BUZZ: Testing Context-Dependent Policies in Stateful Networks

Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, Vyas Sekar
CMU

Abstract

Checking whether a network correctly implements intended policies is challenging even for basic reachability policies (Can X talk to Y?) in simple stateless networks with L2/L3 devices. In practice, operators implement more complex *context-dependent* policies by composing *stateful* network functions; e.g., if the IDS flags X for sending too many failed connections, then subsequent packets from X must be sent to a deep-packet inspection device. Unfortunately, existing approaches in network verification have fundamental expressiveness and scalability challenges in handling such scenarios. To bridge this gap, we present BUZZ, a practical model-based testing framework. BUZZ's design makes two key contributions: (1) Expressive and scalable models of the data plane, using a novel high-level traffic unit abstraction and by modeling complex network functions as an ensemble of finite-state machines; and (2) A scalable application of symbolic execution to tackle state-space explosion. We show that BUZZ generates test cases for a network with hundreds of network functions within two minutes (five orders of magnitude faster than alternative designs). We also show that BUZZ uncovers a range of both new and known policy violations in SDN/NFV systems.

1 Introduction

The security, performance, and availability of networks depend on the correct implementation of critical policy goals. Network operators realize these goals by configuring and composing network appliances, such as switches/routers, firewalls, and proxies.

Unfortunately, making sure that the network correctly implements a given policy is challenging, error-prone, and entails significant manual effort and operational costs [20,59]. As recent advances in network verification show, checking correctness is challenging even for simple *reachability* policies (Can X talk to Y?) in networks with *stateless* switches and routers [44, 52, 53, 57, 75].

In practice, operators' intended policies go well beyond reachability— operators implement a range of rich *context-dependent* policies using *stateful* network functions (*NFs*)[1] to ensure traffic goes through the intended sequence of *NFs*; e.g., if an intrusion detection system (IDS) flags host X for generating too many connections (i.e., if traffic context is "alarm"), then reroute subsequent flows to a deep packet inspection (DPI) filter [23]. Such rich policies and stateful data planes are quite common (e.g., the number of stateful *NFs* in a network may be comparable to the number of routers [70]). Looking forward, software-defined networking (SDN) [60] and network functions virtualization (NFV) [34] are poised to enable even richer in-network traffic processing services [22, 26, 29, 34, 42, 56].

What is critically lacking today is a principled way to check whether a stateful data plane correctly implements intended context-dependent policies. Existing approaches [44,52,53,57,75] face fundamental *expressiveness* and *scalability* challenges in this regard. First, current abstractions cannot capture stateful behaviors (e.g., how many connections host X has tried to establish) or express context-dependent policies (e.g., on-demand deep inspection). Second, trying to reason about stateful behaviors results in state-space explosion; e.g., a naive application of formal verification tools takes $>$ 20 hours even for a small network with 4-5 nodes (see §8).

We address these challenges and develop a principled testing framework called BUZZ. BUZZ takes in intended policies from the operator, and by exploring a model of the data plane, it finds abstract test traffic (i.e., an input that triggers policy-relevant states of a model of the data plane). It then translates the abstract test traffic into concrete test traffic and injects it into the actual data plane. Finally, it reports whether the observed behavior complies with the policies. As an active testing framework, BUZZ provides concrete assurances about the behavior "on-the-wire" and can help operators localize sources of violations [75] (§3).

In designing BUZZ, we make two key contributions:

- **Expressive-yet-scalable data plane models (§5):** We introduce a novel abstraction for network traffic called a BUZZ Data Unit (BDU). BDUs extend the notion of located packets from prior work [52] in three key ways: (1) it enables composition of diverse *NFs* spanning multiple protocol layers; (2) it simplifies models of *NFs* operating above L3 by aggregating a sequence of packets; and (3) it explicitly encodes traffic processing history to expose policy-relevant contexts. Second, we model individual *NFs* as FSMs that process BDUs and explicitly embed the relevant contexts into BDUs. A network then is simply a composition of individual *NF* models. To build tractable models, we

[1] An *NF* may be a switch/router or a middlebox (e.g., firewalls, load balancers, intrusion prevention systems, or proxies). It may be realized by a physical appliance or a virtual machine (VM).

decouple logically independent tasks (e.g., client-side vs. server-side connections) or units of traffic (e.g., distinct TCP connections) within each *NF* to create an ensemble of FSMs representation rather than a monolithic FSM.

- **Scalable test traffic generation (§6):** To generate abstract test traffic to explore the behaviors of the data plane model, we develop an optimized symbolic execution (SE)-based workflow. To combat the challenge of state space explosion [30,32], we engineer domain-specific optimizations (e.g., reducing the number and scope of symbolic variables). We also develop custom translation mechanisms to convert the output of this step into concrete test traffic.

We have implemented BUZZ as an application over `OpenDaylight` [14]. BUZZ provides both text-based and graphical interfaces for operators to input policies and receive test results through an automated workflow. We have written a library of models for several canonical *NFs* and implemented our SE optimizations using KLEE [31]. We have also developed simple monitoring and test resolution mechanisms (§7). BUZZ is open-source, and our code, models, and examples can be found at [1].

Our evaluation (§8) on a real testbed shows that BUZZ: (1) effectively helps detect both new and known policy violations within tens of seconds; (2) tests hundreds of policies in networks with hundreds of switches and stateful *NFs* within two minutes; (3) dramatically improves test scalability, providing nearly five orders of magnitude reduction in time for test traffic generation relative to strawman solutions (e.g., model checking).

2 Motivation

In this section, we use a few illustrative examples to discuss why it is challenging to check the correctness of context-dependent policies in stateful data planes.

Stateful firewalling: Today most firewalls capture TCP semantics. A common usage is reflexive ACLs [5] as shown in Figure 1, where incoming traffic is allowed depending on its *context*. In particular, the context-dependent policy here specifies that only traffic belonging to a TCP connection initiated by a host inside the department (i.e., if traffic context is "solicited") be allowed.

Prior work in network verification models each *NF* as a "transfer" function $T(hdr, port)$ whose input/output is a located packet (i.e., a *header, port* tuple) (e.g., [52, 53, 62]). Unfortunately, even the simple policy of Figure 1 cannot be captured by this stateless transfer function. In particular, it does not capture the policy-relevant *state* of the firewall (e.g., `SYN_SENT`) for a given connection.

Context-dependent traffic monitoring: In Figure 2, the operator uses a proxy to improve web performance.

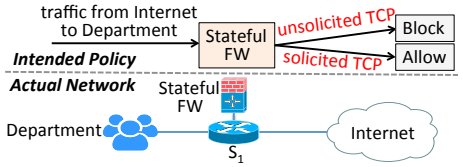


Figure 1: Is firewall allowing solicited and blocking unsolicited traffic?

She also wants to restrict web access; i.e., H_2 (a host in the department) cannot have access to `XYZ.com`. Here the context-dependent policy specifies that both cache hits/misses for H_2 should be monitored. As noted elsewhere [43], there could be subtle policy violations where cached responses evade the monitor because (1) the proxy hides traffic provenance (i.e., true origin), and (2) the proxy's response (i.e., hit vs. miss) depends on the hidden policy-relevant *state* (i.e., the current cache contents).

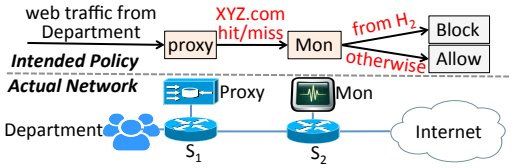


Figure 2: Are both cache hit/miss traffic monitored?

While there are mechanisms to fix this (e.g., [43]), operators need tools to check whether such mechanisms are implemented correctly. Again, a stateless transfer function [52, 53, 57] is insufficient, as it does not capture the state of the proxy.

Multi-stage triggers: Figure 3 uses a light-weight intrusion prevention system (L-IPS) for all traffic, and only subjects suspicious hosts (i.e., flagged by the L-IPS due to generating too many scans) to the expensive heavy-weight IPS (H-IPS) for payload signature matching. Such context-dependent multi-stage detection can minimize latency and reduce H-IPS load [42].

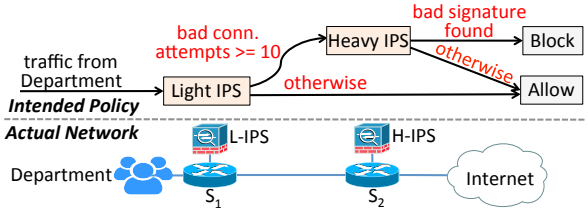


Figure 3: Is suspicious traffic sent to heavy IPS?

Again, we cannot check if such multi-stage policies are enforced correctly using existing mechanisms [44, 52, 53, 75] because they capture neither policy context (e.g., alarm/not alarm) nor data plane state (e.g., the count of bad connection attempts on L-IPS). This example also demonstrates that just capturing packet headers (e.g., [52,53,57]) is not sufficient, as the behavior of the H-IPS may depend on packet contents.

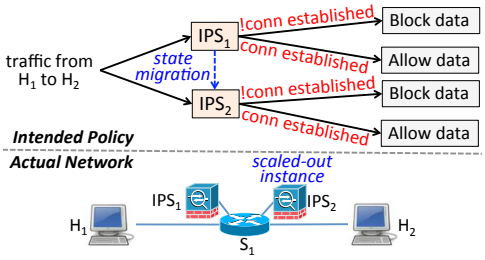


Figure 4: Does the scale-out mechanism honor the stateful semantics of migration?

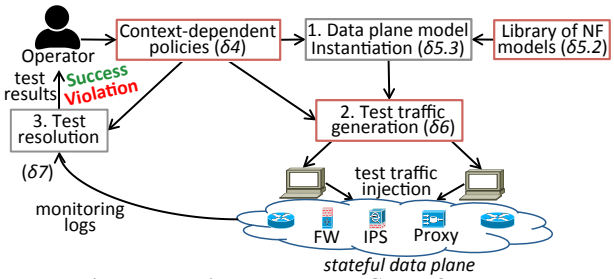


Figure 5: High-level workflow of BUZZ.

Dynamic NF deployments: NFV creates new opportunities for elastic scaling of *NFs* [34]. However, ensuring the correctness of policies in the presence of elastic scaling is not easy. For example, in Figure 4, suppose IPS_1 observes flow f_1 established between the two hosts; later f_1 is migrated to the newly launched IPS_2 for better load balancing [68]. Due to the stateful semantics of the IPS, IPS_2 needs to know that f_1 has already established a TCP connection; otherwise, IPS_2 may incorrectly block this flow. While recent efforts enable state migration [46,68], we need ways to check whether they do so correctly.

Similarly, in dynamic *NF* failure recovery [34], if the main *NF* fails, the backup *NF* needs to be activated with the correct state so that traffic is uninterrupted (e.g., see [69]). Again, we lack the ability to check whether such mechanisms work as intended.

3 Overview

Our goal is to enable network operators to check at human-interactive timescales whether their context-dependent policies are realized in stateful data planes. Next, we present a high-level view of BUZZ to meet this goal and summarize key challenges in realizing it.

To put our work in perspective, we note that there are two complementary approaches: *(1) Static verification* uses network configuration files to check whether the network behavior complies with the intended policies assuming the data plane behaves correctly (e.g., HSA [52], Veriflow [53], NOD [57], Batfish [44]); *(2) Active testing*, on the other hand, checks the behavior of the data plane by injecting test traffic into the network [75]. While both are useful, we adopt an active testing approach for two reasons. First, it provides practical assurances that things are actually working correctly "on-the-wire". Second, network behaviors in certain scenarios such as dynamic *NF* deployment (Figure 4) are hard to capture with a purely static approach.

Due to context-dependent policies and complex stateful behaviors, naive attempts to generate test traffic, either manually or via fuzzing [47,61], are ineffective. For example, in Figure 3, in order to trigger the policy context "L-IPS alarm" and check if traffic will actually go to H-IPS, we need to carefully craft a sequence of packets that drive the count of bad connections on L-IPS to ≥ 10; achieving this via randomly generated packets is unlikely. Our goal is to automate this process.

To bridge the gap between policies and the actual data plane, we adopt model-based testing (MBT) [72], which is useful when the blackbox behavior of a system needs to be actively tested. The high-level idea is to (1) use a *model* (or *specification*) of the system under test and a *search* mechanism to systematically find *test inputs* that trigger certain behaviors of the model, and then (2) compare the behavior of the system under test to the behavior of the model for each input [72].

Figure 5 shows the high-level workflow of BUZZ:

1. *Model Instantiation*: BUZZ instantiates a model of the data plane using the intended policies (the only input by the operator) and a library of *NF* models;
2. *Test Traffic Generation*: BUZZ generates abstract test traffic to trigger policy-relevant behaviors of the data plane model. BUZZ then translates it into concrete test traffic, which is then injected into the actual data plane;
3. *Test Resolution*: BUZZ monitors the actual data plane and compares the observed behavior to the intended policies. The result (i.e., success/violation) is reported to the operator.

There are two challenges in realizing this workflow:

- *Expressive-yet-scalable data plane models:* To see why this is challenging, let us consider some seemingly natural candidates. A natural starting point would be the transfer function abstraction [52, 62]; however, it is not expressive, as it offers no stateful semantics and no binding to the relevant context. On the other hand, using an *NF*'s implementation code as its model is not tractable (e.g., Squid [18] has $\geq$ 200K lines of code) and may suffer from other practical limitations (e.g., code may not be available, or implementation bugs may affect test traffic).
- *Scalable test traffic generation:* Exploring data plane's behaviors is challenging even for simple reachability policies in stateless data planes [75]. Our setting is worse, as reasoning about stateful behaviors requires addressing the challenge of state-space explosion. Off-the-shelf mechanisms (e.g., model checking) struggle beyond a few hundred lines of code (see §6 and §8).

Listing 1: An abstract stateful NF.

```
1  //Input: packet inPkt on port inPort
2  ⟨outPkt, state⟩ ← process(inPkt, state)
3  context ← stateToContextMap(state)
4  outPort ← applyPolicy(outPkt, context)
5  dispatch(outPkt, outPort)
```

We address these two challenges in §5 and §6, respectively. Before doing so, in the next section (§4), we first formalize our problem to shed light on the key requirements of modeling the data plane and generating test traffic.

4 Problem Formulation

In this section, we formalize our model-based testing framework to see what a data plane model should capture and what test traffic needs to do. These inform our approach to modeling (§5) and test traffic generation (§6).

4.1 Intuition behind model and test traffic

What should the data plane model capture: First, we give the intuition behind what an *NF* model needs to capture. As we saw in §2, data planes are stateful (e.g., the bad connection attempts count in Figure 3). However, being stateful is not sufficient for a data plane model to be expressive. Specifically, to test context-dependent policies, the model needs to explicitly map each state to a context. For example, if we want to trigger an alarm on L-IPS in Figure 3 (e.g., to check if the traffic will actually go to H-IPS), we need to capture the mapping from the bad connection attempts count (e.g., ≥ 10 or < 10) to the context (e.g., alarm or not alarm).

To understand what an *NF* model should capture, we consider the abstract *NF* shown in Listing 1 that shows the *NF* model as running three logical steps: (1) It processes an input packet and updates some relevant state (e.g., an IPS updating `bad_conn_attempts_count`) (Line 2); (2) It extracts the relevant *context* for the processed packet (e.g., alarm on an IPS based on `bad_conn_attempts_count`) (Line 3); (3) It applies the corresponding policy (e.g., drop, forward) via function *applyPolicy*(.) and then dispatches the packet to the policy-mandated port (Lines 4-5).

What should test traffic do? At a high level, test traffic for a given policy needs to drive the data plane to a state corresponding to the context. In Listing 1, this means we need to find a sequence of packets that drives the *NF* to a state (Line 2) that maps to the intended context (Line 3). If the *NF* is policy-compliant, the traffic at this point will be sent to a policy-mandated port (Lines 4-5). For example, to exercise the context of "L-IPS alarm" in Figure 3, test traffic needs to make `bad_conn_attempts_count` to exceed 10; then, we check whether traffic at this point actually goes to H-IPS.

4.2 Formal framework

Having seen the intuition behind state, context, and test traffic, we formalize these to inform our design.

Context-dependent policies: Let $context_{NF_i}^{pkt}$ denote the processing context corresponding to packet *pkt* at NF_i (Line 3 of Listing 1). Then, the *context sequence* of the packet is the sequence of contexts along the *NFs* it has traversed; i.e., if *pkt* has traversed NF_1, ..., NF_i, its context sequence is $ContextSeq^{pkt} = \langle context_{NF_1}^{pkt}, \ldots, context_{NF_i}^{pkt} \rangle$.

Context-dependent policies are expressed as a set of rules of the form:

$$Policy : TrafficSpec \times ContextSeq \mapsto PortSeq$$

Here, *TrafficSpec* is a predicate on the IP 5-tuple (e.g., source IP and transport protocol), *ContextSeq* is a context sequence, and *PortSeq* is a sequence of network ports *Ports* (interfaces).[2] For example, in Figure 3, the policy that mandates "if traffic triggers an alarm on L-IPS, it must be sent to H-IPS" is specified as:

$$\langle \texttt{srcIP=Dept} \rangle, \langle alarm_{L-IPS} \rangle \mapsto$$
$$\langle L{-}IPS \rightarrow S_1, S_1 \rightarrow S_2, S_2 \rightarrow H{-}IPS \rangle$$

(Policies for dynamic *NF* deployments, such as Figure 4, are defined slightly differently—see §6.4.)

Stateful data planes: Contexts are convenient "shorthands" to define policies. In reality, however, the data plane operates in terms of the related but (possibly) lower-level notion of state.

As we saw in Listing 1, a stateful *NF* takes an input packet on one of its ports, processes it, goes to a new state, and outputs a packet on one of its ports. A stateful *NF* can be naturally expressed as a finite-state machine (FSM) of the form $NF_i = (S_i, I_i, Ports_i, T_i)$, where S_i is the set of NF_i states, I_i is the initial state of NF_i, $Ports_i$ is the set of ports of NF_i (where $Ports_i \in Ports$), and $T_i : Pkts \times Ports_i \times S_i \mapsto Pkts \times Ports_i \times S_i$ is the stateful (as opposed to stateless, e.g., [52]) transfer function of NF_i. We model intended packet drops as sending packets to a virtual "drop port" on the *NF*. To model the entire data plane, the topology function $\tau : Ports \mapsto Ports$ captures the physical interconnection of *NFs*. Finally, we define the state of the data plane, S_{DP}, as the conjunction of the states of its individual *NFs*.

There are many levels of abstraction to write such an FSM on, from low-level code variables to high-level logical states (e.g., proxy cache state). Irrespective of this

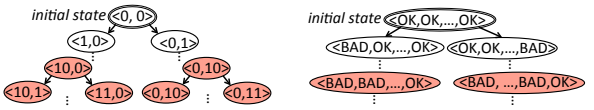
[2] Without loss of generality, we assume policies are in terms of physical *NF* instances as opposed to logical types of *NFs*. This is more precise because the semantics of stateful *NFs* (e.g., NATs) requires that both directions of a flow pass the same *NF* instance.

granularity, to be expressive for testing the model needs to provide a mapping from the states to the corresponding traffic specification and context:

$$stateToContextMap_i : 2^{S_i} \mapsto TrafficSpec \times C_i$$

where C_i denotes the set of all contexts of NF_i.

To illustrate this, let us revisit Figure 3. Figure 6 shows two possible ways of modeling L-IPS as an FSM. In both Figures 6a and 6b, each of the red states maps to $\langle$`srcIP=Dept`$\rangle, \langle alarm_{L-IPS}\rangle$—these mappings make the models expressive. (In §5, we will discuss other requirements of an FSM-based NF model in addition to expressiveness.)



(a) Each state is of the form $\langle badAttmpCnt_{H_1}, badAttmpCnt_{H_2}\rangle$ (b) Each state is of the form $\langle connStatus_{f_1}, \ldots, connStatus_{f_{20}}\rangle$

Figure 6: Two example FSM models of L-IPS of Figure 3 assuming a world with 2 hosts and 20 flows. The states corresponding to alarm (i.e., at least 10 bad connection attempts) are highlighted in red.

Test traffic: Test traffic needs to trigger the policy context by driving the data plane to a state that corresponds the context (e.g., a red state in Figure 6). Thus, $trace = \langle pkt_1, \ldots, pkt_m, \ldots, pkt_r\rangle$ is a test trace for $policy : trafficSpec \times contextSeq \mapsto portSeq$ iff:

1. Each packet $pkt \in trace$ satisfies $trafficSpec$, and
2. S_{DP} does not correspond to $contextSeq$ after injection of each of packets $\langle pkt_1, \ldots, pkt_{m-1}\rangle$, and
3. S_{DP} corresponds to $contextSeq$ after injection and processing each of packets $\langle pkt_m, \ldots, pkt_r\rangle$.

After $trace$ is injected into the actual data plane, test resolution involves checking whether packets $\langle pkt_m, \ldots, pkt_r\rangle$ actually traverse ports $portSeq$.

Takeaways: This framework suggests two key design implications: (1) While an FSM is a natural starting point to model a stateful NF, an expressive model should bridge the gap between its states and policy-mandated traffic specification and context (§5); and (2) Test traffic should satisfy the traffic specification and drive the data plane to a state that corresponds to the policy context (§6).

5 Data Plane Model Instantiation

In this section, we discuss how to instantiate a model of the data plane. Recall from §3 that this stage takes as input a library of NF models and the policy. The challenge in building such a library is to model each type of NF (e.g., stateful firewall, web proxy) such that these models are (1) *composable*, despite diverse types of NFs operating at different network layers; (2) *expressive*, despite stateful behaviors and hidden context; and (3) *scalable* to explore. After presenting our high-level approach (§5.1), we introduce a new abstract data unit for modeling input-output of NFs and describe how we create scalable NF models via an ensemble-of-FSMs representation (§5.2). Finally, we describe how we construct the network-wide model composing individual models of NFs (§5.3).

5.1 High-level idea

A natural starting point to model an NF that is composable is the *transfer function* from prior work [52, 62]. Each NF is modeled as: $lp \leftarrow T(lp)$. Here, the input/output is a located packet $lp = (pkt, port)$, an IP packet (header) along with its location in the network. However, as we saw in §2, this is not expressive on several fronts w.r.t. state and context. To see how we can make it expressive, let us revisit our abstract NF from Listing 1 and contrast it with the transfer function. This highlights two key missing elements: (1) there is no notion of state, and (2) the located packet has no binding to the relevant context.

Our formalism from §4 suggests two extensions: (1) Instead of the (stateless) transfer function, we need to move to an FSM-like abstraction that captures state and the state-to-context mappings; and (2) We need some way to logically bind a packet to its relevant context. To this end, we extend the located packet abstraction so that it carries the relevant context history as it traverses the data plane model. Then, we can consider an NF as an FSM that processes this extended located packet and explicitly includes the policy-relevant context in the outgoing packet. In a nutshell, this summarizes our basic insight to create an expressive model.

Next, we discuss how we translate this insight into a concrete realization. We also address the scalability requirement of NF models, as a naive FSM model will have too many states to explore.

5.2 Modeling individual NFs

The BUZZ Data Unit (BDU): We start by presenting our approach to modeling the extended located packet idea described above and explain how it enables composability, expressiveness, and scalability. Concretely, a BDU is a *struct* as shown in Listing 2 that extends a located packet [52, 62] in three key ways:

1. *Multi-layer abstraction with IP as the common denominator:* Unlike a located packet, a BDU can explicitly encode higher-layer semantics (e.g., HTTP GET or responses). The key to achieving model composability while enabling higher-layer semantics is simple. Borrowing from the design of IP, we pick the network layer as the narrow waist across diverse NFs.

Listing 2: BDU is the I/O unit of an *NF* model.

```
1  struct BDU{
2   // IP fields
3   int srcIP, dstIP, proto;
4   // transport
5   int srcPort,  dstPort;
6   // TCP specific
7   int tcpSYN, tcpACK, tcpFIN, tcpRST;
8   // HTTP specific
9   int  httpGetObj, httpRespObj;
10  // BUZZ-specific
11  int dropped, networkPort, BDUid;
12  // Each NF updates traffic context
13  int c-Tag[C_TAG_MAX]; //context tags
14  int p-Tag; //provenance tag
15  ...};
```

Each *NF* model processes only relevant fields of an input BDU (e.g., an L2 switch ignores HTTP fields).

2. *Tag fields for context and provenance:* First, to ensure a BDU carries its context as it goes through the network, we introduce the context tag, or `c-Tag`, field, which explicitly binds the BDU to its context (e.g., 1 bit for cache hit/miss, 1 bit for alarm/no-alarm). When the *NF* model receives an input BDU, it generates an output BDU with the updated `c-Tag` (e.g., a proxy may set the cache hit bit). Second, a BDU preserves its provenance via its `p-Tag` field. This field encodes the BDU's original 5-tuple indicating its *TrafficSpec*. This binding is needed because certain *NFs* (e.g., NATs, proxies) rewrite the original IP 5-tuple of a BDU. We ensure the provenance field `p-Tag` is left unchanged by *NF* models the BDU traverses.
3. *Aggregation for scalability:* Each BDU can represent a sequence of packets associated with higher-layer *NF* operations. This aggregation helps shrink the search space for finding test traffic (§6). For example, all packets of an HTTP reply are captured by a single BDU with the `httpRespObj` field indicating the retrieved object id; a proxy's state (e.g., cache contents) gets updated after receiving this BDU.

To design a BDU struct in practice, we need to identify the protocols that affect any context mentioned in the policies. The struct's fields are simply the union of the policy-related headers of these protocols. For example, if our policy involves a stateful firewall, then TCP `SYN` and `ACK` should be part of the fields, as these are the fields that denote connection establishment semantics. Since each *NF* model processes only relevant fields of an incoming BDU, our BDU abstraction is future-proof. For example, if we later need to add an ICMP field to the BDU of Listing 2, existing *NF* models will remain unchanged, as they simply ignore this new field.

Ensemble of FSMs representation: While there are many ways to expressively model a stateful *NF*, not all models may be scalable. To see why, consider modeling the state-space as the concatenation of state variables we have identified (e.g., in a proxy this concatenation may have three variables: per-host and per-server connection states and per-object cache state). Taking this approach means with *var* variables each with *val* possible values, such a monolithic FSM has val^{var} states (i.e., an exponential growth with the number of values). While it may be tempting to reduce the state space by moving to a layer-specific abstraction (e.g., a proxy model that ignores TCP and purely works at the HTTP layer), this is not viable, as the models of diverse *NFs* will not be composable.

To build a scalable FSM without compromising composability, we borrow insights from the design of actual *NFs*. *NF* programs in practice are not monolithic; rather, they independently track "active" connections, and different functional components of an *NF* are segmented; e.g., client- vs. server-side handling in a proxy are separate. This naturally suggests two opportunities:

1. *Decoupling independent traffic units:* Consider a stateful firewall. If modeled as a monolithic FSM, each state of the model involves states of individual connections. While this is expressive, it is not scalable as the number of connections grow. By decoupling per-connection states, we model the *NF* as an ensemble of FSMs. In general, this insight cuts the number of states from $|state|^{|conn|}$ to $|conn| \times |state|$, where $|conn|$ and $|state|$ denote the number of connections and states per connection, respectively.
2. *Decoupling independent tasks:* To illustrate this, consider a proxy. The code of a real proxy (e.g., `Squid` [18]) typically has three logical modules in charge of managing client-side and server-side connections and the cache. We decouple such logically independent tasks in the model so that instead of a monolithic FSM model with each state being of the "cross-product" form $\langle client\_TCP\_state, server\_TCP\_state, cache\_content \rangle$, we use an *ensemble* of three smaller FSMs, i.e., $\langle client\_TCP\_state \rangle$, $\langle server\_TCP\_state \rangle$, and $\langle cache\_content \rangle$. In general, if an *NF* has $|T|$ independent tasks with task i having S_i states, this idea cuts the number of states from $\prod_{i=1}^{|T|} |S_i|$ to $\sum_{i=1}^{|T|} |S_i|$.

Putting it together: Taken together, our BDU abstraction as the traffic I/O unit and FSM ensembles as *NF* models satisfy the three modeling requirements of composability, expressiveness, and scalability (§5.1). As an illustration, Listing 3 shows a code snippet of a proxy model focusing on the actions when a client requests a non-cached HTTP object and while the proxy has not established a TCP connection with the server. Each *NF* instance is identified by a unique `id` that allows us to index into the relevant variables. Since the traffic I/O of the model (Line 1) is a BDU, the model is com-

Listing 3: Proxy as an ensemble of FSMs.

```
1  BDU  Proxy(NFId id, BDU inBDU){
2      ...
3      if ((frmClnt(inBDU)) && (isHttpRq(inBDU))){
4         if (!cached(id, inBDU)){
5               if(srvConnEstablished(id, inBDU))
6                  outBDU=rqstFrmSrv(id, outBDU);
7               else
8                  outBDU=tcpSYNtoSrv(id, inBDU); }}
9      //set c-Tags based on context (e.g., hit/miss)
10     outBDU.c-Tags = ...
11     ...
12     return outBDU;}
```

posable with other *NF* models. Second, instead of a monolithic FSM, it is partitioned into these three dimensions (i.e., client-, server-side connections and cache) making the model scalable. The state variables of different proxy instances are naturally partitioned per *NF* instance (not shown) and help track the relevant *NF* states, and are updated by the *NF*-specific functions such as `srvConnEstablished`.[3] If the input `inBDU` is an HTTP request (Line 3) and the requested object is not cached (Line 4), the proxy checks the status of the server TCP connection. If it has already been established (Line 5), the output BDU is an HTTP request (Line 6). Otherwise, the proxy initiates a TCP connection with the server (Line 8). Finally, note that the proxy updates `c-Tags` of the output BDU before sending it out.

5.3 Composing the data plane model

Next we discuss how to instantiate a model of the data plane given the models of individual *NFs*. Listing 4 illustrates this for the network of Figure 2. BUZZ uses the policy to automatically concretize the relevant model parameters (e.g., lines 3–4 specify which content/host to watch). Lines 8–10 model the stateless switch, where we model a switch as a static data store lookup [52]. Note that a BDU captures its current location in the network via its `networkPort` field, which gets updated as it traverses the network. Function `lookUp()` takes an input BDU, looks up its forwarding table, and creates a new `outBDU` with its port value set based on the forwarding table.

Similar to prior work [52,75], our network model processes one-packet-per-*NF* at a time, without modeling (a) batching or queuing inside the network, (b) parallel processing inside *NFs*, or (c) simultaneous processing of different packets across *NFs*. As a result, the data plane model is a simple loop (Line 26); in each iteration, a BDU is processed (Line 27) in two steps: (1) it is forwarded to the other end of the current link (Line 28), (2) it is then passed as an argument to the *NF* connected at this end (e.g., a switch or firewall) (Line 29). The output BDU is then processed in the next iteration. The loop is executed until the BDU is "DONE"; i.e., it either reaches its destination or is dropped by an *NF*.[4] Based on the policy, wee identify the `Next_NF` in line 29. (As an optimization, our implementation prepopulates switches' `lookup()` and `Next_NF()` based on shortest-path routing between policy-relevant *NFs*.) The role of the `assert` statement will become clear in §6, where we discuss test traffic generation.

[3]The choice of passing *id*s and modeling state in per-*id* global variables is not fundamental but an artifact of using C/KLEE.

Listing 4: Data plane pseudocode for Figure 2.

```
1  // Symbolic BDUs to be instantiated (see §6).
2  BDU A[20];
3  int objToWatch = XYZ.com;
4  int hostToWatch = H2;
5  // Global state variables
6  bool Cache[2][100]; // 2 proxies,  100 objects
7  // Model of a switch
8  BDU  Switch(NFId id, BDU inBDU){
9      outBDU=lookUp(id, inBDU);
10     return outBDU;}
11 // Model of a monitoring NF
12 BDU  Mon(NFId id, BDU inBDU){
13      ...
14     outBDU = inBDU;
15     if (isHttp(id, inBDU)){
16       takeMonAction(id, inBDU);/* if inBDU
17       contains objToWatch destined to
18       hostToWatch, set outBDU.dropped to 1.*/}
19     ...
20     return outBDU;}
21 // Model of a proxy NF; See Listing 3
22 BDU Proxy(NFId id, BDU inBDU){...}
23 main(){
24   // Model of the data plane
25   initializeProvenanceTags(A[]);
26   for each injected A[i]
27    while (!DONE(A[i])){
28     Forward A[i] on current link;{
29     A[i] = Next_NF(A[i]);{
30     assert(
31     (!(A[i].p-Tag==hostId[H2]))
32     ||(!(A[i].c-Tags[cacheContext]==objToWatch));
33 }}}}
```

6 Test Traffic Generation

In this section, we discuss how to generate test traffic given the policies and the data plane model. First, we highlight why we choose symbolic execution (SE) as a starting mechanism to explore the data plane model (§6.1). Then we present our domain-specific optimizations to scale SE to generate abstract test traffic consisting of BDUs (§6.2). Then, we show how to convert this abstract test traffic into concrete test traffic (§6.3). Finally, we present an extension to test dynamic *NF* scenarios (§6.4).

6.1 Why symbolic execution (SE)?

For BUZZ to be usable by operators at human interactive timescales, it should generate test traffic within seconds to a few minutes even for large networks. This is challenging on two fronts:

[4]*NFs* may be time-triggered (e.g., TCP time-out), so we capture time using a BDU field. These "time BDUs" are injected by the network model periodically to update time-related states.

- **Traffic space explosion:** Unlike prior work where an IP packet header is an independent unit of test (hence mandating a search only over the header space [51, 53, 75, 76]), we need to search over a very large *traffic space* of all possible sequences of traffic units. While BDUs, as compared to IP packets, improve scalability via aggregation (§5.2), we still have to search over the space of possible BDU value assignments.
- **State space explosion:** Even though using the FSM ensembles abstraction significantly reduces the number of states (§5.2), it does not address *state space explosion* due to composition of NFs; e.g., if the models of NF_1 and NF_2 can reach K_1 and K_2 states, respectively, their composition will have $K_1 \times K_2$ states.

Unfortunately, several canonical search solutions (e.g., model checking [4, 36] and AI planning tools [7]) do not scale beyond 5-10 stateful NFs; e.g., model checking took 25 hours for policy involving only two contexts.

As the first measure to address the search scalability challenge, we choose symbolic execution (SE), which is a well-known approach to tackle state-space explosion [30]. At a high level, an SE engine explores possible behaviors of a program (in our case, the data plane model) by assigning different values to its *symbolic variables* [32]. In our implementation, we use KLEE [31], a popular SE engine.

6.2 Generating abstract test traffic

BUZZ employs SE as follows. For each $policy : trafficSpec \times contextSeq \mapsto portSeq$, we constrain the symbolic BDUs to satisfy the *TrafficSpec*. Then, to drive the SE engine to generate test traffic that satisfies $contextSeq = \langle context_{NF_1}, \ldots, context_{NF_N} \rangle$, we introduce the logical negation of *contextSeq* as an *assertion* in the network model code. In practice, if *contextSeq* involves contexts of N NFs $context_1, \ldots, context_N$, BUZZ instruments the network model with an assertion of the form $\neg(context_1 \wedge \cdots \wedge context_N)$, where each term is expressed in terms of BDUs' `c-Tag` sub-fields. The assertion guides the SE engine toward finding a "violation" of the assertion by assigning concrete values to symbolic BDUs.[5] In effect, SE generates *abstract test traffic* by concretizing a sequence of symbolic BDUs. The abstract test traffic will be then translated into concrete test traffic (§6.3), which in turn, will be injected into the actual data plane. The injected concrete test traffic must traverse the sequence of ports specified in *portSeq*; otherwise, the actual data plane violates *policy*.

To illustrate this, let us revisit Listing 4, where we want a test trace to check cached responses from the proxy to host H_2. Lines 30-32 show the assertion to get a sequence of i BDUs that change the state of the data plane such that the ith BDU in the abstract traffic trace: (1) is from host H_2 (Line 31), and (2) corresponds to a cached response (Line 32). For example, the SE engine may generate 6 BDUs: three BDUs between a host other than H_2 in the *Dept* and the proxy to establish a TCP connection (the 3-way handshake) where the third BDU has `httpGetObj = httpObjId` (this effectively makes the proxy cache the object), followed by another 3 BDUs, this time from H_2 with the field `httpGetObj` set to `httpObjId` to induce a cached response. Similarly, Listing 5 shows an assertion in Lines 7-8 to trigger alarms at both L-IPS and H-IPS of the example from Figure 3.

[5] Note that an assertion of the form $\neg(A_1 \wedge \cdots \wedge A_n)$, or equivalently $(\neg A_1 \vee \cdots \vee \neg A_n)$, is violated only if each term A_i is evaluated to `true`.

Listing 5: Assertion pseudocode for Figure 3 to trigger alarms at both IPSes.

```
1 // Global state variables
2 int L_IPS_Alarm[noOfHosts];//alarm per host
3 int H_IPS_Alarm[noOfHosts];//alarm per host
4 ...
5 //A[] is an array of symbolic BDUs
6 ...
7     assert((!(A[i].c-Tags[L_IPS_Alarm]==1)) ||
8            (!(A[i].c-Tags[H_IPS_Alarm]==1)));
```

While SE is significantly faster than other candidates, it is not sufficient for interactive use. Even after a broad sweep of configuration parameters to customize KLEE, it took several hours for a small network (§8.3). To scale to large topologies, we implement two optimizations:

- **Minimizing number of symbolic variables:** Making an entire BDU structure (Listing 2) symbolic forces KLEE to find values for every field. Instead, BUZZ identifies the policy-related subset of BDU fields and only makes these symbolic and concretizes the rest. For instance; when BUZZ is testing a data plane with a stateful firewall but no proxies, it makes the HTTP-relevant fields concrete (i.e., non-symbolic) by assigning a don't care value $*$ (represented by -1 in our implementation) to them.
- **Scoping values of symbolic variables:** The *trafficSpec* scopes the range of values a BDU may take. BUZZ further narrows this range using the policy and protocols semantics. For example, even though the `tcpSYN` field is an integer, BUZZ constrain it to be either 0 or 1.

Test coverage: Ideally, test traffic should cover the space of all possible traffic, including (1) packet traces of all possible lengths (in terms of number of packets in the trace), and (2) enumerating all possible values of the fields of each packet. However, this is impractical with respect to both test traffic generation and injection overheads. That is why even in case of simple reachability policies and stateless data planes in prior work [75], only one sample packet out of an equivalence class of packets (i.e., the set of all packets that experience the same forwarding behavior) is selected as the test packet. Sim-

ilarly, we define our test coverage goal as obtaining one test trace to exercise each policy. In §8, we will show that BUZZ (1) successfully satisfies this goal, and (2) can be used to satisfy alternative coverage goals.

6.3 Generating concrete test traffic

The output of the SE step is a sequence of BDUs $BDUSeq^{SE}$. Since BDUs are abstract, we cannot directly inject them into the actual data plane. Moreover, we cannot simply do a one-to-one translation between BDUs and raw packets and do a trace replay [3, 75] because we need to honor session semantics (e.g., for TCP or FTP) of the policies—several parameters of such sessions (e.g., TCP seq. numbers) are outside of our control and are chosen by the OS of the end hosts at run time.

To this end, we translate abstract test traffic into *test traffic injection scripts* that are run on end hosts to inject concrete test traffic. The translation algorithm uses a library of traffic injection commands that maps a known $BDUSeq_l$ into a script. For example, if a BDUSeq consists of 3 BDUs for TCP connection establishment and a web request, we map this into a `wget` with the required parameters (e.g., server IP and object URL). In the most basic case, the script will be an IP packet. Using our domain knowledge, we populated this library with commands (e.g., `getHTTP(.)`, `sendIPPacket(.)`) that support IP, TCP, UDP, FTP, and HTTP.

For completeness, its pseudocode is presented in Appendix A. Here we give the intuition behind our translation algorithm. We partition the $BDUSeq^{SE}$ based on srcIP-dstIP pairs (i.e., communication end-points) of BDUs; i.e., $BDUSeq^{SE} = \bigcup_l BDUSeq_l$. Then for each partition $BDUSeq_l$, we do a longest-specific match (i.e., match on a protocol at the highest possible layer of the network stack) in our test script library, retrieve the corresponding command for each subsequence, and then concatenate these commands to form a traffic injection script.

6.4 Testing dynamic NF deployments

Next we describe the extensions needed to handle dynamic *NF* deployment scenarios. Intuitively, the goal in these scenarios is to ensure the change is transparent with respect to stateful semantics of traffic. To be concrete, let $Policy_{before}$ and $Policy_{after}$ denote the policies that the operator intends to enforce before and after the "change" occurs, where the change is captured by *changeCond* (e.g., an *NF*'s scale-out, or failure). We define the correct enforcement of a dynamic *NF* deployment policy as follows: For each data plane state $s \in S_{DP}$, if *changeCond* is triggered while the data plane is in s, then $Policy_{after}$ is enforced correctly.

In Figure 4, $Policy_{before}$ is the top part of the policy graph (i.e., involving IPS_1), $Policy_{after}$ is the bottom part of the policy graph (i.e., involving IPS_2), and *changeCond* is IPS_1's scale-out. Irrespective of the state in which IPS_1 scales out, IPS_2 must start processing traffic with the same state at which IPS_1 has scaled out.

Abstract test traffic generation for dynamic *NF* deployment scenarios is slightly different from what we described in §6.1. At a high-level, for every data plane state $s \in S_{DP}$, BUZZ (1) generates test traffic to drive the data plane to s, (2) triggers *changeCond* (e.g., by scaling-out an *NF*), and (3) test if the data plane is compliant with $Policy_{after}$. For completeness, we describe the full procedure in Appendix B.

7 Implementation

BUZZ comprises $\approx$ 10,000 lines of code, including *NF* models, code for test traffic generation, test resolution, extensions to KLEE, and the operator interfaces. The entire workflow of BUZZ is implemented atop `OpenDayLight` [14]. The source code is available at [1].

Operator interface: Operators can enter policies using either a text-based or a graphical interface (example screenshots in Appendix C). BUZZ then performs a set of sanity checks on the policies and warns the operator of any mistakes (e.g., an overlap between *TrafficSpec* of two policies). This I/O is the only effort that BUZZ needs from the operator. Once policies are entered, the workflow of BUZZ (Figure 5) is entirely automated.

NF models: We have written `C` models for switches, ACL devices, stateful firewalls, NATs, L4 load balancers, HTTP and FTP proxies, passive monitoring, and simple intrusion prevention systems (e.g., counting failed connection attempts and matching payload signatures). Our models are between 10 (for a switch) to 100 lines (for a proxy cache) of C code. We reuse common templates across *NFs*; e.g., TCP connection sequence used in both the firewall and proxy models. Note that modeling *NFs* is a one-time offline task and can be augmented with community efforts [12]. We validated models by inspecting call graphs visualization [9, 21] on extensive manually generated input traffic to ensure the models are correct.

Test traffic generation and injection: We use KLEE with the optimizations discussed in §6.2 to generate BDU-level test traffic (i.e., abstract test traffic), and then translate it to test scripts that run at the injection points.

Test traffic monitoring and test resolution: We use offline monitoring via `tcpdump` (with suitable filters). BUZZ uses the monitoring logs to determine the test result. For completeness, we have provided the monitoring and test resolution pseudocode in Appendix D. Here we give the intuition behind this process. From the input policy, BUZZ inspects the monitoring logs to check

whether traffic has traversed the policy-mandated ports. If so, the test concludes with success. Otherwise, a policy violation along with the first violating port on which traffic appeared is declared.

8 Evaluation

In this section, we show that:

1. BUZZ can help detect a broad spectrum of both new and known policy violations (§8.1);
2. BUZZ works in close-to-interactive time scales (i.e., within two minutes) even for large topologies with 100s of switches and stateful *NFs* (§8.2); and
3. BUZZ's design is critical for its scalability (§8.3).

Testbed and topologies: We use a testbed of 13 server-grade machines (20-core 2.8GHz servers with 128GB RAM) connected via direct 1GbE links and a 10GbE Pica8 OpenFlow switch. On each server, with KVM installed, we run injectors and software *NFs* as separate VMs, connected via `Open vSwitch`. The specific stateful *NFs* are iptables [8] as a NAT and a stateful firewall, Squid [18] as a proxy, Snort [17] and Bro [65] as IPS/IDS, Balance [2], and PRADS [15].

In addition to the example scenarios from §2, we use 8 randomly selected recent topologies from the Internet Topology Zoo [19] with 6–196 nodes. We also use two larger topologies (400 and 600 nodes) by extending these topologies. These serve as switch-level topologies; we extend them with different *NFs* to enforce policies. For the scalability experiments, we augment each switch-level topology with stateful *NFs* (§8.2) by connecting each stateful *NF* to a randomly selected switch. As a concrete policy enforcement scheme, we used prior work to handle dynamic middleboxes [43]. (We reiterate designing this scheme is not the goal of BUZZ; we simply needed *some* concrete solution.)

8.1 BUZZ end-to-end use cases

First, we demonstrate the effectiveness of BUZZ in finding both new and known policy violations.

Finding new violations: Using BUZZ, we uncovered several policy violations in recent systems, a few of which we present here:

- **Violations due to reactive control in Kinetic [10]**: We set up a simple policy composed of an IDS followed by a Kinetic dynamic firewall. By generating malicious traffic, BUZZ found that the first few malicious packets are wrongly let through. The root cause of this violation is the delay between (1) the IDS's detection of malicious traffic and sending an "infected" event to the controller, and (2) the controller's reconfiguration of the data plane to block malicious traffic.
- **Incorrect state migration using OpenNF [46]**: We used the OpenNF-enhanced PRADS [15, 46] to enforce the following policy: if a host spawns more than *Thresh* TCP connections, its traffic should be sent to a rate limiter. BUZZ revealed a violation due to the incorrect state migration when we elastically scale a PRADS instance. Specifically, BUZZ made a host establish n_1 and n_2 sessions with a server before and after migration, respectively, such that: n_1, $n_2 < Thresh$, but $n_1+n_2 > Thresh$. BUZZ then found that traffic did not go to the rate limiter. This is because OpenNF does not migrate the session count (i.e., n_1) from $PRADS_1$ to $PRADS_2$.
- **Faulty policy composition using PGA [66]**: We used PGA[6] to compose two policies on traffic from H_1 to H_2: it should pass a load balancer and a stateful firewall ($policy_1$), and if it is found suspicious, it then should go to an IPS ($policy_2$). After enforcing the composition of the two policies, BUZZ found that the test traffic exercising $policy_1$ did not go through the firewall. This is because the SDN switch rules corresponding to $policy_1$ took precedence over the switch rules for $policy_2$, rendering $policy_2$ ineffective.
- **Incorrect tagging using FlowTags [43]:** BUZZ helped us identify a bug in our FlowTags implementation in OpenDaylight [14]. In the scenario of Figure 2, the controller code in charge of decoding tags (e.g., to distinguish hosts behind the proxy) would assign the same tag value to traffic from different hosts. Our test traffic showed that the proxy's cache hit replies bypass the monitoring device. BUZZ's traffic trace indicated that the tag values of cache miss/hit are identical; this gave us a hint as to focus on the controller code in charge of configuring the tagging behavior of the proxy.

Finding known violations: We used a "red team–blue team" exercise, to evaluate the utility of BUZZ in finding known policy violations. In each scenario, the red team (Student 1) secretly picks one of the policies (at random) from the set of policies that is known to both teams, and creates a failure that causes the network to violate this policy; e.g., misconfiguring L-IPS count threshold. The blue team (Student 2) uses BUZZ to identify a violation and localize the source of the policy violation.

Table 1 highlights the results for a subset of these scenarios and the specific traces that BUZZ generated. Three of the scenarios use the motivating examples from §2. In the Conn. limit. scenario, two hosts are connected to a server through an authentication server to prevent brute-force password guessing attacks. The authentication server is expected to halt a host's access after 3 consecutive failed log in attempts. Finally, in the asymmetric routing scenario, upstream and downstream traffic traverse different paths [55]. In all scenarios, the blue-team

[6] We used our implementation of PGA, as its code was unavailable.

"Red Team" scenario	BUZZ test trace	Violating NF
Cascaded NATs using Click IPRewriter [54] ; NAT_2 incorrectly rewrites srcIP triggering "assertion failure" on NAT_1 [38]	H_1 attempts to access to the server	NAT_2
Multi-stage triggers (Fig. 3); L-IPS miscounts by summing three hosts	H_1 makes 9 scan attempts followed by 9 scans by H_2	L-IPS
Conn. limit.; Login counter resets	H_1 makes 3 continuous log in attempts with a wrong password	Login counter
Conn. limit.; S_1 missing switch forwarding rules from Auth-Server to the protected server	H_2 makes a log in attempt with the correct password	S_1
Conflicting firewall rules: Rule 1, if internal connect to external IP, allow IP to access any internal port; Rule 2, block external access to internal port 443	A tcp connection from internal C_1 to external S_1 followed by an access from S_1 to C_1 : *port*443	Firewall
Asymmetric routing; Client-to-server TCP traffic goes through Bro, but the response bypasses Bro. Since Bro does not see the SYN_ACK packet, it (mistakenly) blocks the connection.	a tcp connection followed by tcp data packets	switch close to dst.

Table 1: Example red-blue team scenarios.

successfully localized the failure (i.e., which NF or link is the root cause) within 10 seconds.

It is useful at this time to reiterate that these types of violations could not be exposed by existing debugging tools such as ATPG [75], ping, or traceroute, as they do not capture violations w.r.t. stateful/context-dependent aspects. We also tried using fuzzing to generate test traffic, using both Scapy [16] and a custom fuzzer. Across all scenarios, fuzzing did not find any test trace within 48 hours. This is because we need targeted search to trigger specific data plane states, which fuzzing is not suited for.

8.2 Scalability

Recall that we envision operators using BUZZ in an interactive fashion; i.e., the time for test generation should be within 1-2 minutes even for large networks with hundreds of switches and stateful NFs.

We evaluate how BUZZ scales with topology size and policy complexity. We define policy complexity as the number of stateful NFs whose contexts appear in the policy. We consider a baseline policy that has 3 stateful NFs (a NAT, followed by a proxy, followed by a stateful firewall). The firewall is expected to block access from a fixed subset of origin hosts to certain web content. To create more complex policies, we linearly "chain" together repetitions of the baseline policy.

Figure 7 shows the average test traffic generation latency for various topology sizes and policy complexities. There are two takeaways. First, BUZZ generates test traffic in human-interactive time scales; even in the largest topology with 600 switches and the most complex policy it takes only 113 seconds. Second, BUZZ's test traffic generation latency only depends on the policy complexity: if we increase the topology size without increase the policy complexity, this will not add to the test traffic generation latency. This is expected, as test traffic generation involves a search over the data plane state space, which naturally is a function of stateful NFs.

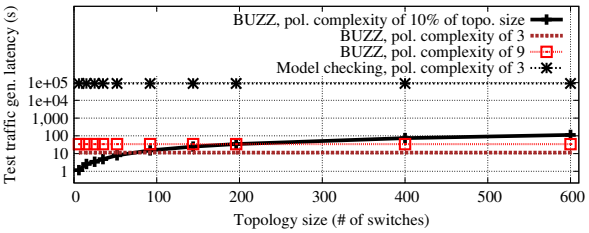


Figure 7: Test generation latency of BUZZ.

To put the traffic generation latency of BUZZ in perspective, Figure 7 also shows the traffic generation latency of a strawman solution of using the model checker CMBC [4]. Even on a small 9-node topology (6 switches and 3 stateful NFs), it took 25 hours; i.e., on a 90× larger topology, BUZZ is *at least* five orders of faster.

Test coverage: We have evaluated the test coverage of BUZZ, and here, we discuss the three takeaways. First, across all scenarios of §8.1 and §8.2, we explicitly enumerated all contexts, and observed that BUZZ provided full coverage with respect to the coverage goal of §6.1 (i.e., one test case to trigger each context). Second, we extended BUZZ to satisfy an alternative coverage goal of generating > 1 test trace per context. We enabled this through an iterative test generation process, where in each iteration, we obtain a new test case by using assertions such that a previously generated test case will not be generated again. Finally, while, in general, using multiple test cases per context may reveal new violations, in our experiments, we did not find new violations by doing so.

8.3 BUZZ design choices

Next, we do a component-wise analysis to demonstrate the effect of our key design choices and optimizations.

BDUs vs. packets: To see how aggregating a sequence of packets as a BDU helps with scalability, we use BUZZ to generate test traffic to test the proxy-monitor policy (Figure 2), first in terms of BDUs and then in terms of raw MTU-sized packets, on varying sizes of files to retrieve from the web. Figure 9 shows that on the topology with 600 switches and 300 stateful NFs, in case of packet-level test traffic generation, test traffic generation latency increases linearly with the file size. On the other hand, since the number of test packets is dominated by the number of object retrieval packets, aggregating all file retrieval packets as one BDU significantly cuts the latency. (The results, not shown, are consistent across topologies as well as using FTP instead of HTTP.)

Impact of SE optimizations: We examine the effect of the SE-specific optimizations (§6.2) in Figure 8. To put

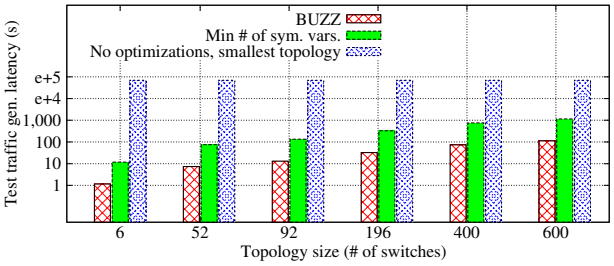


Figure 8: Improvements due to SE optimizations.

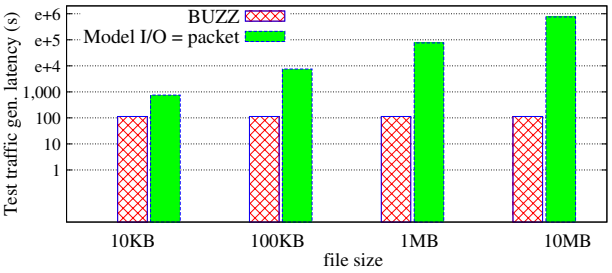


Figure 9: BDUs vs. packets for various request sizes.

these numbers in context, using KLEE without the optimizations on a network of six switches and a policy chain with three stateful *NFs* takes ≥ 19 hours. We see that (1) minimizing the number of symbolic variables cuts the test generation latency by three orders of magnitude, and (2) scoping the values yields a further $> 9\times$ reduction.

9 Related work

Network verification: There is a rich literature on checking reachability [40, 44, 51, 52, 57, 58, 73, 74]. The work closest to BUZZ is ATPG [75]. As discussed earlier, these do not capture the stateful behaviors and context-dependent policies.

Code verification: The work in [39] focuses on finding Click [54] code faults (e.g., crash) as opposed to verifying traffic processing policies (e.g., reachability). NICE combines model checking and SE to find bugs in control plane software [33]. BUZZ is complementary to these efforts.

Modeling stateful networks: Joseph and Stoica formalized middlebox forwarding behaviors but do not model stateful behaviors [50]. The only work that also models stateful behaviors are FlowTest [41], Symnet [71], and the work by Panda et al [64]. FlowTest's [41] high-level models are not composable and the AI planning approaches do not scale beyond 4-5 node networks. Symnet [71] uses models written in Haskell to capture NAT semantics similar to our example; based on published work we do not have details on their models, verification procedures, or scalability. The work by Panda et al. is different from BUZZ in terms of both goals (only reachability policies) and techniques (static checking) [64].

Policy enforcement: There are several frameworks to facilitate policy enforcement [10,43,46,63,66,67]. There are also efforts to generate correct-by-construction SDN programs [25, 27, 45]. Our work is complementary, as it checks whether the intended behavior manifests correctly in the actual data plane.

Simulation and shadow configurations: Simulation [13], emulation [6, 11], and shadow configurations [24] are common methods to model/test networks. BUZZ is orthogonal in that it focuses on generating test traffic. While our current focus is on active testing, BUZZ applies to these platforms as well. We also posit that our techniques can be used to validate these efforts.

10 Discussion

Model synthesis: BUZZ uses hand-generated models of *NFs*. A natural direction for future work is to use program analysis to automatically synthesize *NF* models from middlebox code (e.g., [35]) or logs (e.g., [28]).

Soundness vs. completeness: For "infinite-state" systems, it is not possible to simultaneously achieve both guarantees [49]. BUZZ's design favors soundness (i.e., if we report a violation, then the data plane actually has that behavior) over completeness (i.e., if we do not find a violation, then there are no bugs). In our setting, this is a worthwhile trade-off as we can repeat tests for greater coverage [49, 75] (e.g., see §8.2).

New use cases: Looking forward, we believe BUZZ can be extended to systematically check interoperability of new protocols with middleboxes [48]. As preliminary evidence, we were able to replicate a known problem with a middlebox-cooperative TCP extension called HICCUPS [37], where the protocol fails in the presence of middleboxes that modify certain headers or if there are multiple middleboxes on the path.

11 Conclusions

BUZZ tackles a key missing piece in network verification—checking *context-dependent policies* in *stateful data planes* introduces fundamental expressiveness and scalability challenges. We make two key contributions to address these challenges: *(1)* Developing expressive and scalable network models; and *(2)* An optimized application of symbolic execution to tackle state-space explosion. We demonstrate that BUZZ is scalable and it can help diagnose policy violations.

Acknowledgments

This work was supported in part by grant number N00014-13-1-0048 from the Office of Naval Research, NSF awards 1440056 and 1440065, and Intel Labs' University Research Office. Seyed K. Fayaz was supported by the VMware Graduate Fellowship and CMU Bertucci Fellowship. We thank the anonymous reviewers and our shepherd Kobus Van der Merwe for their suggestions.

References

[1] BUZZ. https://github.com/network-policy-tester/buzz.

[2] Balance. http://www.inlab.de/balance.html.

[3] Bit-Twist. http://bittwist.sourceforge.net.

[4] CBMC. http://www.cprover.org/cbmc/.

[5] Cisco's Reflexive Access Lists. http://bit.ly/1O8N5p2.

[6] Emulab. http://www.emulab.net/.

[7] Graphplan. http://www.cs.cmu.edu/~avrim/graphplan.html.

[8] iptables. http://www.netfilter.org/projects/iptables/.

[9] KCachegrind. http://kcachegrind.sourceforge.net/html/Home.html.

[10] Kinetic. http://resonance.noise.gatech.edu/.

[11] Mininet. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

[12] Network Function Virtualization Research Group (NFVRG).

[13] ns-3. http://www.nsnam.org/.

[14] OpenDaylight project. http://www.opendaylight.org/.

[15] PRADS. http://gamelinux.github.io/prads/.

[16] Scapy. http://bit.ly/1FiqZyK.

[17] Snort. http://www.snort.org/.

[18] Squid. http://www.squid-cache.org/.

[19] The Internet Topology Zoo. http://www.topology-zoo.org/index.html.

[20] Troubleshooting the network survey. http://eastzone.github.io/atpg/docs/NetDebugSurvey.pdf.

[21] Valgrind. http://www.valgrind.org/.

[22] High Performance Service Chaining for Advanced Software-Defined Networking (SDN) . http://intel.ly/1ilX5PG, 2014.

[23] Tackling the Dynamic Service Chaining Challenge of NFV/SDN Networks with Wind River and Intel. http://intel.ly/1EFmEVQ, 2014.

[24] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. In *Proc. SIGCOMM*, 2008.

[25] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.

[26] B. Anwer, T. Benson, N. Feamster, D. Levin, and J. Rexford. A slick control plane for network middleboxes. In *Proc. HotSDN*, 2013.

[27] T. Ball, N. Bjorner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarskyi. VeriCon: Towards verifying controller programs in software-defined networks. In *Proc. PLDI*, 2014.

[28] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proc. ESEC/FSE*, 2011.

[29] M. Boucadair, C. Jacquenet, R. Parker, D. Lopez, J. Guichard, and C. Pignataro. Differentiated Service Function Chaining Framework. https://tools.ietf.org/html/draft-boucadair-service-chaining-framework-00, 2013.

[30] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^20 States and Beyond. *Inf. Comput.*, 98(2), 1992.

[31] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, 2008.

[32] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[33] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test openflow applications. In *Proc. NSDI*, 2012.

[34] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Deng, J. Benitez, U. Michel, H. Damker, K. Ogaki, T. Matsuzaki, M. Fukui, K. Shimano, , D. Delisle, Q. Loudier, C. Kolias, I. Guardini, E. Demaria, R. Minerva, A. Manzalini, D. Lpez, F. Javier, R. alguero, F. Ruhl, and P. Sen. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. http://portal.etsi.org/nfv/nfv_white_paper.pdf, 2012.

[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855. 2000.

[36] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[37] R. Craven, R. Beverly, and M. Allman. A middlebox-cooperative tcp for a non end-to-end internet. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 151–162. ACM, 2014.

[38] M. Dobrescu, K. Argyarki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *Proc. NSDI*, 2012.

[39] M. Dobrescu and K. Argyraki. Software dataplane verification. In *Proc. NSDI*, 2014.

[40] D. J. Dougherty, T. Nelson, C. Barratt, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *Proc. LISA*, 2010.

[41] S. K. Fayaz and V. Sekar. Testing stateful and dynamic data planes with FlowTest. In *Proc. HotSDN*, 2014.

[42] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic DDoS defense. In *Proc. USENIX Security Symposium*, 2015.

[43] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *Proc. NSDI*, 2014.

[44] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Proc. NSDI*, 2015.

[45] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. *SIGPLAN Not.*, 46(9), Sept. 2011.

[46] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *Proc. SIGCOMM*, 2014.

[47] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *ACM Queue*, 2012.

[48] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? In *Proc. IMC*, 2011.

[49] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 2009.

[50] D. Joseph and I. Stoica. Modeling middleboxes. *Netwrk. Mag. of Global Internetwkg.*, 22(5), 2008.

[51] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. NSDI*, 2013.

[52] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proc. NSDI*, 2012.

[53] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.

[54] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 2000.

[55] F. Le, E. Nahum, V. Pappas, M. Touma, and D. Verma. Experiences deploying a transparent split tcp middlebox and the implications for nfv. In *Proc. HotMiddlebox*, 2015.

[56] W. Liu, H. Li, O. Huang, M. Boucadair, N. Leymann, Z. Cao, and J. Hu. Service Function Chaining (SFC) Use Cases. `http://bit.ly/1JTVneh`, 2014.

[57] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *Proc. NSDI*, 2015.

[58] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proc. SIGCOMM*, 2011.

[59] N. McKeown. Mind the Gap: SIGCOMM'12 Keynote. `http://bit.ly/1izyVld`.

[60] N. McKeown et al. OpenFlow: enabling innovation in campus networks. *CCR*, March 2008.

[61] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 1990.

[62] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proc. NSDI*, 2013.

[63] S. Palkar, C. Lan, S. Han, K. J. amd Aurojit Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A framework for NFV applications. In *Proc. SOSP*, 2015.

[64] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying Isolation Properties in the Presence of Middleboxes. `arXiv:submit/1075591`.

[65] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1999.

[66] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. PGA: Using graphs to express and automatically reconcile network policies. In *Proc. SIGCOMM*, 2015.

[67] Z. Qazi, C. Tu, L. Chiang, R. Miao, and M. Yu. SIMPLE-fying middlebox policy enforcement using sdn. In *Proc. SIGCOMM*, 2013.

[68] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Proc. NSDI*, 2013.

[69] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback recovery for middleboxes. In *Proc. SIGCOMM*, 2015.

[70] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. SIGCOMM*, SIGCOMM, 2012.

[71] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Static checking for stateful networks. In *Proc. HotMiddlebox*, 2013.

[72] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5), 2012.

[73] G. Xie, J. Zhan, D. Maltx, H. Z. G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In *Proc. INFOCOM*, 2005.

[74] L. Yuan and H. Chen. FIREMAN: a toolkit for FIREwall Modeling and ANalysis. In *Proc. IEEE Symposium on Security and Privacy*, 2006.

[75] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT*, 2012.

[76] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proc. NSDI*, 2014.

A Translating abstract test traffic into test traffic injection scripts

Figure 10 shows the pseudocode for the translation mechanism (§6.3).

```
▷ Inputs:
#1: a sequence of BDUs from Symbolic Execution
BDUseq^SE = ⟨BDU_n : n = 1,2,...,N⟩,
each BDU_n has an abstract pred_d
#2: a cmd-BDUs library cmdlib = {⟨cmd_1, Seq^cmd_1⟩,
⟨cmd_2, Seq^cmd_2⟩,...,⟨cmd_M, Seq^cmd_M⟩}
#3: a set of end-hosts H = {H_k : k = 1,2,...,K} to execute cmds
▷ Outputs:
#1: a number of scripts S = {script^H1 ··· script^HK} to
be executed on end-hosts {H_k : k = 1,2,...,K},
where script^Hk is a sequence of ⟨··· cmd_i^Hk ···⟩, such that
⟨··· Seq^cmd_i^H1 ···⟩ is equivalent to BDUseq^SE
▷ Sort cmd-BDUs library from cmds with most BDUs to
least BDUs
cmdlib = Sort(cmdlib)
▷ Decompose BDUseq^SE sequence into subsequences
BDUsubseq^SE of BDUs with same predicate pred
{BDUsubseq^SE_pred_d : d = 1,2,...,D} = Decompose(BDUseq^SE)
for each BDUsubseq^SE_pred_d in {BDUsubseq^SE_pred_d : d = 1,2,...,D}
    ▷ Instantiate a script_pred_d to store cmd for BDUsubseq^SE_pred_d
    script_pred_d ← empty
    ▷ Match the BDUs in BDUsubseq^SE_pred_d with cmds in cmdlib
    for each cmd_m in cmdlib
        for BDU_n in BDUsubseq^SE_pred_d
            ▷ if Seq^cmd_m equals to a BDU substring of BDUsubseq^SE_pred_d
            started at BDU_n
            if Substring(BDUsubseq^SE_pred_d, BDU_n, len(Seq^cmd_m)) == Seq^cmd_m
                ▷ add the matched cmd_m and the first matched BDU's index n
                to script_pred_d
                script_pred_d.add(cmd_m^n)
                ▷ mark all BDUs in Substring(BDUsubseq^SE_pred_d, BDU_n,
                len(Seq^cmd_m))
                Mark(Substring(BDUsubseq^SE_pred_d, BDU_n, len(Seq^cmd_m)))
        ▷ remove all marked BDUs from BDUsubseq^SE_pred_d
        RemoveMarked(BDUsubseq^SE_pred_d)
        if all BDUs in BDUsubseq^SE_pred_d are marked, then break
    ▷ Sort every cmd_m^n in script_pred_d by its first matched BDU's index n
    script_pred_d = Sort(script_pred_d)
    ▷ Map abstract pred_d to real test host H_pred_d and assign script to host
    script_H_pred_d = script_pred_d
```

Figure 10: Pseudocode for translating abstract test traffic into test traffic injection scripts.

B Abstract test traffic generation for change management policies

Figure 11 shows the abstract test traffic generation pseudocode for change management policies (§6.4).

C Operator interface of BUZZ

Figures 12 and 13 show operator's interface (§7).

D Test resolution

Figure 14 shows the pseudocode for test resolution (§7).

```
▷ Inputs:
#1: Policy_1:pred_1(5−tuple)×C_1↦Ports_1 before migrate/rollback
#2: Policy_2:pred_2(5−tuple)×C_2↦Ports_2 after migrate/rollback
▷ Outputs:
#1: a sequence of BDUseq^SE = ⟨BDU_n : n = 1,2,...,N⟩ with two substrings,
BDUseq^SE_before and BDUseq^SE_after, which should satisfy:
BDUseq^SE_before exploits all possible context context in C_1 before
migration/rollback happens.
BDUseq^SE_after test all possible context in C_2 after the migration/rollback.
▷ Init BDU sequence
BDUseq^SE = ⟨BDU_n : n = 1,2,...,N⟩
▷ note the values in BDU_n for calculation by Symbolic Execution
makesymbolic(BDU_n)
▷ exploits all possible context in C_1
▷ BDUs processed sequentially by Policy_1
for each BDU_i in BDUseq^SE
    if BDU_i is in BDUseq^SE_before
        ▷ process BDU_i by Policy_1 and update C_1
        C_1 = Policy_1(pred_1(BDU_i), C_1, Ports_1)
▷ do migrate/rollback and change service chain from Policy_1 to Policy_2
▷ map ports
Ports_2 = g(Ports_1)
▷ migrate/rollback context
C_2 = C_1
▷ test all possible context in C_2
▷ BDUs processed sequentially by Policy_2
for each BDU_j in BDUseq^SE
    if BDU_j is in BDUseq^SE_after
        ▷ process BDU_i by Policy_2 and update C_2
        C_2 = Policy_2(pred_2(BDU_i), C_2, Ports_2)
▷ generate BDU sequence with values assigned by Symbolic Execution
symbolicoutput = ⟨BDU_n : n = 1,2,...,N⟩
```

Figure 11: Pseudocode for abstract test traffic generation for change management policies.

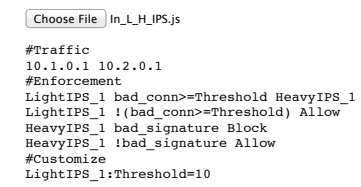


Figure 12: Text-based interface to input policies (e.g., multistage-triggers policy in Figure 3).

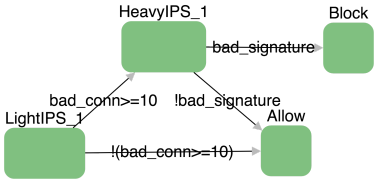


Figure 13: Graphical interface to input policies (e.g., multistage-triggers policy in Figure 3).

```
▷ Inputs:
#1: packet traces pkttrace_port_i dumped at each port_i in Ports
#2: policy Policy:pred(5−tuple)×C↦Ports, where C includes all possible
contexts
▷ Outputs:
#1: The resolution result of each context context_i in C in terms of pass/fail
#2: The port of the NF that causes the failure
▷ perform resolution scheme for each context context_i in C
for each context_i in C
    ▷ Trace = ⟨pkt_m, ..., pkt_r⟩ is the test packets for this context
    for testpkt in ⟨pkt_m, ..., pkt_r⟩
        ▷ calculate the logically correct ports testpkt should reach
        Ports^logical_testpkt = Policy(pred(testpkt)), context_i)
        ▷ find the real ports testpkt has reached
        Ports^reality_testpkt = search testpkt in each pkttrace_port_i
        if Ports^reality_testpkt == Ports^logical_testpkt
            context_i test pass
        else
            context_i test fail
            ▷ Compare the port of Ports^reality_testpkt and Ports^logical_testpkt and find the first
            different port, which is the NF that causes the failure.
            FailedNFPort = FirstDiffPort(Ports^reality_testpkt, Ports^logical_testpkt)
        mark context_i as tested
```

Figure 14: Pseudocode for BUZZ test resolution.