NetSMC: A Symbolic Model Checker for Stateful Network Verification

YIFEI YUAN LIMIN JIA VYAS SEKAR
CARNEGIE MELLON UNIVERSITY

Abstract

Formal verification of computer networks is critical for ensuring that network configurations correctly implement operators' high-level intent. In recent years, tremendous progress has been made in network verification techniques. However, achieving a suitable balance between efficiency and expressiveness remains a fundamental and open challenge in verifying stateful networks, which contain complex network functions such as stateful firewalls and load balancers. In this paper, we approach the stateful network verification problem from first principles and develop a verification tool using symbolic model checking. By taking advantage of the key characteristics of stateful network functions, we are able to succinctly represent a large set of network states symbolically using a fragment of existential first order logic. This symbolic representation allows us to leverage known theoretical results to develop a more efficient model checking algorithm and allows us to verify an expressive set of network policies including flow affinity and dynamic service chaining. Our evaluation results show that our tool, NetSMC, achieves orders of magnitude speedup compared to existing approaches and at the same time supports more expressive network policies.

1 INTRODUCTION

Network policies (e.g. reachability, service chaining) are often implemented by a combination of network functions, ranging from simple ones such as switches and routers, to complex *stateful* functions such as stateful firewalls and NATs [35]. Formally verifying whether a network correctly implements those policies is critical for ensuring its availability, security, and safety [17, 19–21, 26, 27, 34, 40].

In recent years, tremendous progress has been made in the field of network verification that target both data plane (e.g. [17, 19–21, 26]) and control plane verification (e.g., [5]) problems to check if the given network or configuration satisfies the operators intent. Most of this body of work focuses on networks with simple *stateless* elements (i.e., with simple switches and forwarding tables with match-action semantics) and shows that even in this case the verification problem has fundamental expressiveness vs. efficiency tradeoffs.

Our focus in this paper is specifically on *verification of stateful data planes*. That is, we consider the problem where the network can contain a number of advanced and *stateful functions* (e.g., stateful firewalls, NATs, proxies, load balancers) and the operators need to verify more *rich and dynamic intents* expressed over these stateful elements.

Given the large state space of stateful networks, those techniques face harder challenges in the tradeoffs between efficiency and expressiveness. As such, there is only a limited exploration of this problem and most efforts resort to incomplete testing approaches [13, 36] while some early efforts tackle a subset of this problem [34]. For example, the best existing approach for stateful network verification, VMN [34], relies on a clever encoding of the verification problem into SMT constraint solving. However, VMN can only verify reachability policies and cannot handle other more complex, yet practical policies such as flow affinity and dynamic service chaining (which we discuss in detail in Sec. 2).

Rather than take the route of finding yet another encoding of the problem to feed into a general-purpose solver like Z3 (e.g., [6, 26, 34]), we ask a different question:

Is it possible to develop a custom verification tool for stateful networks to improve the efficiency and expressiveness?

To this end, we approach the stateful network verification problem from first principles and revisit this problem through the lens of a classical symbolic model checking approach [29]. Now, any such framework requires three main components: (1) an expressive way to specify the policies or invariants to be checked; (2) a concise representation of the state space of the system being modeled; and (3) an efficient algorithm to explore the state space to check if the policies are satisfied.

Having thus recast the stateful data plane verification problem, we identify the foundational challenges in realizing a symbolic model checking framework on each of these fronts. First, today's networks need to enforce a wide spectrum of network policies on a variety of network functions. How to effectively model and specify those network functions and policies while supporting efficient symbolic model checking algorithms is a key challenge. Second, stateful networks often induce a large state space. For example, a typical stateful firewall needs to maintain the set of all legitimate flows (5-tuples). As a result, the entire state space of the firewall can be as large as 2^F, where F is the size of all legitimate flows and can be on the order of thousands for a typical firewall. Therefore, how to succinctly represent a large set of states remains another challenge. Third, we need efficient algorithms to reason about the symbolic representation in the symbolic model checking framework. In particular, how to efficiently compute the transformation and check containment of symbolic states, as two key steps required in symbolic model checking, remains the third challenge.

Seen in a general context, these are indeed open challenges. Fortunately, we show that a number of domain-specific insights on the structure of the stateful networks and the policy intents that we want to verify can enable us to tackle these challenges in our context. Specifically, we make three key design contributions:

- We develop a restrictive, yet expressive model of stateful networks that captures key characteristics of network functions, and propose a policy specification language based on the ACTL [10] that can specify a wide range of practical policies.
- Our network model and policy specification enable the use of simple existential first order logic (EFO) formulas as the symbolic representation of network states.

- We develop efficient algorithms to reason about the symbolic representation. Particularly, as two critical steps required by the model checking framework, we develop custom algorithms for computing the transformation of symbolic network states and extend algorithms of the query containment problem in the database community [8, 23] for checking containment of symbolic states.

We implement NetSMC, a prototype symbolic model checker for stateful networks. We prove the correctness of our algorithms and show that NetSMC achieves orders of magnitude speedup, while at the same time, supporting more expressive network policies compared to existing approaches.

2 MOTIVATION

In this section, we first provide an overview of related work on network data plane verification, then we motivate our work by describing practical network policies that cannot be effectively verified using existing approaches.

2.1 Overview of Network Verification

Based on the capability of network functions, existing network verification techniques can be classified into two categories: (1) ones that can only verify stateless functions (e.g. switches and routers) and (2) ones that can verify both stateless and stateful functions (e.g. stateful firewall and NAT).

Stateless networks verification. There have been several tools for verifying stateless networks; however, it is not straightforward to extend them for verifying policies in stateful networks. For example, Header Space Analysis (HSA) [20] 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 us to add some notion of state to the transfer function, which would require a complete redesign of the verification algorithm.

Veriflow [21] 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 [26] is based on a generic Datalog framework to check reachability policies, where 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; in particular, temporal properties such as flow affinity and dynamic chaining (see next section) cannot be easily specified. In addition, it is unclear if such a stateful extension (if exists) can scale up to large networks and policies.

Stateful network verification. Compared to stateless verification, there is relatively less work in stateful network verification, perhaps due to the seeming intractability [38]. As such many existing tools effectively avoid verification and opt for a testing based approach [13, 36] and sacrifice soundness/completeness in

favor of a more bug-finding like approach. Other related work [33, 41] focuses on specific scenarios for NATs/firewalls and use off-the-shelf solvers; it is unclear if they are extensible or scalable.

VMN [34] is arguably the closest related work which encodes network functions and the policies as constraint formulas solvable by SMT solvers. To achieve efficiency and scalability, VMN sacrifices the expressiveness of verifiable policies and considers a restricted scope of four types of reachability policies: basic reachability (A can directly talk to B), flow reachability (A can talk to B only after B talks to A), data reachability (B's data can be sent to A), and pipeline (a type of packets must traverse a specific middlebox).

2.2 Policies Excluded by Existing Tools

To motivate the need for a new verification approach for stateful networks, we present several practical policies that cannot be handled by existing tools [25].

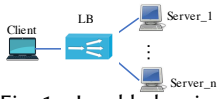


Fig. 1. Load balancing.

Flow affinity. Let us consider a load balancer that distributes traffic among n servers as shown in Fig. 1. 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. Unfortunately, existing tools cannot verify this policy. First, the load balancer needs to maintain internal state (i.e. the server that each host first connects to) in order to correctly forwarding packets, which rule out all the tools for stateless networks. Second, the flow affinity policy cannot be expressed as the reachability policies required by VMN.

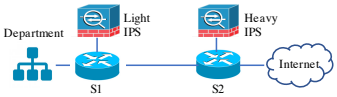


Fig. 2. Multi-stage IPS.

Dynamic service chaining. Fig. 2 shows a multi-stage IPS system consisting of a light IPS and a heavy IPS. Each device in the network is configured such that all traffic from the subnet Department is sent to the light IPS, which performs basic detection such as counting the number of bad connections for each host. Moreover, if a host is detected suspicious by Light IPS (e.g. issuing more than 10 bad connections) at some point, all *future* packets from it should be directed to the heavy IPS for further processing; otherwise its traffic can be directly sent to the Internet. Existing tools cannot verify the dynamic policy for similar reasons as the previous example.

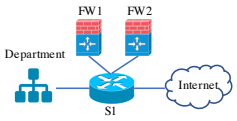


Fig. 3. Multiple stateful firewalls.

Path pinning. Often a network needs to deploy multiple instances of the same middlebox function for better throughput. Consider the network shown in Fig. 3 which is configured to forward traffic between the subnet Department and the Internet to one of the firewalls. The network operator enforces the following path pinning policy: if a packet from H1 in the Department to H2 in the Internet goes through the i-th firewall, then future packets from H2 to H1 should traverse the same firewall. None of the existing tools can verify this policy efficiently because that there is state involved and that the policy cannot be expressed as simple reachability.

In summary, these examples above show that there is much to be done in verifying stateful networks. As we will see in Sec. 7, current tools are also lacking in efficiency and scalability. We see an opportunity in designing an efficient tool to verify a wide range of policies for stateful networks.

3 OVERVIEW OF NETSMC

At a high level, the trajectory of a number of recent network verification efforts, including VMN, has largely been one of identifying efficient encodings of the problem that can subsequently fed into SMT solvers such as Z3. While this has indeed advanced the state of art of network verification substantially, it appears that this approach seems to have hit a fundamental limit in the expressiveness-efficiency tradeoff when it comes to stateful networks.

This and the success of other parallel efforts in developing domain-specific solvers (e.g., HSA, Veriflow) motivate us to revisit this problem from first principles and explore the feasibility of building a custom symbolic model checker for the stateful verification problem. In this section, we provide necessary background on standard symbolic model checking algorithms, then we elaborate on the challenges and our insights specific to the stateful network domain.

3.1 Background on SMC

Symbolic model checking is a classical verification technique that has been proven to be effective and efficient in many domains [10, 29]. Typically, the policies that SMC accepts are temporal logic formulas such as CTL. Given a system model N and a policy P in CTL to be checked, a generic SMC algorithm computes the set S of system states 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 $\neg P$, a SMC algorithm typically computes the set of states that satisfies each sub-formula of $\neg P$ in a bottom-up fashion. As an example, Algorithm 1 shows the generic SMC algorithm for computing states that satisfy $\texttt{EF}(\neg P)$ (i.e. negation of the policy $\texttt{AG}(P)$). Algorithm 1 computes the set of states from which there exists an execution path of the system that violates P.

Initially, the algorithm computes the set of states satisfying the sub-formula $\neg P$ (Line 2). Then it repeatedly adds states that can reach some state satisfying $\neg P$. 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 (computePreImage in Line 6). The algorithm converges when every state in S_{Pre} is contained in S (Line 7), and then returns the desired set.

Algorithm 1 Generic SMC algorithm for $\mathsf{EF}(\neg P)$

```
1: function computeEF(N, ¬P)
2:     S := {s|s satisfies ¬P}
3:     S_Pre := ∅
4:     repeat
5:         S := S ∪ S_Pre
6:         S_Pre := computePreImage(N, S)
7:     until S_Pre ⊆ S
8:     return S
```

3.2 Challenges and Insights

Network model (Sec. 4). We need an expressive yet restrictive model of stateful networks for efficient verification. On one end of the spectrum, we could use a general purpose language (e.g., C in Buzz [13]), but that leads to highly inefficient verification. Motivated by recently proposed network function models [4, 42], the essence of a stateful network function can be modeled as a pair of a set of state tables, indexed by packet header fields, and a set of rules that modify those state tables based on testing (matching) results of the incoming packet. Such restricted formalization enables simple representation of network states and efficient model checking algorithms. Further, we adopt a run-time model that only allows one packet being processed in the network at each state. In effect, we only consider a sequential execution model of networks; the network processes each incoming packet in a sequential manner. This is a standard assumption, especially in network testing [13, 36]. While this model will not reveal concurrency bugs, it is still useful for identifying practical issues, which is already a very hard problem to solve.

Policy specification language (Sec. 5) Policies outlined in Sec. 2 are temporal properties, and therefore CTL and LTL are the natural choices of the policy specification language. It is known that CTL results in more efficient model checking algorithms than LTL [10]; further, it is expressive enough for a wide set of network policies. However, the full fragment of CTL is still difficult to handle. Our insight is that the set of network policies of interest combined with our restricted network model allows us to work within the space of a small fragment ACTL, which includes only all-path temporal connectives, simple equality and set membership checks, and only allows universal quantifiers at the outermost level.

Efficient SMC algorithm (Sec.6). The operations on lines 6 and 7 of Algorithm 1 (i.e., computing states that can reach a set of states and checking containment of sets of states) are typically expensive, and sometimes even undecidable. The symbolic representation of the states dictates the complexity of these two operations. Given our restricted network model and policy specification language, we can succinctly encode a large set of states into a fragment of existential first-order logic (EFO). As an example for the stateful firewall above, the EFO formula $\exists x, y. Trust[x, y] = 1$ represents all network states where there is a legitimate flow between x and y. Thanks to our network model, we are able to design custom algorithms for efficiently computing the pre-image for a set of states encoded in EFO. Finally, we identify the connection between checking containment of sets of states in EFO and conjunctive query containment problem in the

Field Name	f	$\in$	{srcip, dstip, srcport...}
Value	v	$\in$	Int $\cup$ 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	::=	$v \mid f \mid \textbf{pickFrom}(D) \mid T[\overrightarrow{e_i}]$
Atomic Test	at	::=	True $\mid$ loc=$v \mid f \in D \mid T[\overrightarrow{e_i}] = v$
Test	t	::=	$at \mid \neg at \mid t,t$
Update	u	::=	$T[\overrightarrow{e_i}] := e$
Action	a	::=	$\textbf{fwd}(e) \mid \textbf{drop} \mid \textbf{modify}(f, e)$
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	$topo$	$\in$	Loc $\rightarrow$ Loc
Network Config	N	::=	$(topo, [NF_1, .., NF_k])$
Table Valuation	Δ	$\in$	TableNames $\rightarrow \delta_T$
Network State	s	::=	(lp, Δ)

Fig. 4. Syntax of stateful network model.

database community [8, 23]. We are able to adapt existing results from the database literature to efficiently check the termination condition of the SMC algorithms (line 7 of Algorithm 1).

4 STATEFUL NETWORK MODEL

In this section, we formally define the syntax and semantics of our stateful network model and show example encodings of common network functions. Our model is similar to previously proposed models (e.g. [4, 34, 42]) in spirit, but adds useful constructs (e.g. non-deterministic value choice) for modeling practical stateful functions.

4.1 Syntax

Key syntactic constructs of our model is summarized in Figure 4. We write *pkt* to denote network packets, which are records of packet fields. Throughout this paper, we use the vector notation as a shorthand for a list of elements. For instance, $\{\overrightarrow{t_i}\}$ is a shorthand for $\{t_1, \cdots, t_n\}$. Packet field names, denoted f, are drawn from a set of pre-defined names, which includes common field names such as `srcip`, `dstip`, `srcport`, `dstport` 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, which denotes that packets are dropped and Exit, which denotes that packets exit the network. A located packet, denoted *lp*, is a pair of a location and a packet.

We model all devices in the network as *network functions*, denoted *NF*. A network function is a tuple consisting of a set of associated 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 have an empty set of state tables.

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 always return true; tests that check whether a field of the incoming packet matches a specific value; tests that check the current location of the incoming packet; tests that check whether a field value is in a specific domain (e.g. an interval) D; and tests that check if the value of a state table entry matches a constant. 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 to produce and place the outgoing packet, denoted a.

We write e to denote expressions, 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 key-value map, where we write $T[\overrightarrow{e_i}]$ to denote the value in an entry indexed by the key value $\overrightarrow{e_i}$.

A state table entry can be updated. The update $T[\overrightarrow{e_i}] := e$ generates a new table that is the same as the old one except that the value in the entry indexed by the key value $\overrightarrow{e_i}$ is updated to the value of e. Note that we do not allow arithmetic computations of expressions for efficiency of our verification tool. Such a restriction does not prevent us from modeling all the network functions we use in our evaluation. We consider the following actions that can be applied to the incoming packet: forwarding, dropping, and modifying the value of a packet field.

Finally, a network configuration N consists of a set of links, denoted *topo*, and a set of NFs. A network state is a pair of a located packet being processed and a table valuation (Δ), which maps each state table name T to a concrete table δ_T. In fact, $T[\overrightarrow{e_i}]$ refers to the value stored in δ_T. Notice that our model includes only one located packet in each state. We explain our rational in Section 4.3.

4.2 NF Examples

To demonstrate the expressiveness of our model, we show example encodings of several stateful network functions.

Stateful firewall. A stateful firewall protects an internal network by restricting accesses from external hosts. Fig. 5 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 first rule). When

a packet comes from an external host (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 second rule); otherwise the packet is dropped (the third rule).

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

Fig. 5. Stateful firewall.

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. 6 shows a load balancer example for a service with virtual IP address VIP, where we assume that servers are connected with location 1 and client are connected with 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 simply forward the packets as shown in the third rule. Lastly, for traffic going from servers to clients, the load balancer simply modify the destination address of the packet to be 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 => fwd(1);
loc=1 => modify(dst, VIP), fwd(0);
```

Fig. 6. Load balancer.

NAT. A NAT allows hosts in an internal network to share a public IP address `PIP` while maintaining each host's connection to the outside. A NAT typically rewrites both the source port of each outgoing packet and the destination port of packets heading to the internal network. Fig. 7 shows the code for a NAT, with location 0 and 1 connects to the internal and external network respectively. The NAT maintains three state tables: `Port` stores the modified port number for each outgoing flow; `OIP` and `OPort` stores the original IP address and port number for the modified port number respectively. The rules of the NAT are similar to those of a load balancer. The most complex is the second rule, which accounts for the case where an outgoing packet was not assigned a new port. The NAT picks a port from a pool `P` of ports, updates the

three state tables accordingly, and finally modifies the source IP and port of the packet and then sends it to the outside.

```
loc=0, Port[srcip, srcport]!=0 =>
  modify(srcip, PIP),
  modify(srcport, Port[srcip,srcport]),
  fwd(1);
loc=0, Port[srcip, srcport]=0 =>
  Port[srcip, srcport]:=pickFrom(P)
  OIP[Port[srcip, srcport]]:=srcip,
  OPort[Port[srcip, srcport]]:=srcport,
  modify(srcip, PIP),
  modify(srcport, Port[srcip, srcport]),
  fwd(1);
loc=1, dstip=PIP, OIP[dstport]!=0 =>
  modify(dstip, OIP[dstport]),
  modify(dstport, OPort[dstport]),
  fwd(0);
loc=1, dstip=PIP, OIP[dstport]=0 => fwd(0);
loc=1, dstip!=PIP => fwd(0);
```

Fig. 7. NAT.

4.3 Semantics

The semantic rules of the network form the basis of our symbolic model checking algorithm. We first describe the semantics of individual network functions, then, we present the semantics of the entire network. Due to space constraints, we only provide high-level explanation and omit the detailed mathematical formulations.

NF. 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, the atomic test `Trust[dst,src]=1` in the second rule in Fig. 5 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. For instance, the first rule in Fig. 5 updates the state table `Trust` by modifying the entry associated with the source and destination of the incoming packet pkt to `1`, and then forwards the packet to location `1`. In our model, we assume that an NF can nondeterministically apply actions/updates of any matched rule if multiple rules can match an incoming packet. It is trivial to translate other semantics such as the one based on applying the first matched rule into our model.

Stateful Networks. The semantics of the network are defined as a transition system over all network states. We write $(lp, \Delta) \rightarrow_N (lp', \Delta')$ for such transitions. The network configuration N never changes, so we omit them from the rules. As we mentioned in Section 3, we only allow one located packet in the network in each state.

There are three top-level transition rules, shown below. Rule Net-Trans-NF applies when a packet is received by a *NF*. *NF* updates the state tables to a new valuation Δ' and modifies the located packet to lp'. We invoke the network function transition rules described earlier. Rule Net-Link applies when a packet moves from one end of a link to the other based on the topology. Only the location of the located packet is updated. Finally, rule Net-Packet applies to situations where the current packet is dropped or exits the network, a new packet at an ingress location is brought into the network.

$$\text{Net-Trans-NF}\quad \frac{NF = (L, \_, \_) \qquad l \in L \qquad lp = (l, pkt) \qquad NF; (l, pkt); \Delta \rightarrow (l', pkt'); \Delta'}{(lp, \Delta) \rightarrow ((l', pkt'), \Delta')}$$

$$\frac{topo(l) = l'}{((l, pkt), \Delta) \rightarrow ((l', pkt), \Delta)}\ \text{Net-Link} \qquad \frac{l = \mathsf{Drop/Exit} \quad l' \in IngressLocs}{((l, \_), \Delta) \rightarrow ((l', pkt), \Delta)}\ \text{Net-Packet}$$

5 NETWORK POLICIES IN ACTL

As we outlined in Section 3, for efficiency of our model checking algorithm, we use ACTL, a fragment of the computational tree logic (CTL) as our specification language. Our framework can be generalized to LTL as well. In the following, we first present the syntax and semantics of our specification language, then present the formal specification of policies shown in Section 2.

5.1 ACTL

Syntax. The syntax of our specification language for network policies 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). We write γ to denote basic formulas, which include predicates and propositional connectives. A temporal formula, denoted ρ, includes temporal constructs: AG, AF, AU, and AX.

Predicate	θ	::=	$f = x \mid \mathtt{loc} = x \mid T[\overrightarrow{e_i}] = x$
Basic formula	γ	::=	$\theta \mid \neg\gamma \mid \gamma_1 \wedge \gamma_2 \mid \gamma_1 \vee \gamma_2$
Temporal formula	ρ	::=	$\gamma \mid \mathsf{AG}\rho \mid \mathsf{AF}\rho \mid \mathsf{A}(\rho_1 \mathsf{U} \rho_2) \mid \mathsf{AX}\rho$
			$\rho_1 \wedge \rho_2 \mid \rho_1 \vee \rho_2$
Policy	P	::=	$\forall \overrightarrow{x_i \in D_i}.\rho$

Intuitively, $\mathsf{AG}\rho$ holds on the current network state if ρ holds on all states of all possible execution traces of the network. $\mathsf{AF}\rho$ holds on the current network state if on all possible execution traces of the network, ρ eventually holds on a state of that trace. $\mathsf{AX}\rho$ holds on the current network state if on all possible execution traces of the network, ρ holds on the next state of that trace. $\mathsf{A}(\rho_1 \mathsf{U} \rho_2)$ holds on the current network state if on all possible execution traces of the network, ρ_1 holds on every network state until ρ_2 holds on some

network state. Finally, a network policy is a closed temporal formula, universally quantified at the outermost layer.

Semantics. We define the semantics of open formulas ρ over a tuple $(\mathtt{V}, lp, \Delta, N)$, where $\mathtt{V}$ is the valuation function of all free variables appearing in ρ, (lp, Δ) is the network state, and N is the network configuration. Similar to the semantic rules, we omit the network configuration N for simplicity of presentation. We say that the state $(\mathtt{V}, lp, \Delta)$ satisfies ρ, written $(\mathtt{V}, lp, \Delta) \models \rho$. Formally:

- $(\mathtt{V}, lp, \Delta) \models f = x$ iff $pkt.f = \mathtt{V}(x)$;
- $(\mathtt{V}, lp, \Delta) \models \mathtt{AG}\rho$ iff for all execution traces initiated from (lp, Δ), $(\mathtt{V}, lp', \Delta') \models \rho$ for all (lp', Δ') on the trace.
- $(\mathtt{V}, lp, \Delta) \models \mathtt{AF}\rho$ iff for all execution traces initiated from (lp, Δ), $(\mathtt{V}, lp', \Delta') \models \rho$ for some (lp', Δ') on the trace.
- $(\mathtt{V}, lp, \Delta) \models \mathtt{A}(\rho_1 \mathtt{U} \rho_2)$ iff for all execution traces initiated from (lp, Δ), $(\mathtt{V}, lp_j, \Delta_j) \models \rho_2$ for some (lp_j, Δ_j) on the trace, and $(\mathtt{V}, lp_i, \Delta_i) \models \rho_1$ for all $i < j$
- $(\mathtt{V}, lp, \Delta) \models \mathtt{AX}\rho$ iff $(\mathtt{V}, lp', \Delta') \models \rho$ for all network state (lp', Δ') such that $(lp, \Delta) \rightarrow (lp', \Delta')$

The semantics of policies are defined as follows: $(lp, \Delta) \models \forall \overrightarrow{x_i \in D_i}.\rho$ iff for all $\overrightarrow{v_i \in D_i}$, $([\overrightarrow{x_i \mapsto v_i}], lp, \Delta) \models \rho$. A stateful network N satisfies a policy P, denoted $N \models P$ iff for all initial network state (lp_0, Δ_0) of N, $(lp_0, \Delta_0) \models P$.

5.2 Policy Examples

We present the specification of two policies shown in Sec. 2. More examples can be found in Sec. 7.

Flow affinity. Suppose the client's address is A and the location of its interface is C. We consider the case where packets from the client are sent the server S_1. Other cases can be specified similarly.

We specify this policy in a top-down fashion. First, the policy can be specified in the following format: $\mathtt{AG}(\gamma_1 \rightarrow \gamma_2)$, which intuitively means that "it always holds that if γ_1 happens then γ_2 should also happen". Then, we can use γ_1 to specify that a packet from C reaches S_1, and γ_2 to specify that all future packets in the same flow should be sent to S_1. For γ_1, we can use a basic formula $(\mathtt{src} = A \wedge \mathtt{flow} = i \wedge \mathtt{loc} = S_1)$ to specify that a packet with source A reaches location S_1. Here, we use a symbolic value i to specify the flow ID associated with the packet. In practice, however, a 5-tuple can be used to define the flow ID. Next for γ_2, we specify that if a packet with flow ID i is sent from C, then it will eventually reach S_1. Similar to above, we can specify γ_2 as $\mathtt{AG}(\gamma_3 \wedge \gamma_4)$, where $\gamma_3 = (\mathtt{loc} = C \wedge \mathtt{flow} = i)$ specifying that a packet with flow ID i is sent from location C, and $\gamma_4 = \mathtt{A}((\mathtt{loc} \neq \mathtt{Drop} \wedge \mathtt{loc} \neq \mathtt{Exit})\mathtt{U}(\mathtt{loc} = S_1))$ specifying that the packet eventually reaches S_1 before it gets dropped or exits the network. Note that, the use of $\mathtt{AU}$ in γ_4 may not be replaced by $\mathtt{AF}$. For example, $\mathtt{AF}(\mathtt{loc} = S_1)$ specifies that eventually a packet (maybe another one) reaches S_1. Putting together, the flow affinity policy can be specified as follows.

$$\forall i.\quad \mathtt{AG}(\mathtt{src} = A \wedge \mathtt{flow} = i \wedge \mathtt{loc} = S_1 \rightarrow$$
$$\mathtt{AG}(\mathtt{loc} = C \wedge \mathtt{flow} = i \rightarrow$$
$$\mathtt{A}((\mathtt{loc} \neq \mathsf{Drop} \wedge \mathtt{loc} \neq \mathsf{Exit})\mathtt{U}(\mathtt{loc} = S_1)))$$

Dynamic service chaining. For ease of presentation, we consider a sub-policy of the dynamic service chaining from Section 2: if a host is detected suspicious by Light IPS then all 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. Similar to the previous example, the top-level structure of the policy is $\mathtt{AG}(\gamma_1 \rightarrow \gamma_2)$, where $\gamma_1 = (\mathtt{src} = x \wedge susp[x] > 10)$ specifies that the *susp* count for a symbolic address x is larger than 10 and the current packet is from x, and γ_2 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. By quantifying the symbolic address x over all the addresses in the subnet Department *Dept*, we can specify this policy as follows.

$$\forall x \in Dept.\quad \mathtt{AG}(\mathtt{src} = x \wedge susp[x] > 10 \rightarrow$$
$$\mathtt{AG}(\mathtt{src} = x \rightarrow$$
$$\mathtt{A}((\mathtt{loc} \neq \mathsf{Drop} \wedge \mathtt{loc} \neq \mathsf{Exit})\mathtt{U}(\mathtt{loc} = H))))$$

6 ALGORITHMS OF NETSMC

We present key components of the algorithms behind NetSMC.

6.1 Symbolic Network States in EFO

The symbolic representation of network states is a formula summarizing constraints on relevant parts of the states; namely, state tables and the located packet. Consider the firewall example in Fig. 5, 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 there exists an entry in the table `Trust` whose value is 1. That is, $\exists x, y.Trust[x, y] = 1$. Furthermore, formulas representing network states that violate policies specified in our policy language are also existentially quantified over formulas about values of state table entries and the located packet's fields. Therefore, we propose to use the following fragment of existential first-order logic (EFO) as our symbolic state representation. As we will see later, this fragment is closed under operations used by our model checking algorithms.

$$\begin{array}{lllll}
\text{Atomic Predicates} & \alpha & ::= & x \in D \mid x \neq y \\
 & & \mid & loc = x \mid f = x \mid T[\overrightarrow{x_i}] = y \\
\text{Clauses} & \beta & ::= & \bigwedge_i \alpha_i \\
\text{State Formulas} & \phi & ::= & \bigvee_i \beta_i
\end{array}$$

The existential quantifications are only at the outermost level and we operate on the inner formulas without quantifiers, so we omit the closed top-level formula from the definitions. We call formulas *representing* a set of network states *state formulas*, written ϕ. We say that a network state (lp, Δ) is represented by ϕ, if there exists a substitution 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 represented by ϕ. State formulas ϕ are disjunctions of conjunctive clauses. The atomic predicates, α, in the conjunctive clauses include membership predicate, inequality check, and test for fields, location and state tables. For the firewall example above, the state formula is $\phi = (Trust[x, y] = 1)$.

6.2 Computing Pre-Image

Next, we describe how to compute the state formula of the pre-image of a symbolic state. Recall that given a set of network states S, the pre-image of S, denoted $Pre(S)$, is the set of network states that can transition to a state in S. That is, $Pre(S) = \{s | \exists s'.s \rightarrow s' \text{ and } s' \in S\}$. Given a state formula ϕ, we need to compute the state formula ϕ_{Pre}, such that $Sat(\phi_{Pre}) = Pre(Sat(\phi))$.

Standard algorithms would require to compute the projection of $\phi[\overrightarrow{T_i \mapsto T_i'}, \ldots] \wedge R(\vec{T_i} \ldots, \vec{T_i'} \ldots)$ to $(\vec{T_i}, \vec{f}, \texttt{loc})$. Here $\phi[\overrightarrow{T_i \mapsto T_i'}, \ldots]$ is the same as ϕ except that every occurrence of field/location/table names is replaced by a new name, and $R(\vec{T_i} \ldots, \vec{T_i'} \ldots)$ is the formula encoding possible network transitions. In short, the formula for pre-image represents states where the network transition is possible and the transition results in states that satisfy ϕ. While this works for symbolic approaches such as BDD-based model checking, it may be inefficient to compute the corresponding EFO state formula. Instead, we develop a custom algorithm that directly generates ϕ_{Pre} by transforming ϕ based on the transition rules.

Notation. Before explaining the algorithm, we define some auxiliary notation. Without loss of generality, we assume that for each clause β in ϕ, each field f appears at most once (any formula can be easily rewritten to this form). We write $var(f, \beta)$ to denote the variable 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 resulting formula from removing the clause α from β.

Top-level algorithm. Our pre-image computing algorithm is shown in Alg. 2 and 3. 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 makes use of the semantic rule Net-Trans-NF. Function computeLastPkt on line 6 computes the pre-image when ϕ represents the state where a new packet enters the network (based on the rule Net-Packet). Function computeLink on line 7 computes the pre-image when the transition is Net-Link. The algorithm returns the disjunction of all formulas for each possible transition. Next, we describe two key functions. We omit the third as it is very similar.

Packet transitions. Function computeLastPkt computes the pre-image under the transition rule Net-Packet. 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 rule Net-Packet, then the state before this

Algorithm 2 Computing the pre-image of a state formula.

```
1:  function computePreImage(N, φ)
2:      for all NF in the network do
3:          (L, T⃗, R) ← NF
4:          φ_r := computePreRule(r, φ)
5:          φ_NF := ⋁_{l∈L, r∈C} ⋁_{β∈φ_r} ((loc = l) ∧ β)
6:      φ_pkt := computeLastPkt(φ)
7:      φ_link := computeLink(φ)
8:      return ⋁_NF φ_NF ∨ φ_pkt ∨ φ_link
9:  function computePreRule(r, φ)
10:     r ← t ⇒ c
11:     φ_c := computePreCmd(c, φ)
12:     return ⋁_{β∈φ_c} (⋀_{at∈t} trans(at) ∧ β)
13: function computePreCmd(c, φ)
14:     match c with
15:         | a ⇒
16:             return computePreAction(a, φ)
17:         | u ⇒
18:             return computePreUpdate(u, φ)
19:         | c_1, c_2 ⇒
20:             φ_2 := computePreCmd(c_2, φ)
21:             return computePreCmd(c_1, φ_2)
22: function computeLastPkt(φ)
23:     φ′:=False
24:     for all β in φ do
25:         β′ := β\_{loc=var(loc,β)}
26:         for all f = x appearing in β do
27:             β′ := β′\_{f=x}
28:         for all ingress location l do
29:             β_1 := (var(loc, β) = l) ∧ β′ ∧ (loc = Drop)
30:             β_2 := (var(loc, β) = l) ∧ β′ ∧ (loc = Exit)
31:             φ′ := φ′ ∨ β_1 ∨ β_2
32:     return φ′
```

transition has the same state tables but a different packet. Therefore, all constraints on packet fields and locations are removed from β (line 25-27) 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 loc = Drop, loc = Exit are added, the same for $var(loc, \beta) = l$ for each ingress location l (line 29, 30).

NF transitions. The function computePreRule iteratively computes the pre-image ϕ_c under the actions and updates in r (line 11). The pre-image under rule r is obtained by adding constraints of the tests t in r

Algorithm 3 Sub-functions of computing the pre-image.

function computePreAction(a, ϕ)
  match a **with**
    | **fwd**(e) ⇒
      $(g_e, x_e) := F(e)$
      return $\bigvee_{\beta\in\phi} \beta \setminus_{\texttt{loc}} \wedge g_e \wedge (var(\texttt{loc}, \beta) = x_e)$
    | **drop** ⇒
      return $\bigvee_{\beta\in\phi} \beta \setminus_{\texttt{loc}} \wedge (var(\texttt{loc}, \beta) = \mathsf{Drop})$
    | **modify**(f, e) ⇒
      $(g_e, x_e) := F(e)$
      return $\bigvee_{\beta\in\phi} \beta \setminus_f \wedge g_e \wedge (var(f, \beta) = x_e)$
function computePreUpdate(u, ϕ)
  $\mathrm{T}[\overrightarrow{e_i}]{:=}e \leftarrow u$
  $(g_{e_i}, x_{e_i}) := F(e_i)$ for all e_i
  $(g_e, x_e) := F(e)$
  $g := \bigwedge_i g_{e_i} \wedge g_e$
  for all β_j in ϕ **do**
    ϕ_j := computeClause(u, β_i, $[(g_{e_i}, x_{e_i})]$, (g_e, x_e))
  return $\bigvee_i \bigvee_{\beta\in\phi_i} g \wedge \beta$
function computeClause(u, β, $[(g_{e_i}, x_{e_i})]$, (g_e, x_e))
  let *tList* be the list of state tests $T(\overrightarrow{x}) = y$ appearing in β
  match *tList* **with**
    | nil ⇒
      return β
    | h::hs ⇒
      ϕ_0 = computeClause(u, $\beta\setminus_h$)
      $\mathrm{T}[\overrightarrow{e_i}]{:=}e \leftarrow u$, $(T(\overrightarrow{x}) = y) \leftarrow h$
      $\beta_0 := (\overrightarrow{x} = \overrightarrow{x_{e_i}}) \wedge (y = x_e)$
      $\beta_j := h \wedge \bigwedge_{k=1..j-1}(x_k = x_{e_k}) \wedge (x_j \neq x_{e_j})$ for $j = 1, .., m$
      return $\bigvee_{\beta'\in\phi_0}((\beta_0 \wedge \beta') \vee \bigvee_j(\beta_j \wedge \beta'))$

using the helper function *trans* (not shown due to space) which translates each atomic test into a clause. The core of the function is to compute the pre-image under actions and updates in the rule (Algorithm 3).

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 $e = (T[\texttt{src}, \texttt{dst}])$, $g_e = (\texttt{src} = y_1 \wedge \texttt{dst} = y_2 \wedge T[y_1, y_2] = y_3)$, and $x_e = y_3$. Then the algorithm adds the constraint g_e and

$var(\texttt{loc}, \beta) = x_e$. 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.

To compute the pre-image under an update u, the function computePreUpdate computes the pre-image of each clause β using the sub-procedure computeClause, which then recursively enumerates all possible effects of u to β. 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. Thus the pre-image of β in the case is β_0 shown in line 27. In the second case, $x_i \neq x_{e_i}$ for at least one x_i. Therefore, we obtain the pre-image in this case as a disjunction of β_j shown in line 28.

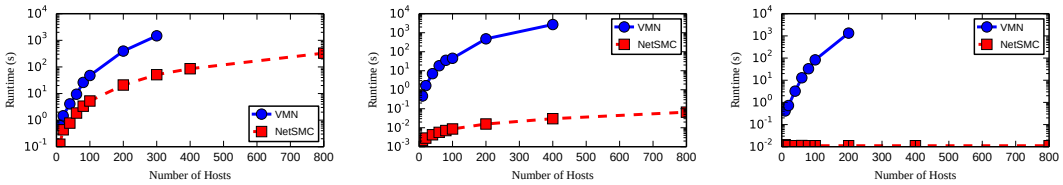
6.3 Containment of Network States

As we discussed in Sec. 3, we need an efficient approach to check the containment of two sets of network states (line 7 in Alg. 1). That is, 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 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 of relying on a general purpose solver, we observe that the state containment problem we face is a variant of the *query containment* problem well-studied in database theory [8]. 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 more clear, consider the state formula $\phi_1 = (\texttt{src} = x \wedge \texttt{dst} = y \wedge Trust[y, x] = 1)$. This formula can be viewed as a (conjunctive) query $q_1(x, y) :- src(x), dst(y), Trust(y, x)$ on a database with three tables: *src*, *dst* and *Trust*. 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 [8].

However, there are still a few challenges for applying the algorithm to our problem. First, as shown in [23], 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 is ranging over a continuous domain (e.g. rational numbers) [23], 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 represented by ϕ_1, but no homomorphism exists.



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

Fig. 8. Scalability with NF complexity

To address the first challenge, we break each clause in ϕ_1 into *atomic clauses*, which has been shown to handle inequalities [23]. 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. 4).

Algorithm 4 Checking containment

```
1: function checkCmt(φ1, φ2)
2:     for all β in φ1 do
3:         let [β0, .., βk] be the set of atomic clauses w.r.t. φ2 obtained from β
4:         for all i = 0 to k do
5:             if isEmpty(βi) then continue
6:             if there is no homomorphism from β' to βi for all β' ∈ φ2 then
7:                 return False
8:     return True
```

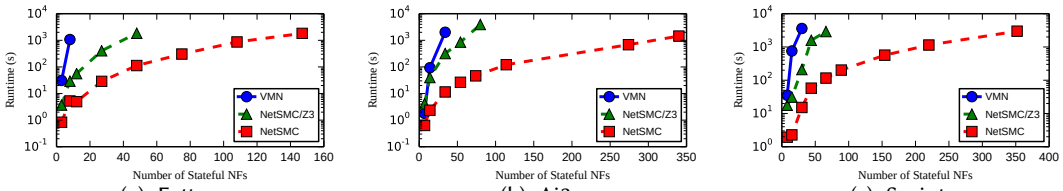
We prove the correctness of our algorithm w.r.t. the semantics: if NetSMC says policy verified, then all possible executions of the network satisfy the policy; and if NetSMC says policy violated, then there exists an execution that violates the policy. Formally:

Theorem 1 (Correctness). *Given a stateful network model N and a policy P, NetSMC returns True if and only if N satisfies the policy P.*

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 real-size networks and is orders of magnitude more efficient compared to alternative approaches based on general-purpose solvers;



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

Fig. 9. Scalability with network complexity.

- Our custom algorithm on containment checking in NetSMC is effective and is up to 42 times more efficient than naive approaches based on SMT solvers;
- NetSMC can verify a wide range of practical network policies in various network scenarios, which can not be supported in alternative tools.

7.1 Scalability

We evaluate the scalability of NetSMC on the complexity of NFs, topologies, and policies. For comparison, we consider the state-of-the-art stateful network verification tool VMN [34]. All experiments are conducted on a server with 20 cores (2.8GHz) and 128GB RAM.

NF complexity. Stateful NFs may implement complex functionalities by using multiple configuration rules. To evaluate the scalability of NetSMC w.r.t. the complexity of NFs, we consider three types of stateful NFs, namely, a stateful firewall, a load balancer, and 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. 8 shows the runtime of both tools on verifying reachability of a server to a host. We observe that 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 [1] topology as well as two real topologies, Ai3 and Sprint, from Topology Zoo [24]. 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 sizes. For each topology with n switches, we add additional $2n/3$ stateful NFs with each switch attached at most one stateful NF. We use each tool to verify the reachability policy of two hosts in each network. Given that VMN critically relies on the slicing technique of networks [34], we slice the flow-space of all tested networks before applying verification. In addition, to evaluate the benefit of our custom algorithms in model checking, we further consider an alternative approach by using Z3 to solve the containment problem of network states in NetSMC (shown as NetSMC/Z3).

Fig. 9 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×).

Second, we observe that our custom algorithm on containment checking significantly improves the scalability of our tool. In particular, when using Z3 to solve the containment checking problem, 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 various number of NFs on the chain. Since VMN does not support this type of policy, we consider the variant of NetSMC that uses Z3 for containment checking for comparison.

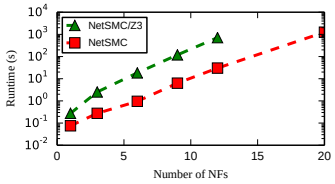


Fig. 10. Scalability with policy complexity.

Fig. 10 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 variant approach based on Z3.

Comparison with BDD-based model checking. To evaluate the benefit of our custom symbolic model checking algorithm, we further compare NetSMC with a classical BDD-based symbolic model checker NuSMV [9]. 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 a reachability policy, while NetSMC only uses 0.015 seconds with no such size constraint on the state tables. The result confirms our conjecture that BDD might not be the best encoding for stateful networks, and shows that the encoding and algorithms of NetSMC are efficiently.

7.2 Expressiveness

We highlight a wide range of policies that can be specified and checked using NetSMC, as summarized in Table 1. For each policy, we configure networks as described in the table. To simulate the case where a network violates a policy, we delete and modify some rules in the network configuration to introduce

misconfigurations. We use NetSMC to check each network scenario and report the time of verifying the policy or finding bugs in the network configurations.

We note that NetSMC significantly expands the scope of network policies which are hard to be checked in existing tools. In particular, today's stateless verification tools cannot model any network scenarios considered in the table, and the stateful network verification tool VMN can only handle the first three reachability policies. As comparison, NetSMC can successfully check all policies in all network scenarios with no false positives or false negatives. To the best of our knowledge, NetSMC is the only tool today that can efficiently check all these policies.

Policy	Network scenario	Time	
		Verification	Bug find
Flow reach. [34]	A stateful firewall with ACL rules.	0.06s	0.03s
Data reach. [34]	A content cache with a client and a server.	2.23s	0.0007s
Pipeline [34]	A stateful firewall with two hosts and servers.	0.001s	0.0006s
Flow affinity	As described in Fig. 1.	0.19s	0.04s
Dynamic service chaining	As described in Fig. 2.	0.1s	0.008s
Path pinning	As described in Fig. 3.	0.03s	0.0007s
Tag-based routing	Network as in Fig. 2. To check if a packet is labeled a tag, then it will not pass a specific MB.	0.04s	0.94s
Tag preservation	Network as in Fig. 2. To check if a packet is labeled a tag by a MB, this tag should be not be modified.	0.03s	0.98s
NAT consistency	If a NAT modifies a packet's port number then all future packets in the flow should have the same port number.	0.09s	0.078s

Table 1. Example policies supported by NetSMC.

8 RELATED WORK

Our stateful network model is motivated by existing work on network modeling and programming languages [3, 4, 15, 22, 30, 31, 42]. Our NF model shares key characteristics with the models in NetEgg [42] and SNAP [4] and is used by NetSMC and the correctness proof of our algorithms.

There is a rich body of work for testing and verifying forwarding behaviors in stateless networks [17, 19–21, 26, 27, 37, 39, 40, 43, 43, 44]. While those work can efficiently check a number of policies such as reachability and loop freedom, however, it is nontrivial to extend those work to support stateful data planes, which are the target of our work.

Our work is most closely related to recent efforts on stateful data plane testing and verification. Buzz [13] and SymNet [36] generate test cases for stateful networks based on symbolic execution. VMN [34] verifies reachability properties based on SMT encodings. Alpernas et al. presents a modular way of checking safety properties based on Datalog [2]. All of the above mentioned projects reason about stateful networks using

a general-purpose solver, and can only support a subset of our policies. Our approach is entirely different: we build a highly custom symbolic model checking tool for stateful networks to improve the efficiency and expressiveness.

In addition to data plane verification projects, there are several proposals on verifying network control planes, where the problem space is quite different from ours. Batfish [14], ERA [12], ARC [16] and Minesweeper [6] analyze routing control planes. NICE [7], VeriCon [5], SDNRacer [11], FlowLog [32] and Kuai [28] target SDN controllers.

There are other work on verifying firewalls [18, 33, 41, 45]. While they offer some capability to handle statefulness in firewalls, it is not clear how to generalize those work to handle more expressive network functions and policies.

9 CONCLUSION

This paper investigates symbolic model checking for stateful network verification. We define a restrictive, yet expressive model of stateful networks, identify a fragment of ACTL as policy specifications, and develop efficient custom symbolic model checking algorithms for them. We prove the correctness of the algorithms and implement a prototype tool NetSMC. We shows that NetSMC achieves orders of magnitude speedup compared to alternative approaches, while at the same time supports a wide range of policies.

REFERENCES

[1] 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, pages 63–74, New York, NY, USA, 2008. ACM.

[2] Kalev Alpernas, Roman Manevich, Aurojit Panda, Mooly Sagiv, Scott Shenker, Sharon Shoham, and Yaron Velner. Modular Safety Verification for Stateful Networks. *arXiv preprint arXiv:1708.05904*, 2017.

[3] 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*, pages 113–126. ACM, 2014.

[4] 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, pages 29–43, New York, NY, USA, 2016. ACM.

[5] 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.

[6] 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, pages 155–168, New York, NY, USA, 2017. ACM.

[7] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, Jennifer Rexford, et al. A nice way to test openflow applications. In *NSDI*, pages 127–140, 2012.

[8] 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*, pages 77–90. ACM, 1977.

[9] 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 *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

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

[11] 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.

[12] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 217–232, GA, 2016. USENIX Association.

[13] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. Buzz: Testing context-dependent policies in stateful networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 275–289, Santa Clara, CA, 2016. USENIX Association.

[14] 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 *NSDI*, pages 469–483, 2015.

[15] 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.

[16] 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*, SIGCOMM '16, pages 300–313, New York, NY, USA, 2016. ACM.

[17] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time Network Verification Using Atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 735–749, Boston, MA, 2017. USENIX Association.

[18] Alan Jeffrey and Taghrid Samak. Model checking firewall policy configurations. In *Policies for Distributed Systems and Networks, 2009. POLICY 2009. IEEE International Symposium on*, pages 60–67. IEEE, 2009.

[19] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111, Lombard, IL, 2013. USENIX.

[20] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, 2012. USENIX.

[21] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, 2013. USENIX.

[22] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 59–72, Oakland, CA, 2015. USENIX Association.

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

[24] 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.

[25] W Liu, H Li, O Huang, M Boucadair, N Leymann, Z Cao, and J Hu. Service Function Chaining (SFC) Use Cases. *https://tools.ietf.org/html/draft-liu-sfc-use-cases-01*, 2014.

[26] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 499–512, Oakland, CA, 2015. USENIX Association.

[27] 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, pages 290–301, New York, NY, USA, 2011. ACM.

[28] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for software-defined networks. In *Formal Methods in Computer-Aided Design (FMCAD), 2014*, pages 163–170. IEEE, 2014.

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

[30] 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.

[31] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks. In *NSDI*, pages 1–13, 2013.

[32] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. *NSDI, Apr*, 2014.

[33] Timothy Nelson, Christopher Barratt, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The Margrave Tool for Firewall Analysis. In *LISA*, pages 1–18, 2010.

[34] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying Reachability in Networks with Mutable Datapaths. 2016.

[35] 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, pages 13–24, New York, NY, USA, 2012. ACM.

[36] 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, pages 314–327, New York, NY, USA, 2016. ACM.

[37] 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 *IEEE SDN-NFV Conference*, 2016.

[38] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. Some complexity results for stateful network verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 811–830. Springer, 2016.

[39] 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 *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 2170–2183. IEEE, 2005.

[40] 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.

[41] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and Prasant Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[42] 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. ACM, 2015.

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

[44] 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 *NSDI*, volume 14, pages 87–99, 2014.

[45] Shuyuan Zhang, Abdulrahman Mahmoud, Sharad Malik, and Sanjai Narain. Verification and synthesis of firewalls using SAT and QBF. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–6. IEEE, 2012.