangle_{id} = id.\mathtt{loc} = x \\
&\langle f\rangle_{id} = id.f \quad \langle x\rangle_{id} = x \quad \langle v\rangle_{id} = v
\end{aligned}$$

We can prove the following theorem.

Theorem 7 *For all order-preserving network N, $N \models^{one} \varphi^{one}$ if and only if $N \models^{\infty} \langle\varphi^{one}\rangle$.*

PROOF. We only prove the case for the first form of φ^{one}. Proofs for the second is very similar. Let $\varphi^{one} = \forall \overrightarrow{x_i \in D_i}.\mathtt{G}(\gamma_1 \to \mathtt{g}\gamma_2)$.

($\leftarrow$) Suppose $N \not\models^{one} \varphi^{one}$. By the definition of the policy, there exists $\overrightarrow{v_i}$ for $\overrightarrow{x_i}$ and an execution trace $E = (lp_0, \Delta_0) \to \ldots \to (lp_n, \Delta_n) \in \mathsf{E}^{\mathsf{one}}(N)$, and some i, j, such that $0 \leq i \leq j \leq n$, $(V, E[i..]) \models \gamma_1$, and $(V, E[j..]) \not\models \gamma_2$, where $V = [\overrightarrow{x_i \mapsto v_i}]$. and for all k, l_k s.t. $i < k < j$ and $lp_k = (l_k, \_)$, $l_k \notin \{\mathsf{Drop}, \mathsf{Exit}\}$.

By the semantics of one-packet model, $\forall m$ s.t. $i < m < j$, $ID(lp_m) = ID(lp_i) = ID(lp_j) = pid$.

We can then construct an execution trace $E' \in \mathsf{E}^{\infty}(N)$ by injecting one packet at a time to simulate E, and the only trivial difference between E and E' is that incoming packets takes an extra step, rather than being enqueued right after the previous packet exits.

It's straightforward to show that $(V[id \mapsto pid], E'[i..]) \models \langle\gamma_1\rangle_{id}$ and $(V[id \mapsto pid], E'[j..]) \not\models \langle\gamma_2\rangle_{id}$, and $i < k < j$ and $(V[id \mapsto pid], E'[k..]) \models id.\mathtt{loc} \neq \mathsf{Drop} \wedge id.\mathtt{loc} \neq \mathsf{Exit}$, since there is only one packet with ID pid in the packet queue.

By the formula semantics, $(V[id \mapsto pid], E'[i..]) \not\models \langle \mathtt{g}\,\gamma_2\rangle_{id}$. It follows that $(\emptyset, E') \not\models \langle\varphi^{one}\rangle$, so $N \not\models^{\infty} \langle\varphi^{one}\rangle$, which contradicts with our assumption.

($\rightarrow$) Suppose $N \not\models^{\infty} \langle\varphi^{one}\rangle$. By the definition of the policy, there exists $\overrightarrow{v_i}$ for $\overrightarrow{x_i}$, pid for id and an execution trace $E = s_0 \Rightarrow \ldots \Rightarrow s_n \in \mathsf{E}^{\infty}(N)$, and some i, j, such that $0 \leq i \leq j \leq n$, $(V, E[i..]) \models \langle\gamma_1\rangle_{id}$, and $(V, E[j..]) \not\models \langle\gamma_2\rangle_{id}$, where $V = [\overrightarrow{x_i \mapsto v_i}, id \mapsto pid]$. And for all k s.t. $i < k < j$, $(V, E[k..]) \models id.\mathtt{loc} \neq \mathsf{Drop} \wedge id.\mathtt{loc} \neq \mathsf{Exit}$.

By the network semantics, the located packet with ID pid in E_i must have entered the queue at some point. Let $i' \leq i$ be the index of the first such state in E, such that $E = \cdots \xrightarrow{lp_{i'}/lp'_{i'}} s_{i'} \cdots$. Similarly, we identify $j' \leq j$ such that j' is the first state where the located packet in E_j has arrived at the packet queue. W.o.l.g. $E = \cdots \xrightarrow{lp_{i'}/lp'_{i'}} s_{i'} \cdots \xrightarrow{lp_{j'}/lp'_{j'}} s_{j'} \cdots$. It is straightforward that for all k s.t. $i' < k < j'$, $(V, E[k..]) \models id.\mathtt{loc} \neq \mathsf{Drop} \wedge id.\mathtt{loc} \neq \mathsf{Exit}$ (Dropped or Exited packets cannot come back with the same pid). Note that $ID(lp'_{i'}) = ID(lp'_{j'}) = pid$.

By Lemma 1, there is an execution trace $E' \in \mathsf{E}^{\mathsf{one}}(N)$ where $E'|_{pid} = E|_{pid}$. By the semantics of the one packet model, $E' = \cdots E'' \cdots$, where $E''|_{pid} = E|_{pid}$ and E'' contains processing of only packets with ID pid. So, $(V\backslash id, E') \not\models \gamma_1 \to \mathtt{g}\,\gamma_2$.

It follows that $N \not\models^{one} \varphi^{one}$, which contradicts with our assumption. □

Corollary 8 *NetSMC is sound and complete w.r.t. the packet interleaving model when checking isolation, tag preservation, tag-based isolation policies.*

This follows from Theorem 7 and Theorem 3.

Comparison with VMN: VMN accepts policies of the form: $\forall n, p : G\neg(rcv(d,n,p) \wedge predicate(p))$ (see Appendix B.2 in [41]) where $predicate(p)$ may include past events. We can express all the policies in VMN that do not have past events in $predicate(p)$ (i.e., a basic formula). Isolation, tag-based isolation, and tag preservation fall into this category. Further, in the three cases. NetSMC is sound and complete w.r.t. VMN.

The conditional isolation in VMN cannot be expressed in our policy language, since it involves past events and requires a generic until operator, which we do not support. On the other hand, NAT consistency, conditional reachability, flow affinity, double NAT, and dynamic service chaining cannot be expressed in VMN. In these cases, NetSMC is sound and conditional complete w.r.t. the one-packet model.

E Policy Translation to CTL

Our choice of the policy language as a subset of LTL allows us to translate policies to equivalent forms in CTL. Thus we can use the model checking algorithm of CTL to check those policies. To simplify our notation in the proof, we write $(V,s) \models \rho$ to denote that $\forall E$ s.t. $E_0 = s$, $(V,E) \models \rho$.

Theorem 9 *For all network N and policy P, $N \models P$ if and only if $N \models P_{CTL}$.*

PROOF. Suppose $P = \forall \overrightarrow{x_i \in D_i}.\rho$ and thus $P_{CTL} = \forall \overrightarrow{x_i \in D_i}.\rho_{CTL}$. (If) Since $N \models P_{CTL}$, by the definition over the structure of ρ $\forall V, s$ s.t. $V[x_i] \in D_i$ and s is an initial state of N, $(V,s) \models \rho_{CTL}$. By Lemma 10, $(V,s) \models \rho$. Thus $N \models P$. (Only if) Since $N \models P$, by the definition, $\forall V, s$ s.t. $V[x_i] \in D_i$ and s is an initial state of N, $(V,s) \models \rho$. By Lemma 10, $(V,s) \models \rho_{CTL}$. Thus $N \models P_{CTL}$. □

Lemma 10 *For all network N, temporal formula ρ, valuation function V for variables appeared in ρ, state s in N, $(V,s) \models \rho$ if and only if $(V,s) \models \rho_{CTL}$.*

PROOF. Proof by induction over the structure of ρ.
Case 1 $\rho = \gamma$: This is immediate from the definition.
Case 2 $\rho = \mathtt{F}\gamma$: (If) Since $(V,s) \models \mathtt{AF}\gamma$, for all execution trace E where $E_0 = s$, there is some $i \geq 0$ s.t. $E_i \models \gamma$. Thus, $(V,s) \models \mathtt{F}\gamma$. (Only if) Since $(V,s) \models \mathtt{F}\gamma$, for all execution trace E where $E_0 = s$, there is some $i \geq 0$ s.t. $E_i \models \gamma$. Thus, $(V,s) \models \mathtt{AF}\gamma$.
Case 3 $\rho = \mathtt{f}\gamma$: (If) Since $(V,s) \models \mathtt{A}((\mathtt{loc} \neq \mathsf{Drop} \wedge \mathtt{loc} \neq \mathsf{Exit})\mathtt{U}\gamma)$, for all execution trace E where $E_0 = s$, there is some $i \geq 0$ s.t. $E_i \models \gamma$ and for all $j < i$, the location of E_j is not Drop nor Exit. Thus, $(V,E) \models \mathtt{f}\gamma$. Thus, $(V,s) \models \mathtt{f}\gamma$. (Only if) Since $(V,s) \models \mathtt{f}\gamma$, for all execution trace E where $E_0 = s$, there is some $i \geq 0$ s.t. $E_i \models \gamma$ and for all $j < i$ the location of E_j is not Drop nor Exit. Thus, $(V,s) \models \mathtt{A}((\mathtt{loc} \neq \mathsf{Drop} \wedge \mathtt{loc} \neq \mathsf{Exit})\mathtt{U}\gamma)$.
Case 4 $\rho = \mathtt{G}(\gamma \to \rho')$: From Lemma 12, we only need to show for all s', $(V,s') \models \gamma \to \rho'$ if and only if $(V,s') \models \gamma \to \rho_{CTL}$. First, when $(V,s') \not\models \gamma$, we have $(V,s') \models \gamma \to \rho'$ and $(V,s') \models \gamma \to \rho_{CTL}$. Second, when $(V,s') \models \gamma$, from the inductive hypothesis, $(V,s') \models \rho'$ if and only if $(V,s') \models \rho_{CTL}$.
Case 5 $\rho = \mathtt{g}(\gamma \to \rho_1)$: Similar to the proof of Case 4 and use Lemma 11.
Case 6 $\rho = \mathtt{X}(\gamma \to \rho_1)$: Similar to the proof of Case 4 and use Lemma 12. □

Note that the ρ in the following two lemmas are generic temporal formulas, not confined to our policy syntax.

Lemma 11 *For all network N, temporal formula ρ, valuation function V for variables appeared in ρ, if for all state s, $(V,s) \models \rho$ is equivalent to $(V,s) \models \rho_{CTL}$, then for all state s', $(V,s') \models \mathtt{g}\rho$ is equivalent to $(V,s') \models \mathtt{A}((\rho_{CTL})\mathtt{U}(\mathtt{loc} = \mathsf{Drop} \vee \mathtt{loc} = \mathsf{Exit}))$.*

PROOF. (If) By the definition of $(V,s') \models \mathtt{A}((\rho_{CTL})\mathtt{U}(\mathsf{Drop} \vee \mathsf{Exit}))$, for all execution trace E where $E_0 = s'$ and $i \geq 0$ where the location of E_i is Drop or Exit, $(V,E_j) \models \rho_{CTL}$ for all $j < i$. By the assumption, $(V,E_j) \models \rho$. Thus, $(V,E[j..]) \models \rho$. Therefore, $(V,E) \models \mathtt{g}\rho$. Thus, $(V,s') \models \mathtt{g}\rho$.
(Only if) Suppose $(V,s') \not\models \mathtt{A}((\rho_{CTL})\mathtt{U}(\mathtt{loc} = \mathsf{Drop} \vee \mathtt{loc} = \mathsf{Exit}))$. By its definition, there is a execution trace E and $i \geq 0$ such that $E_0 = s'$ and $(V,E_i) \not\models \rho_{CTL}$ and for all $j \leq i$, the location of E_j is not Drop nor Exit. Since $(V,E_i) \not\models \rho_{CTL}$, we have $(V,E_i) \not\models \rho$; i.e. there is a execution trace E' where $E'_0 = E_i$ such that $(V,E') \not\models \rho$. Consider the execution trace $E'' = E[0..i-1] + +E'$. Note that $E''[i..] = E'$, thus $(V,E''[i..]) \not\models \rho$. In addition, for all $j \leq i$, the location of E''_j is not Drop nor Exit. Thus $(V,E'') \not\models \mathtt{g}\rho$, which contradicts to that $(V,s') \models \mathtt{g}\rho$. □

Lemma 12 *For all network N, temporal formula ρ, valuation function V for variables appeared in ρ, if for all state s, $(V,s) \models \rho$ is equivalent to $(V,s) \models \rho_{CTL}$, then for all state s', $(V,s') \models \mathtt{G}\rho$ is equivalent to $(V,s') \models \mathtt{AG}(\rho_{CTL})$ and $(V,s') \models \mathtt{X}\rho$ is equivalent to $(V,s') \models \mathtt{AX}(\rho_{CTL})$.*

PROOF. Similar to above. □

F Formal Specification of Example Policies

Isolation: Packets sent from host A can never reach host B:

$$\mathtt{G}(\mathtt{loc} = A \to \mathtt{g}(\mathtt{loc} \neq B))$$

Tag-based isolation: Packets tagged with T cannot reach the middlebox *MB*.

$$\mathtt{G}(\mathtt{tag} = T \rightarrow \mathtt{g}(\mathtt{loc} \neq MB))$$

Tag preservation: The tag T should not be modified by NFs.

$$\mathtt{G}(\mathtt{tag} = T \rightarrow \mathtt{g}(\mathtt{tag} = T))$$

NAT consistency: If a NAT modifies a packet's port then all future packets in the flow should have the same port.

$$\begin{aligned} \forall i.\quad & \mathtt{G}(\mathtt{flow} = i \wedge \mathtt{loc} = NAT\_IN \rightarrow \\ & \quad \mathtt{X}(\mathtt{srcport} = p \wedge \mathtt{loc} = NAT\_OUT \rightarrow \\ & \quad\quad \mathtt{G}(\mathtt{flow} = i \wedge \mathtt{loc} = NAT\_IN \rightarrow \\ & \quad\quad\quad \mathtt{X}(\mathtt{srcport} = p)))) \end{aligned}$$

Conditional reachability: Whenever a packet sent from a host A reaches host B, all packets sent from host B afterwards can reach A.

$$\mathtt{G}(\mathtt{loc} = A \rightarrow g(\mathtt{loc} = B \rightarrow \mathtt{G}(\mathtt{loc} = B \rightarrow f(\mathtt{loc} = A))))$$

Flow affinity: Packets in the same flow should be load-balanced ot the same server.

$$\begin{aligned} \forall i.\quad & \mathtt{G}(\mathtt{flow} = i \wedge \mathtt{loc} = S_1 \rightarrow \\ & \quad \mathtt{G}(\mathtt{loc} = C \wedge \mathtt{flow} = i \rightarrow \\ & \quad\quad \mathtt{f}(\mathtt{loc} = S_1))) \end{aligned}$$

Double NAT: Packets from the outside can always reach the inside (despite two NATs).

$$\mathtt{G}(\mathtt{loc} = OUT \rightarrow \mathtt{f}(\mathtt{loc} = IN))$$

Dynamic service chaining: After a host from *Dept* has sent more than 10 suspicious packets, all of its packets should pass heavy IPS H.

$$\begin{aligned} \forall x \in Dept.\quad & \mathtt{G}(\mathtt{src} = x \wedge susp[x] > 10 \rightarrow \\ & \quad \mathtt{G}(\mathtt{src} = x \rightarrow \mathtt{f}(\mathtt{loc} = H))) \end{aligned}$$