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 merely 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. 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 recent 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 [22, 60]. As recent advances in network verification show, checking correctness is challenging even for simple reachability policies (Can X talk to Y) in networks with simple and stateless switches and routers [46, 54, 55, 58, 77].

In practice, operators’ policy intents 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 [25]. Such rich policies and stateful data planes are already quite common; e.g., surveys show that most networks have as many stateful NFs as routers [70]. Looking forward, networking paradigms such as software-defined networking (SDN) [61] and network functions virtualization (NFV) [36] are poised to enable even richer inter-network traffic processing services [24, 28, 31, 36, 44, 74].

What is critically lacking today is a principled way to check whether a stateful data plane correctly implements a given set of context-dependent policies. Existing approaches [46, 54, 55, 58, 77] 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 is a model-based tester: (1) first, 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), (2) it translates the abstract test traffic into concrete test traffic and injects it into the actual data plane, and (3) finally, it reports whether the observed behavior complies with the policy. As an active testing framework, BUZZ provides concrete assurances about the behavior “on-the-wire” and helps operators localize sources of violations [77] (§3).

In designing BUZZ, we make two key contributions:

- **Expressive-yet-scalable data plane models (§5):** First, we introduce a novel abstraction of the network’s traffic unit called a BUZZ Data Unit (BDU). A BDU extends the notion of a located packet from prior work [54] in three key ways: (1) it enables composition of models 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 to the model NFs. Second, we expressively model individual NFs as FSMs that process BDUs and explicitly embed the relevant contexts into BDUs—a network is simply a composition of individual NF models. To build

\(^1\)An NF may be a switch/router or a middlebox (e.g., firewalls, load balancers, intrusion prevention systems, proxies). It may be realized by a physical appliance or virtual machine (VM).
scalable (i.e., tractable to explore) models, we 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 a more tractable ensemble of FSMs representation rather than a monolithic FSM.

- **Scalable test traffic generation (§6):** To generate abstract test traffic by exploring the state space of the data plane, we develop an optimized symbolic execution (SE)-based workflow. To combat the challenge of state space explosion [32, 34], 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 GUI 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 domain-specific SE optimizations on top of KLEE [33]. We have also engineered monitoring and test resolution mechanisms (§7). We have open-sourced our code, models, and examples [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

To see why checking the correctness of context-dependent policies in stateful data planes is challenging, we use a few illustrative examples that highlight the limitations of prior work.

**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 mandates that only traffic belonging to a TCP connection initiated by a host inside the department (i.e., if traffic context is “solicited”) is allowed.

![Figure 1: Is firewall allowing solicited and blocking unsolicited traffic?](image1)

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

Prior work on network verification models each NF as a “transfer” function $T(hdr, port)$ whose input/output is a located packet (a header, port tuple) (e.g., [54, 55, 63]). 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 enhance web experience. 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 (i.e., context) for $H_2$ should be monitored. As noted elsewhere [45], if implemented naively, there could be subtle policy violations where cached responses evade the monitor. There are two root causes for this: (1) the proxy hides traffic provenance (i.e., true origin), and (2) the proxy’s responses (i.e., hit vs. miss) depends on its hidden policy-relevant state (i.e., cache contents).

While there are mechanisms to fix this (e.g., via extended SDN APIs [45]), operators need tools to check whether such policy enforcement mechanisms work correctly. Again, a stateless transfer function [54, 55, 58] is insufficient, as it does not capture the state of the proxy.

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

**Multi-stage triggers:** In Figure 3, we intend to (1) use a light-weight intrusion prevention system (L-IPS) for all traffic, and (2) only subject suspicious hosts flagged by the L-IPS (e.g., 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 [44].

Again, we cannot check if such multi-stage policies are enforced correctly using existing reachability mechanisms [46,54,55,77] 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., [54,55,58]) is not sufficient, as the behavior of the H-IPS may depend on the actual content.

![Figure 3: Is suspicious traffic sent to heavy IPS?](image2)
Dynamic NF deployments: NFV creates new opportunities for elastic scaling of NFs [36]. However, ensuring the correctness of policies in the presence of elastic scaling is not easy. For example, in Figure 4, suppose IPS₁ observes flow f₁; later f₁ is migrated to the newly launched IPS₂ (e.g., to achieve better load balancing [69]). Due to the stateful semantics of the IPS, IPS₂ needs to know that f₁ has already established a TCP connection; otherwise, IPS₂ may incorrectly block this flow. While recent efforts enable state migration [48,69], we need ways to check whether they do so correctly.

Similarly, in dynamic NF failure recovery [36], the goal is to ensure that if the main NF fails, the backup NF kicks in with the correct state so that traffic will be processed uninterrupted (e.g., see [71]). Again, today we lack the ability to check whether such mechanisms actually 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. Here 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 to network verification: (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 [54], Veriflow [55], NOD [58], Batfish [46]); (2) Active testing, on the other hand, checks the behavior of the data plane by injecting test traffic into the network [77]. While both are useful, we take the 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 [49,62], 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

\[ \geq 10; \] 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) [73]. MBT is a broad class of approaches that is used 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 [73].

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 concludes the test by comparing 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 for modeling an NF would be the transfer function abstraction; 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 lacks scalability (e.g., Squid [19] has \[ \geq 200K \] lines of code) and may suffer from other practical limitations (e.g., code may not be available, or implementation bugs may creep into test traffic generated based on this model).

- Scalable test traffic generation: Exploring data plane’s behaviors is challenging even for simple reachability policies in stateless data planes [77]. Our setting is even 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).

Roadmap: We discuss how we address these two challenges in §§5 and 6, respectively. Before that, we first formalize our problem to shed light on the key requirements of the model and test generation steps §4.

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 alone is not sufficient for a data plane model to be expressive. Specifically, in order 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 see if the traffic will actually go to H-IPS), we need to capture the mapping from the bad connection attempts count (e.g., ≥ 10) to context (e.g., alarm or not alarm).

To understand what an NF model should capture, let us consider the abstract NF shown in Listing 1. It takes in the intended policies (via applyPolicy()) and maintains some internal state. We can view the NF as running two logical steps: (1) It processes an input packet and updates some relevant state (e.g., an IPS updating bad_conn_attempts_count) (Line 2), and (2) It extracts the relevant context for the processed packet (e.g., alarm on an IPS based on bad_conn_attempts_count) (Line 3) and then applies the corresponding policy (e.g., drop, forward) (Lines 4-5).

What should test traffic do? At a high level, the goal of test traffic is to exercise the policy context. Specifically, test traffic for a given policy needs to drive the data plane to a state corresponding to the context. Using Listing 1 as a reference, our aim is 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 become ≥ 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 system design.

Context-dependent policies: Let contextNF denote the processing context corresponding to packet pkt at NFᵢ (Line 3 of Listing 1). Then, the (network-wide) context of the packet is the sequence of contexts along the NFs it has traversed; i.e., if pkt has traversed NF₁, ..., NFᵢ, its context is context_pkt = ⟨contextᴺFᵢ, ..., contextᴺF₁⟩.

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

Policy : TrafficSpec × ContextSeq → PortSeq

Here, TrafficSpec is a predicate on the IP 5-tuple (e.g., source IP and transport protocol), ContextSeq is a sequence of contexts, 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:

⟨srcIP=Dept⟩, ⟨alarmᴸ₋IPS⟩ ↦

(L₋IPS → S₁, S₁ → S₂, S₂ → H₋IPS)

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

Stateful data planes: Contexts are convenient “short-hands” 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 NFᵢ = ⟨Sᵢ, Iᵢ, Portsᵢ, Tᵢ⟩, where Sᵢ is the set of NFᵢ states, Iᵢ is the initial state of NFᵢ, Portsᵢ is the set of ports of NFᵢ (where Portsᵢ ∈ Ports), and Tᵢ : Pkts × Portsᵢ × Sᵢ ↦ Pkts × Portsᵢ × Sᵢ is the stateful (as opposed to stateless, e.g., [54]) transfer function of NFᵢ.

We model intended packet dropping of certain NFs as sending packets to a virtual “drop port” of the NF. To help model the entire data plane, the topology function τ : Ports ↦ Ports captures the physical interconnection of NFs. Finally, we define the state of the data plane, Sᵈᵖ, as the conjunction of the states of its individual NFs.

There are many possible levels of abstraction to write such an FSM on, ranging from low-level code variables to high-level logical states (e.g., proxy cache state). Irrespective of this granularity, to be expressive for testing context-dependent policies, any FSM model of NFᵢ

²Without 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.
needs to provide a mapping from its states to the corresponding traffic specification and context:

\[ \text{stateToContextMap}_f : 2^S \rightarrow \text{TrafficSpec} \times C_i \]

To illustrate this, suppose in the example scenario of Figure 3 the operator defines an alarm as observing a host making at least 10 failed connection attempts. In Figure 6, we show two example ways of modeling L-IPS as an FSM. These are both expressive models of L-IPS, as in both Figures 6a and 6b, each of the red states maps to \((\text{srcIP}=\text{Deft}), (\text{alarm}_L\rightarrow \text{IPS})\). (In §5, we will discuss other requirements of an FSM-based NF model in addition to expressiveness.)

(a) Each state is of the form \((\text{badAttempCnt}_1, \text{badAttempCnt}_2)\) \(\langle\text{connStatus}_1, ..., \text{connStatus}_m\rangle\)

(b) Each state is of the form \((\text{badAttempCnt}_1, \text{badAttempCnt}_2)\) \(\langle\text{OK}, \text{OK}, ..., \text{OK}\rangle\)

**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, i.e., “take” the data plane to a state that corresponds the context (e.g., a red state in Figure 6). Thus, \(\text{trace} = \langle\text{pkt}_1, ..., \text{pkt}_m, ..., \text{pkt}_r\rangle\) is a test trace for policy \(\langle\text{trafficSpec}, \text{contextSeq}\rangle \rightarrow \text{portSeq}\) iff:

1. Each packet \(\text{pkt} \in \text{trace}\) satisfies \(\text{trafficSpec}\), and
2. \(\text{S}_D\) does not correspond to \(\text{contextSeq}\) after injection of each of packets \(\langle\text{pkt}_1, ..., \text{pkt}_{m-1}\rangle\), and
3. \(\text{S}_{DP}\) corresponds to \(\text{contextSeq}\) after injection and processing each of packets \(\langle\text{pkt}_m, ..., \text{pkt}_r\rangle\).

Conceptually, after \(\text{trace}\) is injected into the actual data plane, test resolution involves checking whether packets \(\langle\text{pkt}_m, ..., \text{pkt}_r\rangle\) actually traverse ports \(\text{portSeq}\).

**Takeaways:** Our formal framework has 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 policy-mandated 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 under test. Recall from §3 that this stage takes as input a library of NF models and the policy. The key challenge in building such a library is to model each type of NF (e.g., stateful firewall, web proxy) such that it is simultaneously (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 modeling approach (§5.1), we discuss how we achieve these goals by introducing a new abstract data unit for modeling input-output of NFs and describe how we can create scalable NF models via an ensemble-of-FSMs representation (§5.2). We conclude the section by describing how we construct the network-wide model by 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 [54, 63]. Each NF is modeled as: \(lp \leftarrow T(lp)\). Here, the input/output is a located packet \(lp = \langle\text{pkt}, \text{port}\rangle\), an IP packet (header) with its location in the network.

However, as we saw in §2, this abstraction is not expressive on several fronts w.r.t. state and context. To see how we can make it expressive, let us revisit our idealized NF abstraction 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 an almost immediate fix to these problems. First, instead of the stateless transfer function, we need to move to an FSM-like abstraction that captures state and the mapping from state to context. Second, we need some way to logically bind a packet to its relevant context. To this end, suppose 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 appends the policy-relevant context (depending on the current state) to 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 (e.g., if modeled naively, the FSM may have too many states to explore).

### 5.2 Modeling individual NFs

**The BUZZ Data Unit (BDU):** Start by presenting our approach to modeling the extended located packet idea we alluded to 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 [54, 63] 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. 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 that explicitly binds the BDU to its context (e.g., 1 bit for cache hit/miss, 1 bit for alarm/no-alarm). When an NF’s model receives an input BDU, it generates an output BDU with the updated c-Tag fields (e.g., a proxy that provides a cached response sets 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 of a practical reason we mentioned in §2: certain NFs (e.g., NATs, proxies) rewrite the original IP 5-tuple of a BDU. We, therefore, ensure the provenance sub-field of a p-Tag is left unchanged by NF models it goes through.

3. **Aggregation for scalability:** Each BDU can represent a sequence of packets associated with higher-layer NF operations. Via this aggregation, BDUs shrink the search space to find 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; specifically, the struct’s fields are 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 signify connection establishment semantics. Since each NF process 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:** There are numerous ways to expressively model a stateful NF as an FSM, but not all models are scalable. To see why, consider modeling the state-space as the concatenation of state variables (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 complementary avenues to shrink the state space of an NF:

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 being expressive, it is not scalable as the number of connections grow. By decoupling per-connection states, we obtain a firewall model as an ensemble of FSMs. In general, this insight cuts the number of states from \( |\text{state}|^{|\text{conn}|} \) to \( |\text{conn}| \times |\text{state}| \), 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 [19]) 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 \( \langle \text{client\_TCP\_state}, \text{server\_TCP\_state}, \text{cache\_content} \rangle \), we use an ensemble of three smaller FSMs, i.e., \( \langle \text{client\_TCP\_state} \rangle \), \( \langle \text{server\_TCP\_state} \rangle \), and \( \langle \text{cache\_content} \rangle \). 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 composable with other NF models. Second, instead of a monolithic FSM, it is partitioned into these three dimensions (i.e., client-, server-side connections and contexts) 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. 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 before sending it out.

5.3 Composing the data plane model
Next we discuss how to instantiate a model of the data plane by composing the models of individual NFs and given a policy. Listing 4 illustrates this for the network of Figure 2. We consider each switch as a static data store lookup updating BDUs [54]. Lines 8–10 model the stateless switch. 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. BUZZ uses the policy to automatically concretize the relevant model parameters (e.g., lines 3–4 specify which content/host to watch).

Similar to prior work [54,77], 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. Based on the policy, we identify the Next_NF in line 29. (As an optimization, our implementation pre-populates 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.

6 Test Traffic Generation
In this section, we show how to generate test traffic given the policies and the data plane model despite the key challenge of state space explosion. First we discuss 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 in order to generate abstract test traffic (i.e., traffic made of BDUs) (§6.2). We then 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 in interactive timescales, it should generate test traffic within seconds to a few minutes even for large networks. This is challenging on two fronts:

---

Listing 3: Proxy as an ensemble of FSMs.
```c
BDU Proxy(NFid id, BDU inBDU) {
  ...
  if ((frmClnt(inBDU)) && (isHttpRq(inBDU))) {
    if (cached(id, inBDU)) {
      outBDU = reqstFrmSrv(id, inBDU);
    } else {
      outBDU = tcpSYNtoSrv(id, inBDU);
    }
  } else {
    outBDU = rqstFrmSrv(id, outBDU);
  }
  //set c-Tags based on context (e.g., hit/miss)
  outBDU.c-Tags = ...
  ...
  return outBDU;
}
```

Listing 4: Data plane pseudocode for Figure 2.
```c
// Symbolic BDUs to be instantiated (see §6).
BDU A[20];
int objToWatch = XYZ.com;
int hostToWatch = H2;
// Global state variables
bool Cache[2][100]; // 2 proxies, 100 objects
// Model of a switch
BDU Switch(NFid id, BDU inBDU) {
  outBDU = lookUp(id, inBDU);
  return outBDU;
}
// Model of a monitoring NF
BDU Mon(NFid id, BDU inBDU) {
  ...
  outBDU = inBDU;
  if (isHttp(id, inBDU)) {
    takeMonAction(id, inBDU); /* if inBDU contains objToWatch destined to hostToWatch, set outBDU.dropped to 1.*/
  }
  return outBDU;
}
// Model of a proxy NF; See Listing 3
BDU Proxy(NFid id, BDU inBDU) {...}
main() {
  // Model of the data plane
  initializeProvenanceTags(A[]);
  for each injected A[]
    while (!DONE(A[])) {
      Forward A[] on current link;
      A[] = Next_NF(A[]);
      assert(!{(A[].p-Tag=hostId[12]) || ((A[].c-Tags[cacheContext]=objToWatch));})
    }
}
```
• **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 [53, 55, 77, 78]), we need to search over a very large traffic space of all possible sequences of traffic units. While BDUs improve scalability, as compared to IP packets, 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 NF1 and NF2 can reach K1 and K2 states, respectively, their composition will have K1 × K2 states.

Unfortunately, several canonical search solutions (e.g., model checking [4, 38] 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 SE, which is a well-known approach in formal verification to tackle state-space explosion [32]. 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 [34]. In our implementation, we use KLEE [33], which is a popular SE engine.

### 6.2 Generating abstract test traffic

BUZZ employs SE as follows. For each policy : trafficSpec × contextSeq → portSeq, first, we constrain the symbolic BDUs to satisfy the TrafficSpec. Then, to drive the SE engine to generate test traffic that satisfies contextSeq = (contextNF1, ⋯, contextNFN), we introduce the logical negation of contextSeq as an assertion in the network model code. In practice, assuming the policy context involves contexts of N NFs (i.e., C = context1 ∧ ⋯ ∧ contextN), BUZZ automatically instruments the network model with an assertion of the form ¬(context1 ∧ ⋯ ∧ contextN), where each term is expressed in terms of BDUs’ c-Tags 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, after being translated into concrete test traffic (§6.3) and injected into the actual data plane, must traverse 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 H2. 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 H2 (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 H2 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 H2 with the field httpGetObj set to httpObjId to induce a cached response. Similarly, Listing 5 shows an assertion in Lines 7-8 so that an alarm is triggered at both L-IPS and H-IPS of the example from Figure 3.

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 key optimizations:

- **Minimizing number of symbolic variables:** Making an entire BDU structure (Listing 2) symbolic forces KLEE to find values for every field. Instead, BUZZ uses the policy to identify the policy-related subset of each BDU fields and only make these symbolic and concretize the rest. For instance; e.g., when it is testing data plane with a stateful firewall but without a proxy, it makes the HTTP-relevant fields concrete (i.e., non-symbolic) by assigning the don’t care value * (represented by -1 in our implementation) to them.

- **Scoping values of symbolic variables:** The trafficSpec already scopes the range of values a BDU may take. BUZZ further narrows this range using the policy and protocols semantics to constrain the range of BDU fields. For example, even though the tcpSYN field is an integer, BUZZ constrain it to be 0/1.

#### 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, 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 [77]. Similarly, we

---

5Note that an assertion of the form ¬(A1 ∧ ⋯ ∧ An), or equivalently (¬A1 ∨ ⋯ ∨ ¬An), is violated only if each term Ai is evaluated to true.
define our test coverage goal as obtaining one test trace to exercise every 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 SE is an abstract test traffic sequence \( BDUSeq \). Since BDUs are abstract, we cannot directly inject them into the data plane. Moreover, we cannot simply do a one-to-one translation between BDUs and raw packets and do a trace replay [3, 77] because we need to honor session semantics (e.g., for TCP or FTP) of the policies. Note that 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.

We translate abstract test traffic into test traffic injection scripts that will be run on end hosts to inject concrete test traffic. The translation algorithm uses a library of traffic injection commands that maps a known \( BDUSeq \) 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 \( \text{wget} \) 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., \( \text{getHTTP}() \), \( \text{sendIPPacket}() \)) that support IP, TCP, UDP, FTP, and HTTP.

Here we give the intuition behind our translation algorithm. (For completeness, its pseudocode is presented in Appendix A.) Given a \( BDUSeq \), we use this library as follows. We partition the \( BDUSeq \) based on srcIP dstIP pairs (i.e., communication end-points) of BDUs; i.e., \( BDUSeq = \bigcup_i \text{BDUSeq}_i \). Then for each partition \( BDUSeq_i \), 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 extensions 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 \( Policy_{after} \) are 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 any data plane state \( s \in \text{Sdp} \), 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 \( IPS_1 \) scaled out.

Abstract test traffic generation for change management policies scenarios is slightly different from what we described in §6.1. At a high-level, for every data plane state \( s \in \text{Sdp} \), 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

In this section, we present the key aspects of our implementation of BUZZ. 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 source code is available at [1].

Operator interface: Operator can enter policies using either a text-based or a graphical interface (example screenshots in Appendix C). This is the only effort that BUZZ needs from the operator. 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). Once policies are entered, the workflow of BUZZ (Figure 5) is entirely automated.

NF models: We have written \( \mathbb{C} \) models for switches, ACL devices, stateful firewalls, NATs, L4 load balancers, HTTP and FTP proxies, passive monitoring, and simple intrusion prevention systems (counting failed connection attempts and matching payload signatures). Our models are between 10 (for a switch) to 100 lines (for a proxy cache) of \( \mathcal{C} \) 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, 23] 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 the BDU-level test traffic, and then translate it to test scripts that are deployed and run at the injection points.

Test traffic monitoring and test resolution: The entire workflow of BUZZ is implemented atop OpenDayLight. 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 10GbE 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 [19] as a proxy, Snort [18] and Bro [66] as IPS/IDS, Balance [2], and PRADS [16].

In addition to the example scenarios from §2, we use 8 randomly selected recent topologies from the Internet Topology Zoo [20] 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 by connecting each stateful NF to a randomly selected switch. As a concrete policy enforcement scheme, we used prior work to handle dynamic middleboxes [45]. (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 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 [48]: We used the OpenNF-enhanced PRADS [16, 48] to enforce the following policy: if a host spawns more than $\text{Thresh}$ 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: $n_1, n_2 < \text{Thresh}$, but $n_1 + n_2 > \text{Thresh}$. 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 $\text{PRADS}_1$ to $\text{PRADS}_2$.

• Faulty policy composition using PGA [67]: We used PGA to compose two policies on traffic from $H_1$ to $H_2$: it should pass a load balancer and a stateful firewall ($\text{policy}_1$), and if it is found suspicious, it then should go to an IPS ($\text{policy}_2$). After enforcing the composition of the two policies, BUZZ found that the test traffic exercising $\text{policy}_1$ did not go through the firewall. This is because the SDN switch rules corresponding to $\text{policy}_1$ took precedence over the switch rules for $\text{policy}_2$, rendering $\text{policy}_2$ ineffective.

• Incorrect tagging using FlowTags [45]: 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, (a commonly used practice in network security [21]) 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 (a) identify a violation, and (b) localize the source of the policy violation. We also repeated these experiments reversing student roles; but do not show these results for brevity.

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 con-

---

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

7Our original implementation [45] was in POX [15].
Table 1: Example red-blue team scenarios.

<table>
<thead>
<tr>
<th>“Red Team” scenario</th>
<th>BUZZ test trace</th>
<th>Violating NF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cascaded NATs using Click IPRewriter [56] : NAT1 incorrectly rewrites srcIP triggering “assertion failure” on NAT1 [40]</td>
<td>$H_1$ attempts to access to the server</td>
<td>NAT1</td>
</tr>
<tr>
<td>Multi-stage triggers (Fig. 3), L-IPS miscounts by summing three hosts</td>
<td>$H_1$ makes 9 scan attempts followed by 9 scans by $H_2$</td>
<td>L-IPS</td>
</tr>
<tr>
<td>Conn. limit.; Login counter resets</td>
<td>$H_1$ makes 3 continuous log in attempts with a wrong password</td>
<td>Login counter</td>
</tr>
<tr>
<td>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</td>
<td>A tcp connection from internal $C_1$ to external $S_1$ followed by an access from $S_1$ to $C_1 : port443$</td>
<td>Firewall</td>
</tr>
<tr>
<td>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</td>
<td>a tcp connection followed by tcp data packets</td>
<td>switch close to dst</td>
</tr>
</tbody>
</table>

To give a more practical sense of BUZZ, it is useful to discuss three aspects of its test coverage. 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. Here we observed that test generation latency grows linearly with the intended number of test cases (not shown for brevity). 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.

BDU vs. packet: 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
Impact of SE optimizations: We examine the effect of the SE-specific optimizations (§6.2) in Figure 8. To put 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) scaling the values yields a further > 9× reduction.

9 Related work

Network verification: There is rich literature on reachability checking on configurations [42, 46, 53, 54, 58, 59, 75, 76]. The work closest to BUZZ is ATPG [77]. As discussed earlier, these do not capture the stateful behaviors and context-dependent policies.

Code verification: The work in [41] focuses on finding Click [56] 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 [35]. BUZZ is complementary to these efforts.

Modeling stateful networks: Joseph and Stoica formalized middlebox forwarding behaviors but do not model stateful behaviors [52]. The only work that models stateful behaviors are FlowTest [43], Symnet [72], and the work by Panda et al [65]. FlowTest’s high-level models are not composable and the AI planning approaches do not scale beyond 4-5 node networks. Symnet [72] 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).

Policy enforcement: There are several frameworks to facilitate policy enforcement [10, 45, 48, 64, 67, 68]. There are also efforts to generate correct-by-construction SDN programs [27, 29, 47]. Our work is complementary in that it enables checking whether the intended behavior manifests correctly in the actual data plane.

Simulation and shadow configurations: Simulation [13], emulation [6, 11], and shadow configurations [26] are common methods to model/test networks.

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., [37]) or logs (e.g., [30]).

Soundness vs. completeness: For “infinite-state” systems, it is not possible to simultaneously achieve both guarantees [51]. 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 don’t 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 [51, 77] (e.g., see §8.2).

New use cases of BUZZ: Looking forward, we believe BUZZ can be extended to systematically check interoperability of new protocols with middleboxes [50]. As preliminary evidence, we were able to replicate a known problem with a middlebox-cooperative TCP extension called HICCUPS [39], 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


A Translating abstract test traffic into test traffic injection scripts

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

```
1 // Inputs:
2 #1: sequence of BDUs from Symbolic Execution
3 #2: abstract test traffic
4 #3: change management policies
5 Outputs:
6 #1: BDUseq<sup>SK</sup> = (BDU<sub>n</sub> : n = 1, 2, ... N)
7 #2: cmdlibBDU
8 #3: Command scripts for change management policies
9 #4: Outputs:
10 Outputs: BDUseq<sup>SK</sup> and BDUseq<sup>SK</sup> after which should satisfy:
11 BDUseq<sup>SK</sup> after exploits all possible context in C<sub>i</sub> before migration/rollback happens.
12 BDUseq<sup>SK</sup> after test all possible context in C<sub>i</sub> after the migration/rollback.
13 #5: a set of end-hosts H = {H<sub>i</sub> : i = 1, 2, ..., K} to execute cmd
14 #6: a number of scripts S = {script<sup>SK</sup><sub>1</sub> ... script<sup>SK</sup><sub>n</sub>}
15 #7: Do Decompose BDUseq<sup>SK</sup> into sequences
16 BDUs of previous BDUs with same predicate
17 #8: Sort each cmd<sub>n</sub> in cmdlibBDU with most BDUs to least BDUs
18 #9: Instantiate script<sup>SK</sup><sub>1</sub> to store cmd for BDUseq<sup>SK</sup><sub>prod</sub>
19 #10: Script<sup>SK</sup><sub>1</sub> empty
20 #11: Match the BDUs in BDUseq<sup>SK</sup><sub>prod</sub> with cmdlibBDU from cmdlibBDU
21 #12: For each cmd<sub>n</sub> in cmdlibBDU
22 #13: if BDU<sub>n</sub> is a BDU belonging to BDUseq<sup>SK</sup><sub>prod</sub> started at BDU<sub>n</sub>
23 #14: if BDU<sub>n</sub> lower part BDUs<sup>SK</sup><sub>prod</sub>, len(Seq<sup>SK</sup><sub>prod</sub>) == Seq<sup>SK</sup><sub>prod</sub>
24 #15: add the matched cmd<sub>n</sub> and the first matched BDU’s index n
25 #16: sort every cmd<sub>n</sub> in script<sup>SK</sup><sub>prod</sub> by its first matched BDU’s index n
26 #17: Map abstract pred<sub>n</sub> to real test host H<sub>prod</sub> and assign script to host
27 script<sup>SK</sup><sub>prod</sub> = script<sup>SK</sup><sub>prod</sub>
```

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).

1 // Inputs:
2 #1: packet traces pkttrace<sub>prod</sub> dumped at each port in Ports
3 #2: policy Test: pred(5-tuple) × Ports, where C includes all possible contexts
4 Outputs:
5 #1: the resolution result of each context context in C in terms of pass/fail
6 #2: the port of the NF that causes the failure
7 #3: perform resolution scheme for each context context, in C
8 for each context context in C:
9 if Trace = {pkt<sub>1</sub>, ... pkt<sub>n</sub>} is the test packets for this context
10 for testpkt in {pkt<sub>1</sub>, ... pkt<sub>n</sub>}
11 #1: calculate the logically correct ports testpkt should reach
12 Ports<sub>p<sub>pred</sub><sub>prod</sub></sub> = Policy(pred(testpkt), context)
13 #3: find the real ports testpkt has reached
14 Ports<sub>p<sub>real</sub></sub> = search testpkt in each pkttrace<sub>prod</sub>
15 #1: if Ports<sub>p<sub>real</sub></sub> = Ports<sub>p<sub>pred</sub></sub>
16 context, test pass
17 else context, test fail
18 #1: Compare the port of Ports<sub>p<sub>pred</sub></sub> and Ports<sub>p<sub>real</sub></sub> and find the first different port, which is the NF that causes the failure.
19 FailedNFPort = FirstDiffPort(Ports<sub>p<sub>pred</sub></sub>, Ports<sub>p<sub>real</sub></sub>)
20 context, test failed

Figure 14: Pseudocode for BUZZ test resolution.