Testing Stateful and Dynamic Data Planes with FlowTest

Seyed K. Fayaz, Vyas Sekar
Carnegie Mellon University

ABSTRACT

Many recent efforts have leveraged Software-Defined Networking (SDN) capabilities to enable new and more efficient ways of testing the correctness of a network's forwarding behaviors. However, realistic network settings induce two additional sources of complexity that fall outside the scope of existing SDN testing frameworks: (1) complex nature of real-world data planes (e.g., stateful firewalls, dynamic behaviors of proxy caches), and (2) complexity of intended network policies (e.g., service chaining). In this paper, we outline FlowTest, a high-level vision for testing such stateful and dynamic network policies. FlowTest systematically explores the *state space* of the network data plane to verify its behavior w.r.t. policy goals. We show the early promise of our approach and discuss open challenges in realizing this vision in practice.

Categories and Subject Descriptors: C.2.3 [Computer-Communication Networks]: Network Management

Keywords: Network test, stateful data plane, policy enforcement

1. INTRODUCTION

Software-Defined Networking (SDN) has been instrumental in enabling new *testing and verification* capabilities. Specifically, we have seen many frameworks to ensure that the network meets specific reachability properties (e.g., no black holes, no permanent loops, and no access control violations). These include work on *static and runtime checking* of network configurations (e.g., HSA [17], VeriFlow [18]), *programming languages* for control applications (e.g., Frenetic [11]), *test packet generation* tools (e.g., ATPG [29]), and tools to *test control programs* (e.g., NICE [8], VeriCon [6]).

While the aforementioned efforts have taken significant strides, they still fall short of capturing *realistic* network settings on two key dimensions. (We elaborate on these in §2.)

- **Complex data plane elements:** Studies have shown that there are a range of advanced data plane functions (DPF)[1] including firewalls, load balancers, NATs, proxies, intrusion detection and prevention systems, and application-level gateways [15, 25, 26]. Unlike routers and switches, such DPFs are *stateful*. This stateful behavior manifests in different forms: (1) DPFs maintain *connection- and application-level* context; e.g., a stateful firewall allows responses on connections established from inside a secure network; (2) DPFs track *state across connections* depending on the history of observed traffic; e.g., a rate limiter may count the number of active sessions, or an IDS may keep track of recent failed connections [16], or a proxy may store responses in its cache.
- **Complex policy requirements:** While reachability is necessary, it does not capture the full spectrum of requirements that network administrators may want to verify. Administrators may want to ensure that *service chaining* policies are implemented correctly [22]. Other kinds of policies may focus on *dynamic dataflow* properties in that the intended policy depends on the specific "states" of the network. For example, we may want to ensure that traffic from suspicious hosts be routed through deep packet inspection filters [4].

[1]We use the term DPF to encompass both hardware middleboxes and virtualized instances.

HotSDN'14, August 22, 2014, Chicago, IL, USA.
Copyright 2014 ACM 978-1-4503-2989-7/14/08 ...$15.00.
http://dx.doi.org/10.1145/2620728.2620751.

Such complex DPF behaviors, unfortunately, fall outside the scope of existing test and verification tools in the SDN literature. For instance, while verification tools can account for simple "waypointing" policies, they cannot capture hidden actions of non-L2/3 data plane elements; e.g., a proxy sending a cached response or a NAT rewriting headers. Our overarching vision in this work is to bridge this gap between the promise of "CAD for networks" vision [19] and the capabilities offered by today's SDN tools (§2).

In this paper, we focus on sketching the conceptual foundations for a practical *data plane testing* framework called *FlowTest* to systematically test such stateful behaviors and policy requirements. We believe data plane testing is a more practical approach compared to configuration testing or static verification on three fronts. First, stateful behaviors of proprietary DPFs may be *hidden* and *non-deterministic*. For instance, we may not know the current set of cached objects in a proprietary proxy. Similarly, with load balancers or NATs, we may not know the internal mappings between the incoming/outgoing connections. Second, these behaviors may have subtle temporal effects; e.g., the "counting" behavior of a rate limiter or the "eviction" behavior of a cache may depend on internal timers. Third, it might be difficult to precisely model the complex control logic that administrators want atop DPFs; e.g., distributed middlebox load balancing [21] or elastic middlebox scaling or migration [12, 13, 23], or how they react to network dynamics (e.g., link congestion or server load).

In conjunction, these stateful and dynamic effects require us to think beyond individual packets in designing test traffic [29]. Rather, what we need is a *trace* or a specifically interleaved sequence of packets that logically *triggers* a chain of specific state transitions in the data plane. To this end, FlowTest models DPFs as *state machines* where each state represents a state of the DPF (e.g., per-connection state for each session or the objects in the proxy cache). Given these models, test trace generation can be formulated as a problem of identifying a sequence of transition events or a *plan*, that causes the set of network DPFs to transition from their current states to a desired *goal* state.

As an early attempt, we cast the trace generation problem within the framework of *planning* tools from the AI literature [24] and

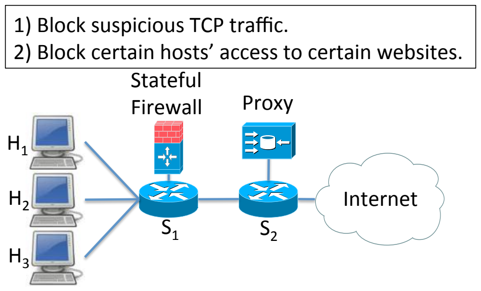

1) Block suspicious TCP traffic.
2) Block certain hosts' access to certain websites.
Stateful Firewall
Proxy
H1
H2
H3
S1
S2
Internet

Figure 1: The stateful firewall maintains a state per TCP connection. The proxy's state (i.e., cache contents) is determined by the history of observed traffic.

demonstrate the preliminary promise of using existing planning tools like GraphPlan [3] to generate test scenarios. That said, however, we do not claim that this is necessary or optimal—we choose it simply because it gives us a natural "language" to express our requirements. We are also exploring techniques from software testing including fuzzing [28] and symbolic checking [7].

In the rest of the paper, we begin by motivating the problem of stateful data plane testing in §2. We give an overview of FlowTest in §3 and show the preliminary promise in leveraging AI planning tools to help generate a test traffic sequence in §4 and §5. We conclude with a discussion of open questions and challenges in §6. We discuss related work inline throughout the paper.

2. MOTIVATION

In this section, we use simple examples to highlight the challenges of verifying complex policies with stateful DPFs. Even though these examples are highly simplified, they are useful to highlight key aspects that any candidate solution in this problem space would likely face: (1) the need to use testing rather than static verification, (2) the need to look beyond generating individual test packets, and (3) the need to capture the state semantics of DPFs. Table 1 summarizes the key aspects of these examples.

2.1 Example Scenarios

Example 1: In Figure 1, our policy goal is to (1) block unsolicited connections from the Internet, and (2) block a subset of users from accessing specific sites. The firewall and the proxy are stateful, operating above L2/3. The stateful firewall operates based on a model of TCP connections and maintains a state per connection (e.g., SYN observed, connection established, or invalid connection attempt). The proxy operates at the session level and responds to HTTP requests directly if it has the object cached; otherwise, it retrieves the object from the remote server. Note that the proxy's state (i.e., the current cache contents) and firewall's state (i.e., current connections status) at any given time might be hidden if these are proprietary DPFs (either hardware or virtual appliances), and depends on the recent history. Even in case of open source DPFs, these states may not be readily visible given the complexity of the DPF's internal logic.

In this network, we may want to test that the firewall is correctly implementing the stateful semantics; i.e., it only allows reverse traffic for previously established connections. Similarly, we want to systematically explore different possible behaviors; e.g., do cached responses violate the "chaining" policies?

Example 2: In Figure 2, we want to use the "light" IPS (L-IPS) to flag hosts as suspicious and then subject traffic from the suspicious hosts to deeper inspection at the "heavy" IPS (H-IPS). For

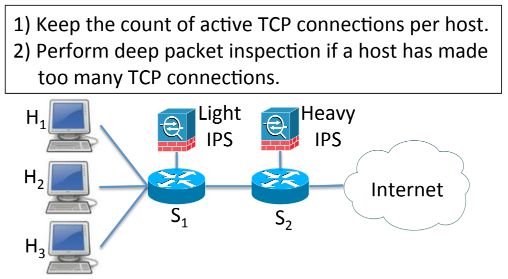

1) Keep the count of active TCP connections per host.
2) Perform deep packet inspection if a host has made too many TCP connections.
Light IPS
Heavy IPS
H1
H2
H3
S1
S2
Internet

Figure 2: The test scheme needs to be cognizant of both IPSes states and their event-driven logical connection.

		Example1	Example2
Data plane	Stateful	Firewall only allows valid connections	Light IPS counts number of connections
	Hidden	Proxy cache state	Counters/timers
Policy	Service chaining	Per-user class service chains	Only suspicious hosts routed through heavy IPS
	Dynamic	Proxy cached responses	Triggered action when connection count exceeds threshold

Table 1: Summary of the motivating examples.

instance, L-IPS may check if a host has k consecutive failed connections [16], and H-IPS should apply known botnet payload signatures to such flagged hosts.

Testing whether the above network configuration behaves as expected will need some way to explore different data plane states: (1) a host having less than k TCP connections established; (2) k TCP connections from this failed and a botnet payload was observed; and (3) k TCP connections failed but no botnet payloads were observed. Exploring these different states requires taking into account the state semantics and configurations of L- and H-IPS. Again, the counting state maintained by L-IPS may be hidden and depend on its internal timeout logic.

2.2 Current Solutions

At a high-level, prior work in the SDN testing/verification space has largely focused on stateless reachability properties, and they do not capture such stateful DPFs or dynamic data flow behaviors.[2]

Static verification: These methods (e.g., HSA [17]) take the data plane configuration and infer whether certain properties hold. However, they cannot be used to capture runtime data plane issues; e.g., they cannot capture the hidden and dynamic states of DPFs.

Test packet generation: ATPG [29] generates test packets to efficiently explore data plane reachability. However, it cannot be used to test stateful data planes; e.g., it does not capture the semantics of TCP sessions or track history of previous packets to trigger cache evictions.

Controller tools: NICE [8] and VeriCon [6] model SDN control applications as state machines and use model checking to find hidden logic bugs; they do not capture data plane effects. Other language frameworks (e.g., Pyretic [20]) simplify SDN programming, but do not capture semantics of stateful DPFs.

Middlebox orchestration: Prior work provides mechanisms to ensure correct forwarding even in the presence of hidden DPF ac-

[2]To be fair, they do explicitly mention that such stateful DPFs are outside their intended scope.

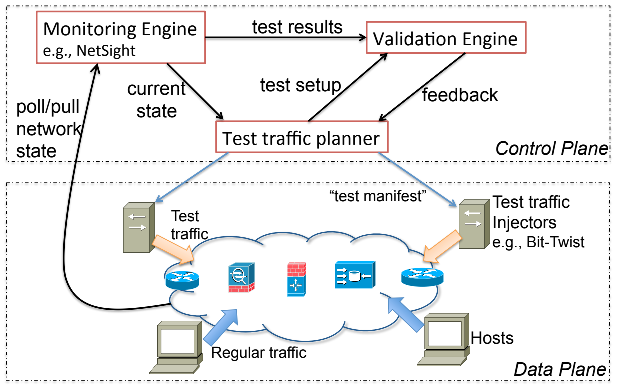


Figure 3: High-level architecture of FlowTest.

tions. FlowTags requires new APIs for DPFs to expose information to SDN controllers [10]. Stratos explicitly replicates virtual middleboxes to avoid such issues [12]. While these provide concrete realizations, they cannot guarantee correctness; e.g., in the presence of failures. In fact, this paper is motivated by the inability to systematically test our implementations in this previous work.

3. FlowTest OVERVIEW

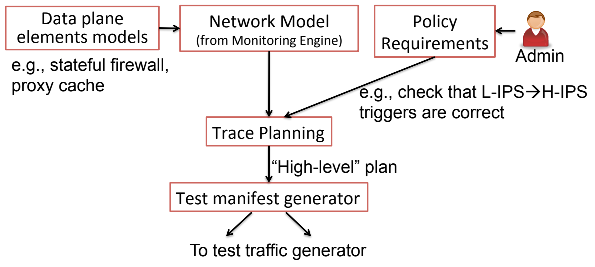
In this section, we begin with the overall vision of the *FlowTest* framework to address the challenges in tackling stateful and dynamic DPFs and policy requirements. Figure 3 shows a high-level overview of FlowTest with three logical components:

1. *Test traffic planner*, which generates a test traffic plan or *manifest* and coordinates the actions of the *injectors* to generate traffic traces that test desired properties.
2. *Injectors* are regular hosts or servers running test traffic generators or trace injection software and run the commands issued by the planner.
3. The *monitoring and validation engines* passively monitor the status of the SDN controller and the data plane. This serves two purposes. First, the current state can inform the actions of the planner; e.g., which are the reachable prefixes. Second, monitoring reports help us validate if a test succeeded or help diagnose why a test may have failed; e.g., is it a legitimate failure or caused by interference from background traffic? (See §6.)

Our focus in this paper is on designing the *test traffic planner*, and we plan to integrate tools from prior work for the other components. For example, we can use existing injection tools (e.g., Bit-Twist [1] or pytbull [5]) or request generators (e.g., Harpoon [27]). Similarly, for the monitoring engine, we can use a combination of controller checkpoints and data plane logging mechanisms enabled by SDN [14].

Figure 4 shows the modules within the test traffic planner component. The output of the planner is a *test manifest* for the various injectors, indicating the sequence of test traces to inject, the time to inject, and the set of locations at which to inject these packets. The main inputs to the test traffic planner are:

- *Models of data plane elements:* First, we need abstract models to capture different DPFs. While stateless DPFs (i.e., switches and routers) can be modeled using *transfer functions* [17], we need new models for stateful DPFs such as proxies and stateful firewalls.
- *Network model:* We need to model the network topology (i.e., how the DPFs are connected) and the forwarding strategy. This is essential to test service chaining policies and dynamic data flows.


Data plane elements models
e.g., stateful firewall, proxy cache
Network Model
(from Monitoring Engine)
Policy Requirements
Admin
e.g., check that L-IPS→H-IPS triggers are correct
Trace Planning
"High-level" plan
Test manifest generator
To test traffic generator
Figure 4: Design of the test traffic planner in FlowTest.

- *Policy requirements:* In addition to traditional network reachability properties (e.g., loop-freeness or black holes), the power of advanced data planes coupled with new SDN control planes will likely lead to more complex policies. We already saw some examples with the service chaining policy in Figure 1 and the dynamic data flow in Figure 2. We may also want to verify other consistency properties; e.g., whether a NAT maps all packets in a flow consistently; whether a stateful firewall blocks reverse-direction traffic for unestablished connections; and whether a rate limiter honors the configured thresholds.

Given these inputs, we envision a *trace planner algorithm* that generates a "high-level" test plan; e.g., *"inject flow f into the network from host/server I at time t"*. Essentially, this involves some mechanism to systematically explore different states of the data plane and test its behavior in these states. For instance, in the example of Figure 2, to ensure that L-IPS triggers a suspicious flag, we need a host to generate three consecutive failed connections.

The test manifest generator translates the output of the planner into a concrete traffic trace, which can be injected into the network. We assume that the test manifest has a repository of test traffic traces that it can choose from to generate this concrete trace. While one could conceptually unify the functions of test planning and test manifest generation, we take a pragmatic decision to decouple them to enable a more modular design so that we can independently incorporate better versions of the individual algorithms for test planning and manifest generation as they become available.

4. GENERATING TEST PLANS

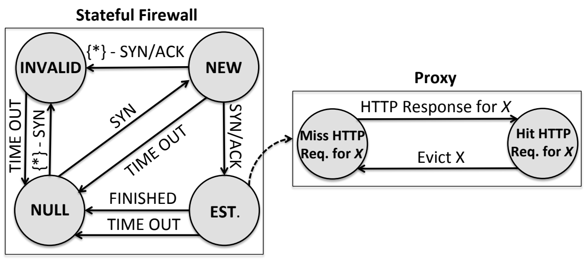
In this section, we discuss our initial approach in designing the test traffic planner. We begin by discussing how we model DPFs as state machines.[3] Then, we show how we can compose DPFs to build a model of the entire network. Finally, we show how we can cast test trace generation in the language of *planning* tools from the AI literature.

4.1 DPFs as State Machines

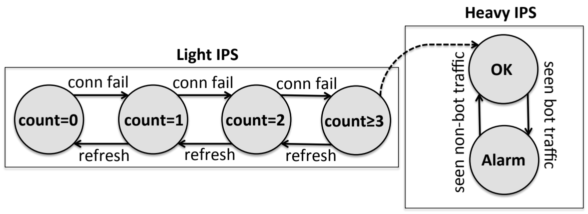
As illustrative examples, Figure 5 highlights how the different DPFs from the previous examples can be naturally modeled as state machines.

1. *Stateful firewall:* Each observed packet belongs to a connection that is in one of the four shown states: `NULL`, `NEW`, `ESTABLISHED`, or `INVALID`. Each connection starts in the `NULL` state. Receiving a packet that corresponds to an outgoing edge of the current state takes the connection to the next state.
2. *Proxy:* The state of a proxy is expressed w.r.t. the HTTP object X (e.g., an image on a web page). The behavior of the proxy

[3]While the set of states could be infinite w.r.t. future traffic patterns (e.g., all HTTP objects), at any given instant, there are a finite set of states.



(a) Example in Figure 1.



(b) Example in Figure 2.

Figure 5: Modeling DPFs as state machines and service chaining or composition as special conditional transitions between individual state machines.

will differ depending on whether *X* is in the cache at a given instant.

3. *Light IPS:* The light IPS (L-IPS) maintains a counter for each source host, tracking the number of recent failed connections. It increases the counter when failed connections recur and periodically decrements the counter to avoid false positives.
4. *Heavy IPS:* The heavy IPS (H-IPS) just maintains two states tracking if a host is sending malicious traffic or not.

Conceptually, there is a separate state machine per some atomic unit of traffic relevant for each DPF. For example, each observed connection in the stateful firewall will have the state machine in the left-hand side of Figure 5a, as each connection can be in a different state of connection establishment.

We currently assume that the DPF vendor or domain experts provide these models as inputs to FlowTest. Automatically synthesizing models from the source code or black-box behaviors is an interesting direction for future work (see §6).

4.2 Modeling the Network

Having modeled individual DPFs, we need to put them together to create a network-wide model of the data plane (the "network model module" in Figure 4). Suppose our chaining policy is to ensure that DPF *A* processes the traffic before DPF *B*. Then, we embed the prerequisite of traffic being processed by *A* as a precondition to enter the state machine model of *B* (shown by dashed arrows in Figure 5). Note that not all transitions across the state machines of individual DPFs may be meaningful; e.g., a proxy may not respond unless the connection is already established.

Since some DPFs may modify packet headers, setting up forwarding entries to compose DPFs might be challenging as observed in prior work [10, 21]. Our goal in this paper is not to mandate a specific data plane realization for service chaining, and we can use a combination of existing approaches [10, 12, 21]. Rather, our goal in FlowTest is to *test* whether such implementations are correct, and the design of FlowTest is agnostic to this implementation.

We also envision including models of stateless DPFs (i.e., switches and routers) as part of this network-wide model. We can leverage existing work on reachability testing to model the forwarding behaviors [17, 29].

4.3 State-Space Exploration

Given the DPF and network state models and some "initial" network state, we can formulate our testing requirement in terms of creating a sequence of events/transitions to move the network to some intended "goal" state. For instance, to check whether H-IPS will block hosts sending three failed connections with a botnet payload, we need to ensure that the network transitions to H-IPS Alarm state.

Based on this intuition, we found that AI planning tools offer a "natural language" in which to formulate this problem [24]. That said, we do not claim that planning is an optimal or the only possible approach and we are also exploring other more traditional software testing approaches (see §6).

We make two other observations here. First, not all states may be relevant or important to explore; e.g., if the alarm count threshold of L-IPS=3, we may not care about the case of having 100 failed connections, but focus instead on values close to the threshold. Second, there might be multiple candidate plans to reach our goal state; we defer the issue of coverage to future work (see §6).

5. EARLY PROMISE

Next, we describe the initial promise of implementing the test traffic planner using an AI planning tool called GraphPlan [3].

5.1 Implementation in GraphPlan

GraphPlan provides a natural way to encode the state machines and DPF composition we saw earlier in §4. Figure 6 shows the corresponding GraphPlan code snippets for the state machines of Figures 1 and 2.

Here, we have a logical universe of possible traffic events, which are modeled as `parameters` within the GraphPlan language. In FlowTest, parameters are network packets or flows with specific attributes; e.g., a port-80 HTTP connection. Each logical state in a DPF state machine is represented as a specific *predicate*. Some predicates may be additionally associated with a given parameter; e.g., for a stateful firewall $new\langle x\rangle$ indicates that the flow $\langle x\rangle$ is logically in state new, whereas for an L-IPS the counting state is not parameterized by a specific flow $\langle x\rangle$. Each "transition" in the state machine is represented as an "operator" in the GraphPlan language. Each operator takes as input one or more parameters and gets triggered if its associated *preconditions* hold true. Once an operator has completed its execution (i.e., a state transition occurs), it has certain *effects*, wherein predicates involving x becoming true or false. As discussed in the previous section, composition of DPFs can be modeled as logical preconditions that carry over from one DPF model to the next.[4]

To make this concrete, let us walk through the L-IPS–H-IPS example in Figure 6b. For brevity, we only show a subset of the possible transitions here. Here, the states for L-IPS are of the form `count-*`. The code shows that in the state count-2, L-IPS will transition to the state count-3 if it sees a new failed connection. It marks this flow as seen using an auxiliary predicate as shown. It also triggers another predicate *caused-lips-alarm* which is used in the composition. H-IPS is much simpler and the only states are HIPS-OK and HIPS-ALARM; its precondition is that the processing is triggered only if the *caused-lips-alarm* is true and it transitions to the HIPS-Alarm state if it observes bot signatures in flow $\langle x\rangle$. Similarly, Figure 6a shows the code snippet for the FW-proxy example. While not shown here for brevity, we can also refine the

[4]We also have models to capture network-level state such as flow creation; we do not show these for brevity.

```
/*Stateful Firewall*/
OPERATOR1  GET-SYN       PARAMS(<x>)
PRECONDS:  (null <x>)
EFFECT:    (DEL fw-null <x>) (fw-new <x>)
OPERATOR2  GET-SYNACK    PARAMS(<x>)
PRECONDS:  (new <x>)
EFFECT:    (DEL fw-new <x>) (fw-est <x>)
...
```

```
/*Proxy*/
OPERATOR   GET-HTTP-RESP    PARAMS(<x>)
PRECONDS:  (miss <x>) (fw_established <x>)
EFFECT:    (DEL proxy_miss <x>) (proxy_hit <x>)
...
```

```
/*Param. Values*/
(website1)
(website2)
(website3)
(website4)
(website5)
(website6)
(website7)
...
```

```
/*Initial State*/
(PRECONDS(fw_null website1)
         (proxy_null website1))
...
```

```
/*Goal State*/
(EFFECTS (fw_est website1)
              (proxy_cached
               website1))
...
```

(a) Firewall-Proxy example (see Figures 1 and 5a).

```
/*Light IPS*/
OPERATOR1  LIPS-Process     PARAMS(<x>)
PRECONDS:  (count2)(new-flow <x>)
EFFECT:    (DEL count2) (count3) (DEL new-flow <x>
           (old-flow <x>) (counted <x>)
           (DEL LIPS-OK) (LIPS-Alarm)
           (alarmed-at-lips <x>))
...
```

```
/*Heavy IPS*/
OPERATOR1  HIPS-Process      PARAMS(<x>)
PRECONDS:  (alarmed-at-lips <x>) (bot-traffic <x>)
EFFECT:    (DEL HIPS-OK) (HIPS-Alarm)
...
```

```
/*Param. Values*/
(flow1)
(flow2)
(flow3)
(flow4)
(flow5)
(flow6)
...
```

```
/*Initial State*/
(PRECONDS(count0)
(HIPS-OK)
(LIPS-OK)
(botnet flow2))
```

```
/*Goal State*/
(EFFECTS (HIPS-Alarm))
```

(b) L-IPS-H-IPS example (see Figures 2 and 5b).

Figure 6: Code snippets for the two example scenarios of §2 in the GraphPlan language. Keywords are shown in Uppercase while predicates are italicized. Each network input (i.e., a packet or a flow) is a parameter. Each state is captured by a predicate with (possibly empty) parameters. Each "operator" models a state transition with (possibly empty) parameters. We input the initial state and the desired goal state.

```
/*row format: <ACTION, TIME STAMP>. (t1<t2<t3)*/
<attempt a failed tcp connection to server1 from H1, t1>
<attempt a failed tcp connection to server2 from H1, t2>
<attempt sending botnet traffic to server2 from H1, t3>
```

Figure 7: The test manifest to take the network of Figure 2 to the goal state that the heavy IPS generates an alarm. The three rows in the figure correspond to $flow1$, $flow3$, and $flow2$ of Figure 6b, respectively.

evict operator in the proxy model to distinguish different eviction reasons such timeout vs. cache being full.

In addition to the DPF and composition models, we also specify the *initial state* and our intended *goal state*. Figure 6b shows these initial conditions as *facts* describing our "world" where both L-IPS and H-IPS start in their default states (i.e., OK). The goal state is to make H-IPS raise an alarm.

We use GraphPlan's built-in solver to generate a *plan* that will take us from the initial state to the goal state. Figure 7 shows the solution generated by GraphPlan showing the sequence of actions to take to reach the goal state for the L-IPS–H-IPS example. Given this high-level sequence of operators, we use a *translation script* that uses a test traffic library to produce the *test manifest*. We currently manually populate this traffic library with different template traces; e.g., failed connections, web requests, etc.

5.2 Preliminary Results

Validation: To validate our test manifest, we instantiate the network topology using MiniNet. We use Snort with different configurations to realize the L-IPS and H-IPS functions, suitably extended to support FlowTags [10] so that we can dynamically reroute traffic depending on the L-IPS output. We use a custom version of POX to set up forwarding rules.

We examine two scenarios. In the first scenario, we set up the network elements correctly and run the test manifest of Figure 7. H-IPS generates an alarm as expected and blocks the malicious payload. In the second scenario, we (mis-)configure L-IPS such that it alarms upon seeing the fourth connection attempt (not the third one as mandated by the policy goal). In this case, we observe that injecting the test manifest does not trigger an alarm at H-IPS and the flow (i.e., the last row of Figure 7) passes through, violating the intended policy.

Scalability: Each of the examples of Figures 1 and 2 takes about 3 ms to be solved using GraphPlan. To evaluate the scalability, we setup synthetic larger "chains" with 6 DPFs and 100 concurrent flows, where each DPF can be in one of four states similar to the stateful firewall. In this case, the solver takes around 1 minute, which is reasonable, as we do not intend the test traffic planner component of FlowTest to be real-time. We found that the time to generate a plan grows roughly linearly as a function of the number of operators in the model (not shown).

6. DISCUSSION

Other approaches: We do not claim that the AI planning-based approach is either necessary or optimal. Our choice was pragmatic—we tried a range of DPF models using datalog, propositional logic, and first order logic, but our initial attempts could not model DPFs as compactly as the language of planning. As an alternative to planning, we also tried bounded model checkers to generate test plans. Here, we wrote DPF state machine models as simple C programs and used CBMC [2] to generate counterexamples. A counterexample indicates the values of variables that lead to the violation of a specific policy. We use these counterexamples as the input to the trace generator. We can also use symbolic execution approaches for test plan generation. That is, we can think of state graph exploration as the symbolic execution of a program that encodes this graph. The leaves of the symbolic execution tree are reachable states from the initial state and each edge indicates a state transition. The counterexample-based approach above, in effect, helps us explore the desired leaves of the tree. As ongoing work, we are evaluating the expressivity and efficiency of different approaches.

Synthesizing middlebox models: We currently model the DPF state machines manually based on our domain knowledge. A natural research direction is to automatically extract such models given the implementation of various middleboxes, either from source code (e.g., [9]) or even from blackbox traces (e.g., [30]). One interesting question is balancing model tractability vs. fidelity; e.g., we can trivially take all variables in the code as a high-fidelity model, but it may needlessly increase model complexity.

Coverage and efficiency: In general, we want to exhaustively test the policy in all possible network states. However, this may be inefficient in very large DPF state machines and large networks. Thus, we need mechanisms to make this state space exploration more efficient; e.g., identify equivalence classes of states where the behaviors might be identical. There might also be many possible test plans that essentially reach the same goal state, in which case we prefer more efficient plans. On a related note, when there are multiple goal states, we may prefer action plans with higher overlap to minimize duplicate efforts. We can also explore opportunities to improve test efficiency by explicitly forcing DPFs into particular configurations; e.g., choosing a different rate limit or failed connection threshold.

Interference and diagnosis: One natural concern is that our test traffic may interfere with background (i.e., non-test) traffic. There are two main concerns here. First, since the normal network traffic changes the state of the data plane, it may impact the correctness of our test scenario and our observations. Second, we need to make sure the test traffic does not change the network state such that adversely affects regular traffic (e.g., an IPS counter might be increased beyond the alarm threshold by test traffic). To tackle these challenge, we plan to use new middlebox APIs such as FlowTags [10] or OpenNF [13] to query the DPF state and use new SDN capabilities for ubiquitous control/data plane logging [14]. This helps in two ways. First, knowledge of the current network state can minimize interference during test generation. Second, given that interference is unavoidable, we can use these status reports to determine the root cause of a test failure.

Acknowledgments

This work was supported in part by grant number N00014-13-1-0048 from the Office of Naval Research and NSF grants CNS-1440056 and CNS-1440065.

7. REFERENCES

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

[2] CBMC. `http://www.cprover.org/cbmc/`.

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

[4] Prolexic. `http://www.prolexic.com/`.

[5] pytbull. `http://pytbull.sourceforge.net/`.

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

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

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

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855, pages 154–169. 2000.

[10] 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.

[11] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. *SIGPLAN Not.*, 46(9):279–291, Sept. 2011.

[12] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *CoRR*, abs/1305.0209, 2013.

[13] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward software-defined middlebox networking. In *Proc. HotNets-XI*, 2012.

[14] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot network. In *Proc. NSDI*, 2014.

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

[16] J. Jung, V. Paxson, A. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proc. IEEE Security and Privacy*, 2004.

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

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

[19] N. McKeown. Mind the Gap: SIGCOMM'12 Keynote. `https://www.youtube.com/watch?v=c9-K5O_qYgA`.

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

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

[22] P. Quinn et al. Network service chaining problem statement. `http://tools.ietf.org/html/draft-quinn-nsc-problem-statement-03`.

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

[24] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[25] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. NSDI*, 2012.

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

[27] J. Sommers and P. Barford. Self-configuring network traffic generation. In *Proc. IMC*, 2004.

[28] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *Proc. CCS*, 2013.

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

[30] Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos. NetCheck: Network diagnoses from blackbox traces. In *Proc. NSDI*, 2014.