BUZZ: Testing Context-Dependent Policies in Stateful Data Planes

Seyed K Fayaz, Yoshiaki Tobioka, Sagar Chaki, Vyas Sekar

September 25, 2014

CMU-CyLab-14-013

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

BUZZ: Testing Context-Dependent Policies in Stateful Data Planes

Seyed K. Fayaz[†]*, Yoshiaki Tobioka*[†]*, Sagar Chaki*[+]*, Vyas Sekar*[†]
[†] *CMU,* [+] *SEI*

Abstract

Network operators spend significant effort in ensuring that the network meets their intended policies. While recent work on checking reachability and isolation policies have taken giant strides in this regard, they do not handle *context-dependent* policies that operators implement via *stateful* data plane elements. To address this challenge, we present the design and implementation of BUZZ, a testing framework to ensure that a network with stateful data plane elements meets complex context-dependent policies. In designing BUZZ, we address significant challenges in: (1) modeling stateful data plane elements, and (2) tackling the state-space explosion problem in generating test scenarios. We also implement practical heuristics to resolve interference from background traffic and to localize sources of policy violations. We demonstrate the scalability of BUZZ in localizing policy violations on networks with more than 100 nodes.

1 Introduction

Many studies highlight the difficulty network administrators face in correctly implementing policies. For instance, one recent survey found that 35% of networks generate more than 100 problem tickets per month and nearly one-fourth of these problem tickets take multiple engineer-hours to resolve [14]. Anecdotal evidence suggests that operators go to great lengths to debug networks; e.g., creating "shadow" configurations of entire infrastructures for testing [25].

This challenge has inspired several projects, including work on statically checking networks [34, 35, 36], formal foundations of networks [17, 31, 41], creating correct-by-construction controllers [19], verifying software data planes [26], automatic test packet generation [55], and debugging control plane software [24, 49]. However, these efforts largely focus on forwarding-centric properties (e.g., loops, black holes) and layer 2/3 data plane functions (i.e., simple switches, routers, ACLs).

While checking reachability properties involving switch configurations is clearly useful, real networks are more complex on two dimensions:

- **Stateful and higher-layer network functions:** Networks rely on a variety of data plane middleboxes and switches [50]. We use the term DPFs to collectively refer to such *data plane functions*, including: (i) stateless L2/L3 elements; and (ii) stateful middleboxes (henceforth called stateful DPFs) that operate at a higher-layer beyond L2/L3 and whose actions depend on the history of traffic; e.g., a proxy operates over HTTP requests and may send cached responses.
- **Context-dependent policies:** Operators use stateful DPFs to implement advanced policies beyond simple forwarding and access control rules. A simple policy is *service chaining*, i.e., HTTP traffic going through a *sequence* of a firewall, IPS, and proxy before exiting the network. Complex policies involve context-dependent information; e.g., a host generating too many failed connections may be flagged as anomalous and rerouted for detailed analysis [10, 18].

Unfortunately, such stateful operations and their attendant context-dependent policies fall outside the scope of the aforementioned network verification and testing tools. Our goal is to address this key missing piece to take us closer to the "CAD-for-networks" vision [39].

In this paper, we present the design and implementation of BUZZ,[1] a framework for testing if a network with stateful DPFs meets specified context-dependent policies. At a high level, BUZZ is a model-based tester [52]. Specifically, given an intended behavioral specification of the network (i.e., all stateful and stateless DPFs and their interconnections) BUZZ generates test traces that exercise specific policies, and then injects them into the real network to see if the observed behavior matches the intended policy. We make key contributions in addressing two related challenges to make this vision practical:

- **Data plane modeling (§5):** While, conceptually, a stateful DPF is a "giant" finite state machine operating on raw IP packets, it is intractable to enumerate all possible states for all possible input packet sequences. To make DPF models tractable, BUZZ uses two key ideas. First, instead of modeling a DPF's operations as a "giant FSM", we model each DPF as a *FSM ensemble* that mirrors the conceptual separation across functions inside the actual DPF's implementation; e.g., a proxy keeps a separate TCP state machine for each client and server. Second, rather than viewing DPFs as operating on low-level packets, we model their input-output behaviors in terms of a new notion of *BDUs* (BUZZ Data Units), which are abstract data units that succinctly capture higher-layer semantics spanning multiple low-level IP packets and also explicitly model the impact of stateful/context-dependent DPF actions. For instance, a full HTTP response can be represented by a single BDU "packet", instead of many low-level packets. Similarly, BDUs allows our DPF models to expose hidden behaviors [29].

[1] BUZZ "explores space"; e.g., "Buzz" Aldrin and Buzz Lightyear.

- **Test traffic generation (§6):** For reasonably interactive testing, new test cases must be generated in seconds. Unfortunately, even if we address the above modeling challenge, we run into scalability issues while trying to exercise a policy-specific sequence of effects due to the well-known state-space explosion problem [21]. To this end, we use a combination of three key ideas. First, we replace exhaustive state-space search with a more efficient *symbolic execution* based approach. Second, we leverage our BDU abstraction to first generate a high-level plan in terms of BDUs and then translate it into raw test traces. Finally, we engineer domain-specific optimizations (e.g., restricting number of symbolic variables) that allow symbolic execution to scale to large networks.

We implement models for various DPFs as FSM ensembles written in C. Our choice of C over a domain-specific language [19, 31, 41] immediately lends BUZZ to a body of tools optimized for symbolic execution of C such as KLEE [22]. We implement our domain-specific optimizations on top of KLEE. We developed a custom translation from BDU sequences generated by KLEE to "raw" request traces used to test the real network. We prototype the overall test orchestration capabilities atop `OpenDaylight` [8]. Finally, given that we are performing tests on the actual network, we engineer heuristics leveraging SDN-based monitoring capabilities to rule out side effects from background traffic. We also implement practical heuristics to help localize diagnostic efforts when policy violations are detected (§7).

Our evaluations on a real testbed, shows that BUZZ:

- can test hundreds of policy scenarios on networks with ≥ 100 nodes in tens of seconds;
- dramatically improves scalability, providing nearly three orders of magnitude reduction in time for test case generation;
- effectively localizes intentional data/control plane bugs within tens of seconds; and
- imposes less than 1% overhead in terms of additional traffic even with adversarially interfering traffic.

2 Motivating Scenarios

In this section, we use small but realistic network scenarios to: (i) highlight *stateful data plane functions* and *context-dependent policies* used by administrators; (ii) motivate challenges in implementing these policies correctly; and (iii) present limitations of existing work (on L2/3 reachability) in addressing these challenges.

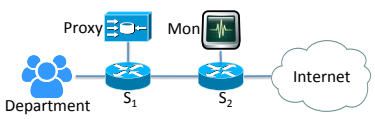


Figure 1: Dynamic cache actions.

Stateful firewalling: Today's firewalls go beyond the traditional match-then-drop paradigm. A common policy is reflexive firewalling; i.e., the firewall tracks outgoing connections from internal hosts and allows incoming packets for previously established connections. Unfortunately, even this basic stateful processing cannot be handled by existing "memoryless" static checkers. For instance, ATPG [55] and HSA [35] can only check for single packet effects and cannot model connection establishment logic and reflexive rules.

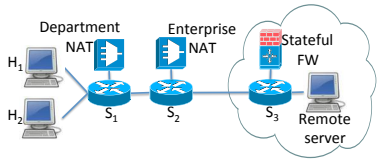


Figure 2: Blocking policy is difficult due to NATs.

Dynamic policy violations: In Figure 1, we want to monitor outgoing web traffic. The problem here is subtle, as the proxy may send cached responses bypassing the monitor, defeating our goal. Potential fixes to this problem include placing the monitor before the proxy or using `FlowTags` to explicitly steer cached responses [29]. The key challenge is in identifying such potential policy violations and ensuring that the solution (e.g., using `FlowTags`) is implemented correctly[2]. Specifically, to identify this policy violation, we need to model the stateful behavior of the proxy across connections.

Firewalling with Cascaded NATs: Figure 2 shows a scenario where hosts are doubly NAT-ed – at the department and the enterprise border. Prior work shows cascaded NATs are notoriously error-prone [20, 42]. Suppose the remote web server's firewall needs to to block host H_1 but allow H_2. Even this seemingly easy access control policy is difficult to check with existing L2/L3 reachability tools because the hosts are hidden behind NATs; e.g., HSA/ATPG models them as "black-boxes'.

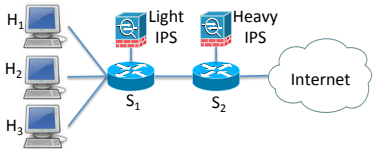


Figure 3: Dynamic "triggers".

Multi-stage triggers: Figure 3 shows two intrusion prevention systems (IPS) used in sequence. The intended policy is to use the light-weight IPS (L-IPS) in the common case and only subject suspicious hosts flagged by the L-IPS (e.g., when a host generates scans) to the

[2] BUZZ was partly motivated by the failure of existing tools to validate that our FlowTags implementation fixes the problem.

more expensive H-IPS. Such multi-stage detection is useful; e.g., to minimize latency and/or reduce H-IPS load. Again, it is difficult to check that this multi-stage operation works correctly using existing static checkers and reachability verifiers [35,55], because they ignore hidden state inside the IPSes and triggered sequence of intended actions (i.e., the context).

	Stateful data planes	**Contextual policies**	**Active testing**
Test packet generation (e.g., [55])	No	No	Yes
Static verification (e.g., [34,35,36,38])	No	No	No
Verifying data plane software (e.g., [26])	Yes	No	No

Table 1: Strawman solutions (rows) vs. requirements from the motivating scenarios (columns).

Strawman solutions: The above scenarios imply three key requirements: (1) capturing stateful data plane behaviors (e.g., stateful firewalls); (2) capturing contextual policies (e.g., LIPS-HIPS); and (3) active testing to reveal subtle traffic-dependent bugs that may not be directly visible from just looking at the network configuration (e.g., dynamic cache actions or cascaded NATs).

Table 1 summarizes if/how some existing solutions address the scenarios described earlier. Across all motivating scenarios, we find that existing tools for checking network policies are inadequate. At a high level, the problem is that many existing tools for network reachability testing explicitly acknowledge these challenges and treat them as being out-of-scope to make their work tractable. While there is some recent work on testing software data planes, the focus is on different types of errors (e.g., crash or CPU cycles/packet) rather than the network-wide policy violations we consider here. Our overarching goal is to bring the benefits that these aforementioned efforts have provided for reachability correctness to the types of stateful network processing and contextual policies introduced by the above scenarios.

3 Problem Formulation

Our high-level goal is to help network administrators *test* that the data plane implements the intended policies. In this section, we begin by formally defining our intended data plane semantics, and what we mean by a policy. In addition to helping us precisely define our goals, the formalism sheds light on the key technical components and challenges underlying any solution for testing stateful data planes for the given context-dependent policies.

3.1 Preliminaries

First, we define the semantics of a DPF and the network.

DPF: Let $\mathscr{P}$ denote the set of packets.[3] Formally, a DPF is a 4-tuple (S, I, E, δ) where: (i) S is a finite set of states; (ii) $I \in S$ is the initial state; (iii) E is the set of network edges; and (iv) $\delta : S \times \mathscr{P} \mapsto S \times \mathscr{P} \times E \times \Sigma$ is the transition relation.

Here, Σ is a set of *effects* that capture the response of a DPF to a packet. Each $\alpha \in \Sigma$ provides contextual information that the administrator cares about. Each α is annotated with the specific DPF generating the effect and its relevant states; e.g., in Figure 3 we can have $\alpha_1 = \langle LIPS : H_1, Alarm, SendToHIPS \rangle$ when the *LIPS* raises an alarm and redirects traffic from H_1 to the H-IPS, and $\alpha_2 = \langle LIPS : H_1, OK, SendToInternet \rangle$ when the *LIPS* decides that the traffic from H_1 was OK to send to the Internet. Using effects, administrators can define high level policy intents rather than worry about low-level DPF states. Note that this DPF definition is general and it encompasses stateful DPFs from the previous section and stateless L2-L3 devices.

Network: Formally, a network data plane *net* is a pair (N, τ) where $N = \{DPF_1, \ldots, DPF_N\}$ is a set of DPFs and τ is the topology map. Informally, if $\tau(e) = DPF_i$ then packets sent out on edge e are received by DPF_i.[4] We assume that the graph has well-defined sources (with no incoming edges), and one more sinks (with no outgoing edges). The *data plane state* of *net* is a tuple $\sigma = (s_1, \ldots, s_N)$, where s_i is a state of DPF_i.

3.2 Processing semantics

To simplify the semantics of packet processing, we assume packets are processed in a lock-step (i.e., one-packet-per-DPF-at-a-time) fashion and do not model (a) batching or queuing effects inside the network (hence no re-ordering and packet loss); (b) parallel processing effects inside DPFs; and (c) the simultaneous processing of different packets across DPFs.

Let $\sigma = (s_1, \ldots, s_i, \ldots, s_N)$ and $\sigma' = (s_1, \ldots, s'_i, \ldots, s_N)$ be two states of *net*. First, we define a *single-hop* network state transition from (σ, i, π) to (σ', i', π') labeled by effect α, denoted $(\sigma, i, \pi) \xrightarrow{\alpha} (\sigma', i', \pi')$ if $\delta_i(s_i, \pi) = (s'_i, \pi', e, \alpha)$, with $DPF_{i'} = \tau(e)$. A *single-hop* network state transition represents processing of one packet by DPF_i while the state of all DPFs other than DPF_i remains unchanged. For example, when the L-IPS rejects a connection from a user, it increments a variable tracking the number of failed connections. Similarly, when the stateful firewall sees a new three-way handshake completed, it updates the state for this session to `connected`.

Next, we define the *end-to-end* state transitions that a packet π^{in} entering the network induces. Suppose π^{in} traverses a path of length n through the sequence of DPFs $DPF_{i_1}, \ldots, DPF_{i_n}$ and ends up in $DPF_{i_{n+1}}$ (note

[3] Packets are "located" [35,46], so that the DPF can identify and use the incoming network interface information in its processing logic.

[4] We assume each edge is mapped to unique incoming/outgoing physical network ports on two different DPFs.

that the sequence of traversed DPFs may be different for different packets). Then the end-to-end transition is a 4-tuple $(\sigma_1, \pi^{in}, \langle \alpha_1, \ldots, \alpha_n \rangle, \sigma_{n+1})$ such that there exists a sequence of packets $\pi_1, \ldots, \pi_{n+1}$ with $\pi_1 = \pi^{in}$, and a sequence of network states $\sigma_2, \ldots, \sigma_{n-1}$ such that $\forall 1 \leq k \leq n$: $(\sigma_k, i_k, \pi_k) \xrightarrow{\alpha_k} (\sigma_{k+1}, i_{k+1}, \pi_{k+1})$.

That is, the injection of packet π^{in} into DPF_{i_1} when the network is in state σ_1 causes the sequence of effects $\langle \alpha_1, \ldots, \alpha_n \rangle$ and the network to move to state σ_{n+1}, through the above intermediate states, while the packet ends up in $DPF_{i_{n+1}}$. For instance, when the L-IPS is already in the `toomanyconn-1` state for a particular user and the user sends another connection attempt, then the L-IPS will transition to the `toomanyconn` state and then the packet will be redirected to the H-IPS.

Let $E2ESem(net)$ denote the end-to-end "network semantics" or the set of feasible transitions on the network *net* for a single input packet.

Trace semantics: Next, we define the semantics of processing of an input packet trace $\Pi = \pi_1^{in}, \ldots, \pi_m^{in}$. We use $\vec{\alpha}$ to denote the vector of DPF effects associated with this trace; i.e., the set of effects across all DPFs in the network. The network semantics on a trace Π is a sequence of effect vectors: $TraceSem_\Pi = \langle \vec{\alpha}_1, \ldots, \vec{\alpha}_m \rangle$ where $\forall 1 \leq k \leq m$: $\pi_k^{in} \in \mathscr{P} \wedge \vec{\alpha}_k \in \Sigma^+$. This is an acceptable sequence of events iff there exists a sequence $\sigma_1, \ldots, \sigma_{m+1}$ of states of *net* such that: $\forall 1 \leq k \leq m$: $(\sigma_k, \pi_k^{in}, \vec{\alpha}_k, \sigma_{k+1}) \in E2ESem(net)$.

3.3 Problem Definition

Given the notion of trace semantics defined above, we can now formally specify our goal in developing BUZZ. At a high-level, we want to test a *policy*. Formally, a policy is a pair $(TraceSpec; TraceSem)$, where *TraceSpec* captures a class of traffic of interest, and *TraceSem* is the vector of effects of the form $\langle \vec{\alpha}_1 \ldots \vec{\alpha}_m \rangle$ that we want to observe from a correct network when injected with traffic from that class. Concretely, consider two policies:

1. In Figure 1, we want: *"Cached web responses to Dept1 should go to the monitor"*. Then, *TraceSpec* captures web traffic to/from Dept1 and $TraceSem = \langle \alpha_1, \alpha_2 \rangle$, with $\alpha_1 = Proxy : Dept1, CachedObject$ and $\alpha_2 = Proxy : Dept1, SendToMon$.
2. In Figure 3 we want: *"If host H_1 contacts more than 10 distinct destinations, then its traffic is sent to $H-IPS$"*. Then, *TraceSpec* captures traffic from H_1, and $TraceSem = \langle \alpha_1, \alpha_2 \rangle$ where $\alpha_1 = L-IPS : H_1, Morethan10Scan$, and $\alpha_2 = L-IPS : H_1, SendtoHIPS$.

Our goal is to check that such a policy is satisfied by the *actual* network. More specifically, if we have a *concrete test trace* Π that satisfies $TraceSpec_\Pi$ and should *ideally* induce the effects $TraceSem_\Pi$, then the network should exhibit $TraceSem_\Pi$ when Π is injected into it.

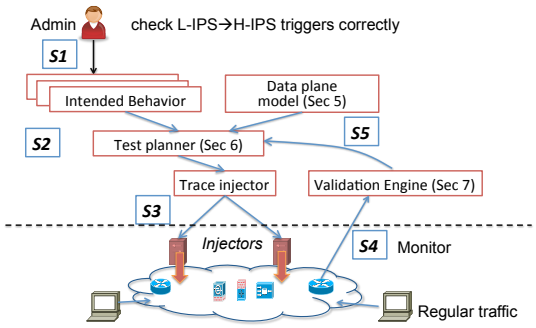


Figure 4: High-level overview of BUZZ.

In practice, generating these concrete test traces is tedious as it requires understanding and dealing with the complex low-level behaviors of DPFs. The goal of BUZZ is to automate this test trace generation. That is, the administrator gives a high-level specification of *TraceSpec* (e.g., Web traffic from/to Dept1) and *TraceSem*, and BUZZ generates a concrete test trace, injects it into the network, and checks if it satisfies the policy. Next, we discuss how BUZZ achieves this goal.

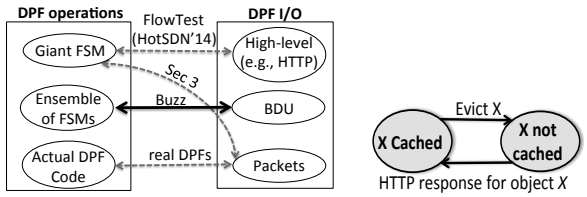
4 BUZZ System Overview

Figure 4 shows the main components of BUZZ. The input to BUZZ is the administrator's policies (**S1**). In practice, we envision that administrators will define these policies in a higher-layer language that can then be translated to more formal $(TraceSpec; TraceSem)$ entries. As such, the design of this policy language is outside the scope of this paper. These input policies refer to physical DPFs (e.g., a NAT followed by a firewall), which can be obtained from policies in terms of corresponding logical DPFs [45]. We envision operators populating a set of such policies.[5] Given these, BUZZ processes these one policy a time as we describe next.

As we saw in the previous section, the effects depend on the processing semantics of the individual DPFs and the data plane as a whole. Thus, BUZZ needs a *model* of the entire data plane (i.e., DPFs and the interconnections). Writing the DPF models is a one-time task for each type of DPF and we envision this can be provided by DPF vendors and other domain experts working together. Note that our simplifying assumption on packet processing semantics (§3.2) only apply to the data plane model not the physical data plane.

Given the data plane model and one policy, BUZZ generates a *test plan* (**S2**). In essence, the test plan satisfies the *TraceSpec* and causes the *model data plane* to exercise the specific *TraceSem*. In practice, we decouple this into two stages: (i) generating a high-level plan in terms of abstract entities called BDUs; and (ii) generating a

[5] BUZZ cannot discover violations of behaviors that cannot be expressed as policies. Thus, BUZZ is not a tool to discover new bugs; rather it tests if the behavior of the real data plane matches the policies.



(a) Design space of DPF models (b) High-level proxy

Figure 5: Illustrating challenges in choosing a suitable option of I/O and FSM state granularity in balancing model tractability, fidelity, and composability.

concrete test manifest (i.e., scripts that create test traffic) to feed into the *injectors*. The injectors are regular hosts or servers running test traffic generators or other trace injection software that execute the test manifest (**S3**).

Then, BUZZ monitors the network and determines whether this test passed/failed (**S4**). That is, while we know that the generated test trace will cause the effect *TraceSem* on the *model data plane*, this phase determines if the physical data plane also shows *TraceSem*. Finally, we also envision additional test cases for further diagnostics to localize the causes for the policy violation (e.g., broken link, middlebox misbehavior) (**S5**).

Given this high-level view, in the following sections we highlight the key challenges in making this vision practical and our solutions to address these challenges.

5 Data Plane Modeling

We begin by highlighting the challenges in balancing fidelity vs. tractability and composability of DPF models. Then, we describe the two key modeling ideas we introduce in BUZZ– BDU and FSM ensembles– that achieve a good balance across these requirements.

5.1 Strawman solutions

Modeling a DPF requires us to fix the granularity of input/output and FSM operations. Figure 5a depicts the space of modeling strategies along these two dimensions. To understand the challenges in deciding these two granularities, we consider three strawman solutions.

1. *Use the "giant" FSM formalism from §3 (i.e., the 4-tuple (S,I,E,δ)):* However, writing down this FSM at a packet granularity is tedious and error-prone. For instance, to model state transitions involving a single HTTP request/reply in a proxy, we need to model the sub-transitions for tens of packets. Furthermore, it is infeasible to explicitly write down all states, as it requires enumerating all input packet sequences.
2. *Use the code as the "model":* This makes test traffic generation challenging because of the code complexity. For instance, `Squid` has $\geq$ 200K lines of code and introduces other sources of complexity that are irrelevant to the policies being checked. Also, we may not have the source code, or the code may not match the policy due to bugs.
3. *Write DPFs at a very high-level and focus on "relevant" states and inputs:* This is labeled as "High-level" in Figure 5a); e.g., write the proxy in terms of HTTP object requests/responses as shown in Figure 5b.[6] Given the diversity of DPF operations that act on different layers, such models are fundamentally non-composable as the input-output granularity of different DPFs will not match; e.g., we cannot simply "chain" the output of a proxy operating at this level to a packet-level firewall as we did in §3.

Next, we describe our solution to the granularity problem. We retain the lock-step processing semantics from §3 and introduce two key ideas to address the shortcomings of the above strawman solutions: (1) the BDU abstraction for input-output behaviors and (2) FSM ensembles for FSM operations.

5.2 BDU abstraction

First, we tackle the issue of the input-output granularity. Our observation is to ensure composability of DPF models, we want a *located packet*-like abstraction [35] since it is a natural "lowest common denominator" across diverse DPFs. A located packet is simply a packet along with a specific network interface denoting its location. However, a packet (even a located packet) is too low-level to express the desired DPF semantics. To this end, we introduce the *BDU* or *Buzz Data Unit*, that extends the notion of a located packet in three main ways.

First, we allow each BDU to represent a sequence of packets rather than an individual packet. The reason is that many "relevant" effects inside a DPFs occur on sets of packets rather than a single IP packet. For example, the proxy's cache state transitions to an "relevant state" (i.e., cached state w.r.t. an object) only after the entire payload has been reassembled. Second, the BDUs capture relevant *features* that might be relevant for the test scenarios; e.g., the notion of a HTTP request/response. Third, BDUs effectively encode the effect semantics of the DPFs along its path in the form of `alphaTags`. Conceptually, we can view an `alphaTag` as an encoding of a specific effect $\alpha \in \Sigma$ from our formalism. This ensures that the BDU carry its "policy-related processing history" as it goes through the network. Note that in the base case, a BDU is a simple IP packet, but BDUs give us the flexibility to define higher-layer operations as well.

Intuitively, BDUs reduce modeling complexity by consolidating different protocol semantics (e.g., HTTP, TCP) and effects involving multiple IP packets (e.g., all packets corresponding to a HTTP reply are represented by one BDU with the httpRespObj field indicating the retrieved object id). Concretely, a BDU is simply a *struct* as shown in Listing 1. Note that the struct fields are a

[6] In fact, this was the approach in our early workshop paper [28].

Listing 1: BDU structure.

```
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 DPF conceptually records its effect
13  int alphaTag[MAXTAG];
14  ...
15 };
```

superset of required fields of the DPFs. Each DPF processes only fields relevant to its function (e.g., the switch function ignores HTTP layer fields of input BDUs).

While we do not claim to have a BDU definition that can encompass all possible network contexts and policy requirements, we suggest a high-level roadmap that has served us well. Specifically, the key to determining the fields of a BDU is to consider all DPFs of interest and identify policy-related state transitions in DPFs of interest. For example, each of `TCP SYN`, `TCP SYN+ACK`, etc. make important state transitions in a stateful firewall and thus should be captured as BDU fields.

5.3 Modeling DPFs as FSM ensembles

We now address the issue of FSM granularity. Here our insight is to borrow from the design of actual DPFs. In practice, DPF programs (e.g., a firewall) do not explicitly enumerate the full-blown FSM. Rather, they have an *implicit* model; e.g., the state machines are created for the subset of observed packets and the different functional components of the DPF are naturally segmented.

To understand this better, consider a proxy. A proxy is instructive because it is quite complex – it operates on a higher layer of sessions, terminates sessions, and it can respond directly with objects in its cache. The code of a proxy, e.g., `Squid`, effectively has three independent modules: TCP connections with the client, TCP connection with the server, and cache. While the proxy is effectively the "product" of these modules, modeling it by computing the product explicitly is not practical as this leads to state-space explosion.

Listing 2 shows a partial code snippet of the proxy model, focusing on the actions when a client is requesting a non-cached HTTP object and the proxy does not currently have a TCP connection established with the server. Here the `id` allows us to identify the specific proxy instance. The specific state variables of different proxy instances are inherently partitioned per DPF instance (not shown). These track the relevant DPF states, and are updated by the DPF-specific functions such as `srvConnEstablished`.[7]

[7]This choice of passing "id"s and modeling the state in per-id global variables is an implementation artifact of using C/KLEE, and is not fundamental to our design.

Listing 2: Proxy as an FSM ensemble.

```
1  BDU  Proxy(DPFId 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    }
10 }
11 /*set alphaTags based on context (e.g.,
12 cache hit/miss, client ip)*/
13 outBDU.alphaTags = ...
14 ...
15 return outBDU;
16 }
```

If the input `inBDU` is a client HTTP request (Line 3), and if the requested object is not cached (Line 4), the proxy checks the status of TCP connection with the server. If there is an existing TCP connection with the server (Line 5), the output BDU will be a HTTP request (Line 6). Otherwise, the proxy will initiate a TCP connection with the server (Line 8).

This example shows that by decoupling the three stateful aspects of the proxy (i.e., client/server-side TCP connections and cache contents) we can move away from an FSM model of a proxy with each state being of the form $\langle client\_TCP\_state, server\_TCP\_state, cache\_content \rangle$ to a simpler *ensemble* of three smaller FSMs each with a single type of state, i.e., $\langle client\_TCP\_state \rangle$, $\langle server\_TCP\_state \rangle$, and $\langle cache\_content \rangle$. In other words, we represent the product implicitly, and thereby avoid state space explosion.

Each DPF encodes the relevant effect in the `alphaTag` field of the outgoing BDU as shown in Line 13. There is a natural correspondence between `alphaTags` and `FlowTags` we used previously to track packet modifications and dynamic middlebox actions [29]. For instance, if a DPF modifies headers, then the BDU to carries the context so that the true *origins* of the packet is not lost, and can be used to check if the relevant policy at some downstream DPF is implemented correctly; e.g., if a NAT modifies the srcIP, then the downstream firewall may not be able to apply its rules consistently. In the next section we explain how these ideas are used to generate a concrete test trace. Note that BUZZ does not require that the actual DPFs be `FlowTags`-enabled; it merely uses these `FlowTags`-like constructs internally to model DPF operations.

5.4 Putting it together

Combining the above ideas, each DPF is thus modeled as an FSM ensemble that receives an input BDU and generates an output BDU. The output BDU encodes the relevant contextual information associated with the effect

Listing 3: Network pseudocode for Figure 1.

```
1  // Symbolic BDUs to be instantiated (see §6).
2  BDU A[20];
3  int httpObjId = httpObjIdToMonitor;
4  // Global state variables
5  bool Cache[2][100]; // 2 proxies,  100 objects
6  // Switch
7  BDU  Switch(DPFId id, BDU inBDU){
8  outBDU=lookUp(id, inBDU);
9  return outBDU;
10 }
11 // Monitoring DPF
12 BDU  Mon(DPFId id, BDU inBDU){
13 ...
14 if (isHttp(id, inBDU)){
15  updateHttpStats(id, inBDU);
16 }
17 ...
18 outBDU = inBDU;//a passive monitoring device
19 return outBDU;
20 }
21 // Proxy DPF; See Listing 2
22 BDU Proxy(DPFId id, BDU inBDU){
23 ...
24 }
25 // Network sequential processing (§3)
26 for each injected A[i]{
27   while (!DONE(A[i])){
28     Forward A[i] on current link;
29     A[i] = Next_DPF(A[i]);
30     assert((!(A[i].alphaTags[1]==CachedHttpObjId)
31                  || (!A[i].port=MonitorPort));
32   }
33 }
```

that the DPF performed on the input BDU. We assume that each DPF instance has a unique id that allows us to identify the "type" of the DPF and thus index into the relevant global state variables.

The generality of BDUs and the fact that they also capture the locations (note `networkPort` in Listing 1, which species the current network port at which the BDU is located) allow DPFs to be easily composed. Concretely, consider the network of Figure1 and see how we compose models of proxies, switches, and the monitoring device as shown in Listing 3. Lines 7–10 model the stateless switch. Function `lookUp` takes the input BDU, looks up its forwarding table, and creates a new `outADU` with its port value set based on the forwarding table. (Following prior work [35], we consider each switch DPF as a static data store lookup updating located packets.) Lines 12–20 capture the monitoring DPF.

We model the data plane as a simple loop (Line 26) following the sequential lock-step semantics from §3. In each iteration, a BDU is processed (Line 27) in two steps: (1) the BDU is forwarded to the other end of the current link, (2) the BDU is passed as an argument to the DPF connected to this end (e.g., a switch or firewall). The BDU output by the DPF is processed in the next iteration until the BDU is "DONE"; i.e., it either reaches its destination or gets dropped by a DPF. The role of `assert` will become clear in the next section when we use symbolic execution to exercise a specific policy behavior.

6 Test Traffic Generation

Given the data plane model (as described in the last section) and a policy $(\mathit{TraceSpec}; \mathit{TraceSem})$, our next goal is to generate a test plan that explores the states of the data plane to induce the intended *TraceSem*. We break this into two logical steps: (1) generating a plan at the granularity of BDUs using symbolic execution, and (2) translate it into a concrete test manifest (test scripts that create test traffic). The key practical requirement is that we want this to be fast (seconds) for interactiveness.

6.1 Symbolic Execution using BDUs

While BDUs address the challenges of modeling data planes, they do not address *state space explosion* due to composition of DPF models. To see why, suppose a BDU (e.g., a `TCP SYN+ACK`) sequentially traverses DPF_1 (a proxy) and DPF_2 (firewall). Suppose DPF_1 and DPF_2 can reach K_1 and K_2 possible states w.r.t. this BDU, respectively (e.g., the proxy waiting for a `TCP SYN+ACK` from a web server, and the firewall watching for unsolicited connection requests). The composition of the DPFs can reach $K_1 \times K_2$ states w.r.t. this BDU. This combinatorial growth with the number of DPFs and possible BDUs makes it difficult to find a test trace.

We tried several approaches to tackle state-space explosion using AI planning, model checking, and custom search techniques. However, these techniques did not scale beyond networks with 5-10 DPFs.

To address this scalability challenge, we turn to *symbolic execution*, which is a well known approach in formal verification to address state-space explosion [21]. At a high-level, a symbolic execution engine explores possible behaviors of a given program by considering different values of *symbolic variables* [23]. One well-known concern is that symbolic execution can sacrifice coverage. In our specific application context, this tradeoff to enable interactive testing is worthwhile. First, administrators may already have very specific testing goals in mind. Second, configuration problems affecting many users will naturally manifest even with one test trace. Finally, if we have a fast solution, we can run several tests changing the values to improve coverage.

Thus, to generate a high-level plan, we use symbolic execution in BUZZ at the granularity of BDUs to produce a Π_{BDU} or a sequence of BDUs. To this end, we define test BDUs as symbolic variables. The symbolic execution engine assigns values to these test BDUs such that "interesting states" of the data plane representing *TraceSem* are triggered. Specifically, given the policy $(\mathit{TraceSpec}; \mathit{TraceSem})$, we use symbolic execution as follows. First, we constrain the symbolic BDUs to satisfy the *TraceSpec* condition. Second, we introduce the negation of *TraceSem* or $\neg$*TraceSem* as an *assertion* in the model code. In practice, the $\neg$*TraceSem* assertion will be

Listing 4: 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     assert((!L_IPS_Alarm[A[i].srcIP]) ||
6             (!H_IPS_Alarm[A[i].srcIP]));
```

expressed in terms of BDU fields (e.g., `networkPort`, `alphaTags`) and the global state variables. Then, we let the symbolic execution engine find an assignment to the symbolic BDU variables that causes this assertion to be violated. Because we use the negation in the assertion, the end result is that we will get a BDU-level trace that induces the effects in *TraceSem*.

To see this more concretely, we revisit the example from Figure 1 in Listing 3. Suppose we want a test plan to observe cached responses from the proxy to Dept. Lines 30-31 shows the assertion so that symbolic execution will instantiate a trace of BDUs that causes a cached response to be returned by the proxy (where being a cached response here is encoded in `alphaTags[1]` part of BDUs) to arrive at the monitor's incoming port. For instance, suppose currently there is TCP connection between a host in the Dept. the symbolic execution engine might give us a test plan with 5 BDUs: three BDUs between a host in the Dept. and the proxy to establish a TCP connection (the 3-way handshake), a fourth BDU has `httpGetObj = httpObjId` from the host to the proxy (which will results in a cache miss at the proxy and triggers fetching the object by the proxy from the remote server), followed by another BDU with the field `httpGetObj` set to `httpObjId` to induce a cached response. Note we set the number of symbolic BDUs conservatively high (i.e., 20 in this example) without facing any slowdown. The reason behind this is that as soon as the assertion is violated (e.g., after 5 BDUs), test planning terminates and the extra symbolic BDUs do not affect the symbolic execution.

Listing 4 shows another example of an assertion ensuring that an alarm is triggered at both L-IPS and H-IPS of Figure 3. The assertion in Lines 5-6 creates a trace of BDUs capturing a sequence of connection attempts that triggers both L-IPS and H-IPS to raise alarms.

6.2 Optimizing Symbolic Execution

While symbolic execution is orders of magnitude faster than other options, it does not provide the speed needed for interactive testing. Even after a broad sweep of configuration parameters and command line arguments to customize KLEE, it took several hours even for a small topology (§9). To make it scale to larger topologies, we implemented a suite of domain-specific optimizations:

- *Minimizing symbolic variables:* Making the entire BDU symbolic will force KLEE to find values for every field. To avoid this, we use policy-specific insights to have a small subset of symbolic fields; e.g., when are testing the stateful firewall without a proxy, we can set the HTTP-relevant fields to concrete values.
- *Scoping values of symbolic variables:* The *TraceSpec* already scopes the range of values each BDU can take. We can further narrow this range and still find good test traces. To this end, we use protocol-specific insights to assign concrete values to as many BDU fields as possible. For example, we set a client's TCP port number to a unique value (as opposed to making the `srcPort` field symbolic). This value is only used in the model for test planning and the actual client TCP port is chosen by the host at run time (§6.3).
- *Other optimizations:* We applied other optimization to further speed up the traffic planning process. This includes "memorizing" BDU traces to trigger previously explored states as well as utilizing overlaps between *TraceSem* components of different policies.

6.3 Generating Concrete Test Traffic

We cannot directly inject BDUs into a physical data plane since these are abstract entities; we need to test the data plane using a set of real IP-layer packets. To this end, we design a custom translation layer that takes as input a high-level test plan (i.e., sequence of BDUs) and generates a concrete *test manifest* that we can feed into the BUZZ traffic injectors. This insight in designing this translation layer is two fold. First, we need to consider the protocol semantics of traffic to which the intended policies apply (we have considered IP, TCP, UDP, and HTTP so far). For example, a sequence of three BDUs that correspond to `TCP SYN` from host *A* to server *B*, `TCP SYN+ACK` from *B* to *A*, and `TCP ACK` from *A* to *B* collectively indicate a TCP connection establishment. Second, we cannot create the concrete test traffic at a packet level (e.g., as opposed to ATPG [55]). This is because many values are determined by the OS at run time; e.g., we cannot predict TCP sequence numbers. Thus, we translate the BDUs to a sequence of traffic generation functions into a *script* that will be run at a given injection point. For example, the above three TCP BDUs are translated to a function `establishTCP(A,B)` runs at host A to connect to TCP server B. We currently use 10 such traffic generation primitive functions that support IP, TCP, UDP, and HTTP. We plan to extend this to accommodate other protocols.

7 Test Monitoring and Validation

After the test traffic is injected into the data plane, the outcome should be monitored and validated. First, we need to disambiguate true policy violations from those caused by background interference. Second, we need mechanisms to help localize the misbehaving DPFs.

	Orig = *Obs*	*Orig* ≠ *Obs*
No interference or resolvable interference	Success	Fail. Repeat on *Orig* − *Obs* using MonitorAll
Unresolvable interference	Unknown; Repeat *Orig* using MonitorAll	

Table 2: Validation and test refinement workflow.

While a full solution to fault diagnosis and localization is outside the scope of this paper, we discuss the practical heuristics we implement.

Monitoring: Intuitively, if we can monitor the status of the network in conjunction with the test injection, we can check if any of the background or non-test traffic can potentially induce false policy violations. Rather than monitor all traffic (we refer to this as MonitorAll), we can use the intended policy to capture a smaller relevant traffic trace; e.g., if the policy is involves only traffic to/from the proxy, then we can focus on the traffic on the proxy's port. To further minimize this monitoring overhead, as an initial step we capture relevant traffic only at the switch ports that are connected to the stateful DPFs rather than collect traffic traces from all network ports. However, if this provides limited visibility and we need a follow-up trial (see below), then we revert to logging traffic at all ports for the follow-up exercise.

Validation and localization: Next, we describe our current workflow to validate if the test meets our policy intent, and (if the test fails) to help us localize the sources of failure otherwise. The workflow naturally depends on whether the test was a success/failure and whether we observed interfering traffic as shown in Table 2.

Given the specific policy we are testing and the relevant traffic logs, we determine if the network satisfies the intended behavior; e.g., do packets follow the policy-mandated paths? In the easiest case, if the observed path *Obs* matches our intended behavior *Orig* and we have no interfering traffic, this step is trivial and we declare a success. Similarly, if the two paths match, even if we have potentially interfering traffic, but our monitoring reveals that it does not directly impact the test (e.g., it was targeting other applications or servers), we declare a success.

Clearly, the more interesting case is when we have a test failure; i.e., *Obs* ≠ *Orig*. If we identify that there was no truly interfering traffic, then there was some potential source of policy violation. Then we identify the largest common path prefix between *Obs* and *Orig*; i.e., the point until which the observed and intended behavior match and to localize the source of failure, we zoom in on the "logical diff" between the paths. However, we might have some logical gaps because of our choice to only monitor the stateful DPF-connected ports; e.g., if the proxy response is not observed by the monitoring device, this can be because of a problem on any link or switch between the proxy and the monitoring device. Thus, when we run these follow up tests, we enable MonitorAll to obtain full visibility.

Finally, for the cases where there was indeed some truly interfering traffic, then we cannot have any confidence if the test failed/succeeded even if *Obs* = *Orig*. Thus, in this case the only course of action is a fall back procedure to repeat the test but with MonitorAll enabled. In this case, we use an exponential backoff to wait for the interfering flows to die.

8 Implementation

DPF models: We wrote C models for switches, ACL devices, stateful firewalls (capable of monitoring TCP connections and blocking based L3/4 semantics), NATs, L4 load balancers, proxies, passive monitoring, and simple intrusion prevention systems. In writing DPF models, we reuse common building blocks across DPFs (e.g., TCP connection sequence).

DPF Model validation: We implemented several measures to validate our DPF models. First, we use a bounded model checker, CMBC, on individual DPF models and the network model to ensure they do not contain software bugs (e.g., pointer violations, overflow). While this was time consuming, it was a one-time task. Second, we used call graphs [4, 15] to check that the model behaves as expected. Third, we compared the input-output behavior of the model with open source DPFs.

Test traffic generation and injection: We use KLEE with the optimizations discussed earlier to produce the BDU-level plan, and then translate it to test scripts that are deployed at the injection points. Test traffic packets are marked by setting a specific (otherwise unused) bit.

Traffic monitoring and validation: We currently use offline monitoring via `tcpdump` (with suitable filters); we plan to integrate more real-time solutions like NetSight [32]. We use `OpenFlow` [40] to poll/configure switch state. We use an OpenDaylight-based implementation of `FlowTags` [29] to gain better visibility into middlebox actions.

9 Evaluation

In this section we evaluate the effectiveness and performance of BUZZ and show that: (1) BUZZ successfully helps diagnose a broad spectrum of data plane problems (§9.1); (2) BUZZ's optimizations provide more than three orders of magnitude speedup and enable close-to-interactive running times even for large topologies (§9.2); and (3) BUZZ is also useful for incremental testing, testing DPF implementations, and testing reachability (§9.3). We begin with our setup and approach.

Testbed: In order to run realistic large-scale experiments with topologies of 100+ nodes, we use a testbed of 13 server-grade machines (20-core 2.8GHz servers

with 128GB RAM) connected via a combination of direct 1GbE links and a 10GbE Pica8 OpenFlow-enabled switch. On each server with KVM installed, we run each injectors and middleboxes as separate VMs. The VMs are interconnected via OpenvSwitch on each server. The middleboxes we use are iptables [3] as a NAT and a stateful firewall, Squid [12] as proxy, Snort [11] as IPS/IDS, Balance [1] as the load balancer, and PRADS [9] as a passive monitor. These were instrumented with `FlowTags` to handle dynamic middlebox actions [29].

Topologies and policies: In addition to the example scenarios from §2, we use 8 recent (>2010) topologies from the Internet Topology Zoo [13]. We use these as switch-level topologies and extend them with middleboxes and use various service chaining policies; e.g., a policy chain of length 3 made of a NAT, IPS, and load balancer.

Background traffic: We generate background traffic intended to *interfere* with the specific behaviors we want to test; i.e., it induces state transitions that can affect our tests. For example, in Figure 3, we use background TCP port scanning traffic as it affects the suspicious connection count per host at the L-IPS. Given the intended policies and data plane model, we modify our trace generation tool for this purpose.

9.1 Validation

Finding hidden errors: We validate the effectiveness of BUZZ using a *red team–blue team* exercise. Here, the blue team (Student 1) has a pre-defined set of policies for each network; i.e., these are the expected behaviors and there is no "overfitting" or additional instrumentation necessary for each run. Then, the red team (Student 2) picks one of the intended behaviors (at random) and creates a failure mode that causes the network to violate this policy; e.g., misconfiguring the L-IPS count threshold or disabling some control module. The blue team used BUZZ to (a) identify that a violation occurred and (b) localize the source of the policy violation. We also repeated these experiments reversing the student roles; but do not show these results for brevity.

Table 3 highlights the results for a subset of these scenarios and also shows the specific traces that blue-team used. Three of the scenarios use the motivating examples from §2. In the last scenario (Conn. limit.), 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. In all scenarios the blue-team successfully localized the failure (i.e., which DPF, switch, or link is the root cause) within 10 seconds. Note that these bugs could not be exposed with existing debugging tools such as ATPG [55], ping, or traceroute.[8]

[8]They can detect obvious failure modes such a link/switch being down but cannot capture subtle bugs w.r.t. stateful/contextual behaviors.

"Red Team" scenario	BUZZ test trace that revealed the failure to "Blue Team"
Proxy/Mon (Fig. 1); S_1-S_2 link is down	Non-cached rqst from inside the Dept, followed by request for the same object from by another source host in the Dept
Proxy/Mon (Fig. 1); The port of S_1 (Pica8) connected to proxy is (mis)configured to not support OpenFlow	HTTP rqst from Dept
Cascaded NATs (Fig. 2); `FlowTags` controller shut down	H_1 attempts to access to the server
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
Conn. limit.; Login counter resets	H_1 makes 3 continuous log in attempts with a wrong password
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

Table 3: Some example red-blue team scenarios.

Finding real bugs: BUZZ also helped us identify an actual bug in our `FlowTags` re-implementation in OpenDaylight.[9] Essentially, the controller code in charge of decoding `FlowTags` (e.g., to distinguish sources hidden behind a NAT or proxy) was assigning the same tag value to traffic from different sources. By using test traffic in network of Figure 1, we observed that proxy hit replies bypass the monitoring device, which is a violation of the intended behaviors. BUZZ validation also localized the policy violation (i.e., at the proxy). It also provided the traffic trace indicating that the tag values of cache miss and hit cases are identical, that gave us a hint to focus on the proxy tag assignment code of `FlowTags` controller.

9.2 Performance and Scalability

One of our goals is that operators should be able to use BUZZ in a reasonably interactive fashion; i.e., the time for an end-to-end test should be a few seconds or less.

Test traffic generation: We measure the time for BUZZ to generate a test trace across different topologies and service chain sizes. We evaluate the utility of the optimizations we introduced in §6.

Figure 6 shows the average time to generate test traffic for a given intended behavior for a fixed logical policy chain of length 3 (composed of a NAT, a firewall, and a proxy) across different topologies. The smallest topology has one instance of this logical policy chain (with 8 individual policies), and we increase the number of instances in other topologies linearly with the number of switches; i.e., number of policies and hence tests grows

[9]Our original implementation was in `POX`; the bug arose during the (non-trivial) process of porting it to OpenDaylight; OpenDaylight is a significant codebase with a lot of complexity.

linearly (this is done for other experiments of this subsection too). Since all values are close to the average, we do not show error bars. To put our numbers in perspective, using KLEE without any of our optimizations even on a network of six switches and one policy chain instance with three middleboxes took over *19 hours* to complete. The graph also shows the (projected) value of the unoptimized setup using this baseline result.

The first optimization (minimizing the number of symbolic variables) dropped the latency of the baseline example to less than 12 seconds (more than three orders of magnitude reduction). Constraining the values of symbolic variables yields another $> 9\times$ latency reduction. Finally, other minor optimizations reduce the time for test traffic generation by about 6%.

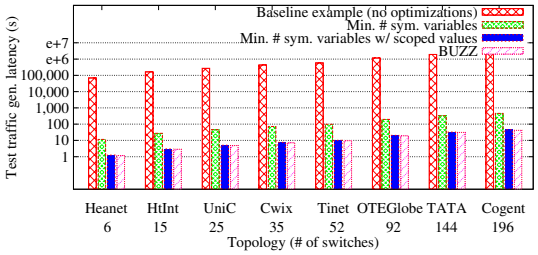


Figure 6: Traffic generation latency with a fixed logical policy chain of length 3 across different topologies.

Figure 7 shows the average traffic generation latency with a fixed topology with 52 switches (Tinet) and variable policy chain length (all values were close to the average in different runs). The effect of optimizations are consistent with Figure 6.

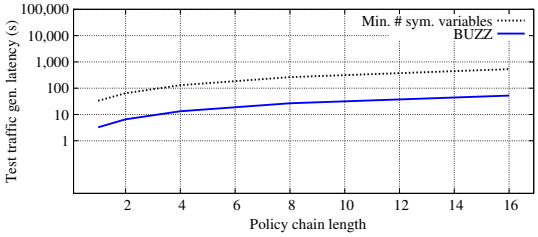


Figure 7: Test generation times for a fixed topology (Tinet + 53 switches) and variable policy chain length.

Recall that test generation in BUZZ has two logical stages: (1) high-level test planning and (2) concrete test manifest generation. In general, we find that the test manifest generation times are between 4–6% of the time to generate the test plan and are largely independent of topology size and policy chain length (not shown).

Overall these results confirm that with our optimizations, BUZZ is practical and generates tests for even large topologies with 100+ nodes in less than 30 seconds.

Monitoring overhead: Across different topologies using the monitoring strategy of monitoring only stateful DPF ports outperforms MonitorAll of §7 by at least %30 for the smallest topology and over $\times 2$ for the largest topology. The total size of the monitoring logs did not exceed 1% of the total traffic volume across different topologies. Furthermore, the fraction of the test traffic to the total network traffic (using background traffic generated based on the gravity model traffic matrix) is extremely small ($<$0.01%). This is not surprising because the duration of test is very short (less than 10 seconds) and the test sequences we generate are targeted to trigger certain network conditions (as opposed to large data transfers).

Topo.(# of switches)	Heanet (6)	UniC (25)	Cwix (35)	OTEGlobe (92)	Cogent (196)
Time (s)	1.7	10.4	15.2	28.0	58.8

Table 4: Validation times for different topologies.

Validation: The latency of validation is composed of two components: analysis of monitoring logs and follow-up test traffic generation (if necessary). Table 4 shows the average validation latency across different topologies. The policy chain in this case is fixed (of length of three), and one randomly selected data plane element fails. In another experiment (not shown) we varied the length of the logical policy chain on fixed topologies. The validation latency in this case increases linearly with the length of the logical policy chain. We also measured the contribution of each of the validation stages (analysis of monitoring and generating to the validation latency). Across different topologies and policy chain lengths, generating follow up test takes between between 26% to 29% of the total validation latency. In the largest topology (Cogent) with 196 switches and 33 stateful DPFs, validation and follow up test took ≈42 and ≈17 seconds, respectively.

9.3 Other use cases

Testing `Flowtags`-enhanced DPFs: One of the original motivations for BUZZ was that existing tools were inadequate to test our `FlowTags` implementation [29]. To this end, we define intended behaviors for single DPFs and exhaustively test externally observable middlebox actions; e.g., the tagging behavior of the IPS to ensure it tags according to the intended alarm thresholds.

Incremental Testing: : One natural question is if we can incrementally test the network; e.g., when policies change. While a full discussion is outside the scope of this work, we have early evidence to show that BUZZ is amenable to incremental testing. We update the set of policies that are affected by the policy change rather than rerun the full suite of behaviors. In general, incremental testing requires time proportional to the fraction of affected policies (not shown).

Loops and reachability: While reachability checking in stateless data planes is a "solved" problem [35], it is unclear if this is true for stateful data planes. reach-

ability properties via new types of assertions. For instance, to detect loops we add assertions of the form: `assert(seen[ADU.id][port]<K)`, where `ADU` is symbolic BDU, `port` is a switch port, and `K` reflects a simplified definition of a loop that the same BDU is observed at the same port $\geq$ `K` times. Similarly, to check if some traffic can reach `PortB` from `PortA` in the network, we initialize a BDU with the port field to be `PortA` and use an assertion of the form `assert(BDU.port != PortB)`. Using this technique we were able to detect synthetically induced switch forwarding loops (not shown).

10 Related Work

Network verification: There is a rich literature in static reachability checking; i.e., a set of Can-A-talk-to-B properties [27, 53, 54]. Recent work provides a geometric header space abstraction (e.g., HSA [35]) and extends this for real-time checking [34]. Other work uses SAT solvers for checking reachability [38]. At a high level, these focus on L2/L3 reachability (e.g., black holes, loops) and do not capture networks with middleboxes. NICE combines model checking and symbolic execution to find bugs in control plane software [24]. BUZZ is complementary in that it generates test cases for data plane behaviors. Similarly, SOFT generates tests to check switch implementations against a specification [37]. Again, this cannot be extended to middleboxes.

Test packet generation: The work closest in spirit to BUZZ is ATPG [55]. ATPG builds on HSA to generate test packets that exercise the intended reachability properties. Unfortunately, ATPG cannot be applied to our scenarios on two fronts. First, middlebox behaviors are not stateless "transfer functions" which is critical for the scalability of ATPG. Second, the behaviors we want to test require us to look beyond single-packet test cases.

Programming languages: Recent work attempts to formalize network semantics to generate "correct-by-construction" programs [17,19,31]. However, to the best of our knowledge, these do not currently capture *stateful* data planes and *context-dependent* behaviors. Furthermore, many of the DPFs we consider may actually be black boxes and thus active testing may be our only option to check if the network behaves as intended.

Network debugging: There is a rich literature for fault localization in networks and systems (e.g., [30, 43, 47, 48]). These algorithms can be used in the inference engine of BUZZ. Since this is not the primary focus of our work, we used simpler heuristics in §7.

Modeling middleboxes: Joseph and Stoica formalized middlebox forwarding behaviors but do not model stateful behaviors [33]. In terms of modeling stateful behaviors, the only work we are aware of are FlowTest [28], Symnet [51], and concurrent work by Panda et al [44]. While our preliminary work on FlowTest highlighted the challenges we addressed here, there were three fundamental shortcomings: (1) the AI planning techniques do not scale; (2) the high-level DPF models generated there are not composable as discussed in §5; and (3) the approach is inflexible as it tightly couples the data plane model and intended policies. Symnet [51] writes high-level middlebox models in Haskell to capture flow affinities in NATs/firewalls; we do not have details on their models or verification procedures. Panda et al also independently model the stateful behaviors, but their work is different both w.r.t goals (reachability and isolation) and solution techniques (verification and making model checking tractable).

Simulation and shadow configurations: Simulation [6, 7], emulation [2, 5], and shadow configurations [16] are the common techniques used today to model/test networks. BUZZ is orthogonal in that it focuses on *generating test scenarios*. While our current focus is on active testing, BUZZ's applies to these platforms as well. We also posit that there are other avenues where our techniques can be used to validate these efforts.

11 Conclusions

In this work, we presented the design and implementation of BUZZ, a practical testing framework to test complex and contextual policies in realistic network settings with stateful middleboxes. We addressed key challenges in developing tractable-yet-useful models of middleboxes and in making symbolic execution tractable in this domain. We believe that the ideas that were critical to make the BUZZ vision practical—BDUs as the unit of input-output operations, modeling DPFs as FSM ensembles, use of symbolic execution at BDU granularity, and the optimizations we implement in KLEE—will be more broadly applicable for other network verification tasks. By demonstrating the scalabilty and viability of BUZZ, this work takes the "CAD for networks" vision one giant step closer to reality.

References

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

[2] Emulab. `http://www.emulab.net/`.

[3] iptables. `http://www.netfilter.org/projects/iptables/`.

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

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

[6] ns-2. http://www.isi.edu/nsnam/ns/.

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

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

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

[10] PyResonance. https://github.com/Resonance-SDN/pyresonance/wiki.

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

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

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

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

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

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

[17] 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.

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

[19] T. Ball et al. VeriCon: Towards verifying controller programs in software-defined networks. In *Proc. PLDI*, 2014.

[20] A. Biggadike, D. Ferullo, G. Wilson, and A. Perrig. Analysis and topology-based traversal of cascaded large scale nats. In *Proc. SIGCOMM Asia Workshop*, 2005.

[21] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10ˆ20 States and Beyond. *Inf. Comput.*, 98(2), 1992.

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

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

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

[25] X. Chen, Z. M. Mao, and J. Van Der Merwe. Shadownet: A platform for rapid and safe network evolution. In *Proc. USENIX ATC*, 2009.

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

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

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

[29] 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.

[30] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proc. NSDI*, 2007.

[31] N. Foster et al. Frenetic: A network programming language. *SIGPLAN Not.*, 46(9), Sept. 2011.

[32] N. Handigol et al. I know what your packet did last hop: Using packet histories to troubleshoot network. In *Proc. NSDI*, 2014.

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

[34] 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.

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

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

[37] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A soft way for openflow switch interoperability testing. In *Proc. CoNEXT*, 2012.

[38] 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.

[39] N. McKeown. Mind the Gap: SIGCOMM'12 Keynote. http://www.youtube.com/watch?v=Ho239zpKMwQ.

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

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

[42] A. Müller, F. Wohlfart, and G. Carle. Analysis and topology-based traversal of cascaded large scale nats. In *Proc. HotMiddlebox*, 2013.

[43] R. N. Mysore, R. Mahajan, A. Vahdat, and G. Varghese. Gestalt: Fast, Unied Fault Localization for Networked Systems. In *Proc. USENIX ATC*, 2014.

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

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

[46] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.

[47] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proc. NSDI*, 2006.

[48] C. Scott et al. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *Proc. SIGCOMM*.

[49] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zaris, and S. Shenker. Troubleshooting blackbox sdn control software with minimal causal sequences. In *Proc. SIGCOMM*, 2014.

[50] J. Sherry et al. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. SIGCOMM*, 2012.

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

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

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

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

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