Towards Consistent SDNs: A Case for Network State Fuzzing

Apoorv Shukla[1] S. Jawad Saidi[2] Stefan Schmid[3] Marco Canini[4] Thomas Zinner[1] Anja Feldmann[2,5]

[1]TU Berlin [2]MPI-Informatics [3]Faculty of Computer Science, University of Vienna [4]KAUST [5]Saarland University

***Abstract*—The conventional wisdom is that a software-defined network (SDN) operates under the premise that the logically centralized control plane has an accurate representation of the actual data plane state. Unfortunately, bugs, misconfigurations, faults or attacks can introduce inconsistencies that undermine correct operation. Previous work in this area, however, lacks a holistic methodology to tackle this problem and thus, addresses only certain parts of the problem. Yet, the consistency of the overall system is only as good as its least consistent part.**

Motivated by an analogy of network consistency checking with program testing, we propose to add active probe-based network state fuzzing to our consistency check repertoire. Hereby, our system, PAZZ, combines production traffic with active probes to periodically test if the actual forwarding path and decision elements (on the data plane) correspond to the expected ones (on the control plane). Our insight is that active traffic covers the inconsistency cases beyond the ones identified by passive traffic. PAZZ prototype was built and evaluated on topologies of varying scale and complexity. Our results show that PAZZ requires minimal network resources to detect persistent data plane faults through fuzzing and localize them quickly while outperforming baseline approaches.

***Index Terms*—Consistency, Fuzzing, Network verification, Software defined networking.**

I. INTRODUCTION

The correctness of a software-defined network (SDN) crucially depends on the consistency between the management, the control and the data plane. There are, however, many causes that may trigger inconsistencies at run time, including, switch hardware failures [1], [2], bit flips [3]–[8], misconfigurations [9]–[11], priority bugs [12], [13], control and switch software bugs [14]–[16]. When an inconsistency occurs, the actual data plane state does not correspond to what the control plane expects it to be. Even worse, a malicious user may actively try to trigger inconsistencies as part of an attack vector.

Figure 1 shows a visualization inspired by the one by Heller et al. [17] highlighting where consistency checks operate. The figure illustrates the three network planes – management, control, and data plane – with their components. The management plane establishes the network-wide policy P, which corresponds to the network operator's intent. To realize this policy, the control plane governs a set of logical rules (R_{logical}) over a logical topology (T_{logical}), which yield a set of logical paths (P_{logical}). The data plane consists of the actual topology (T_{physical}), the rules (R_{physical}), and the resulting forwarding paths (P_{physical}).

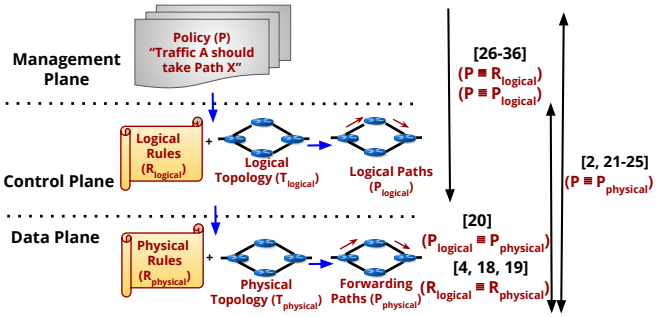


Figure 1: Overview of some of the relevant existing consistency checks.

Consistency checking is a complex problem. Prior work has tackled individual subpieces of the problem as highlighted by Figure 1. Monocle [4], RuleScope [18], and RuleChecker [19] use active probing to verify whether the logical rules R_{logical} are the same as the rules R_{physical} of the data plane. ATPG [2] creates test packets based on the control plane rules to verify whether paths taken by the packets on the data plane P_{physical} are the same as the expected path from the high-level policy P without giving attention to the matched rules. VeriDP [20] uses production traffic to only verify whether paths taken by the packets on the data plane P_{physical} are the same as the expected path from the control plane P_{logical}. NetSight [21], PathQuery [22], CherryPick [23], and PathDump [24] use production traffic whereas SDN Traceroute [25] uses active probes to verify $P \equiv P_{\text{physical}}$. Control plane solutions focus on verifying network-wide invariants such as reachability, forwarding loops, slicing, and black hole detection against high-level network policies both for stateless and stateful policies. This includes tools [26]–[36] that monitor and verify some or all of the network-wide invariants by comparing the high-level network policy with the logical rule set that translates to the logical path set at the control plane, i.e., $P \equiv R_{\text{logical}}$ or $P \equiv P_{\text{logical}}$. These systems, however, only *model* the network behavior which is insufficient to capture firmware and hardware bugs as "modeling" and verifying the control-data plane consistency are significantly different techniques.

Typically, previous approaches to consistency checking proceed "top-down," starting from what is known to the management and control planes, and subsequently checking whether the data plane is consistent. We claim that this is insufficient and underline this with several examples (§II-C)

wherein data plane inconsistencies would go undetected. This can be critical, as analogous to security, the overall system consistency is only as good as the weakest link in the chain.

We argue that we need to complement existing top-down approaches with a *bottom-up* approach. To this end, we rely on an analogy to program testing. Programs can have a huge state space, just like networks. There are two basic approaches to test program correctness: one is static testing and the other is dynamic testing using *fuzz testing* or fuzzing [37]. Hereby, the latter is often needed as the former cannot capture the actual run-time behavior. We realize that the same holds true for network state.

Fuzz testing involves testing a program with invalid, unexpected, or random data as inputs. The art of designing an effective fuzzer lies in generating semi-valid inputs that are *valid enough* so that they are not directly rejected by the parser, but do create unexpected behaviors deeper in the program, and are *invalid enough* to expose corner cases that have not been dealt with properly. For a network, this corresponds to checking its behavior not only with the expected production traffic but with *unexpected or abnormal* packets. However, in networking, what is expected or unexpected depends not only on the input (ingress) port but also the topology till the exit (egress) port and configuration, i.e., rules on the switches. Thus, there is a huge state space to explore. Relying only on production traffic is not sufficient because production traffic may or may not trigger inconsistencies. However, having faults that can be triggered at any point in time, due to a change in production traffic, whether malicious or accidental, is undesirable for a stable network. Thus, we need *fuzz testing for checking network consistency*. Accordingly, this paper introduces PAZZ, a methodology that combines such capabilities with previous approaches to verify SDNs against persistent data plane faults. Therefore, similar to program testing, we ask: *"How far to go with consistency checks?"*

Our Contributions:

- We identify and categorize the causes and symptoms of data plane faults that are currently unaddressed to provide some useful insights into the limitations of existing approaches by investigating their frequency of occurrence. Based on our insights, we make a case for fuzz testing mechanism for campus and private datacenter SDNs (§II);
- We introduce a novel methodology, PAZZ which detects and later, localizes faults by comparing control vs. data plane information for all three components, rules, topology, and paths. It uses production traffic as well as active probes (to fuzz test the data plane state) (§III);
- We develop and evaluate a PAZZ prototype[1] with multiple experimental topologies representative of multi-path/grid campus and private datacenter SDNs. Our results show that fuzzing through PAZZ outperforms the baseline approach in all experimental topologies while consuming minimal resources as compared to Header Space Analysis (HSA) [28] to detect and localize data plane faults (§IV).

[1]PAZZ software and experiments with topologies and configs are made publicly available at https://bitbucket.org/Apoorv1986/pazz/src/master/.

II. BACKGROUND & MOTIVATION

This section briefly navigates the landscape of faults and reviews the symptoms and causes (§II-A) to set the stage for the program testing analogy in networks (§II-B). Finally, we highlight the scenarios of data plane faults manifesting as inconsistencies (§II-C).

A. Landscape of Faults: Frequency, Causes and Symptoms

As per the survey by Zeng et al. [1], the primary causes for abnormal network behaviors or failures in the order of their frequency of occurrence are the following:
1) Software bugs: code errors, bugs, etc., *2) Hardware failures or bugs*: bit errors or bitflips, switch failures, etc., *3) Attacks and external causes*: compromised security, DoS/DDoS, etc., and *4) Misconfigurations*: ACL /protocol misconfigs, etc.

In SDNs, the above causes still exist and are persistent [2]–[4], [12]–[16], [38], [39]. We, however, realized that the symptoms cited in [1] of the above causes can manifest either as functional or performance-based problems on the data plane. To clarify, the symptoms are either functional (reachability, security policy correctness, forwarding loops, broadcast/multicast storms) or performance-based (high router's CPU utilization, congestion, latency/throughput, intermittent connectivity). To abstract the analysis, if we disregard the performance-based symptoms, we realize the functional problems can be reduced to the verification of network correctness. Making the situation worse, the faults manifest in the form of *inconsistencies* where the *expected* network state at control plane is different to the *actual* data plane state.

A physical network or network data plane comprises of devices and links. In SDNs, such devices are SDN switches connected through links. The data plane of the SDNs is all about the network behavior when subjected to inputs in the form of traffic. Just like in programs, we need different test cases as inputs with different coverage to increase code coverage. Similarly, in order to *dynamically* test the network behavior thoroughly, we need inputs in the form of traffic with different *coverage* [40]. Historically, network correctness or functional verification on the data plane has been either path-based verification (network-wide) [20]–[25], [41] or rule-based verification (mostly switch-specific) [2], [4], [18], [19], [25], [42]. Path-based verification can be end-to-end or hop-by-hop whereas rule-based verification performs switch-by-switch verification. The idea of network coverage brings us to the concept of Packet Header Space Coverage.

B. Packet Header Space Coverage: Active vs. Passive

We observe that networks just as programs can have a huge distributed state space. Packets with their packet

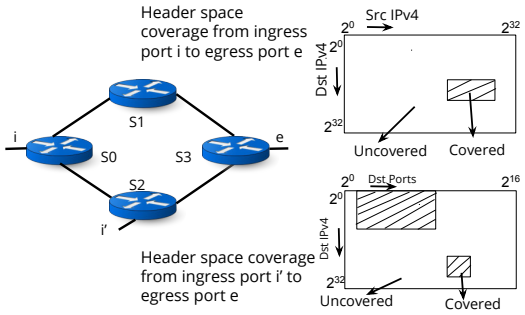


Figure 2: Example topology with two example ingress/egress port pairs (source-destination pairs) and their packet header space coverage (w.r.t IPv4 addresses and destination ports).

Related work in the data plane	Traffic Type (Packet header space coverage)	Type of monitoring/verification	
		Rule-based	Path-based
ATPG [2] [2]	Active	(✓)	(✓)
Monocle [4]	Active	(✓)	×
RuleScope [18]	Active	(✓)	×
RuleChecker [19][3]	Active	(✓)	×
SDNProbe [42]	Active	(✓)	(✓)
FOCES [41]	Passive	(✓)	(✓)
VeriDP [20]	Passive	(✓)	(✓)
NetSight [21]	Passive	(✓)	(✓)
PathQuery [22]	Passive	(✓)	(✓)
CherryPick [23][4]	Passive	(✓)	(✓)
PathDump [24]	Active	(✓)	(✓)
PAZZ	Active, Passive	✓	✓

Table I: Classification of related work in the data plane based on the type of the verification and the packet header space coverage. ✓denotes full capability, (✓) denotes a part of full capability, × denotes missing capability.

headers, including source IP, destination IP, port numbers, etc., are the inputs and the state includes all forwarding equivalence classes defined by the flow rules. Note, that every pair of ingress-egress ports (source-destination pair) can have different forwarding equivalence classes (FECs). We use the term *covered packet header space* to refer to it. Our motivation is that the FECs refer to parts of the packet header space. For a given pair of ingress and egress ports (source-destination pair), when receiving traffic on the egress port from the ingress port, we can check if the packet is covered by the corresponding "packet header space." If it is within the space it is "expected," otherwise it is "unexpected" and, thus, we have discovered an inconsistency due to the presence of a fault on the data plane.

Consider the example topology in Figure 2. It consists of four switches S0, S1, S2, and S3. Let us focus on two ingress ports i and i' and one egress port e. The figure also includes possible packet header space coverage. For i to e, it includes matches for the source and destination IPv4 addresses. For i' to e, it includes matches for the destination IPv4 address and possible destination port ranges. Indeed, it is possible to take into account other dimensions such as IPv6 header space, source-destination MAC addresses.

When testing a network, if traffic adheres to a specific packet header space, there are multiple possible cases. If we observe a packet sent via an ingress port i and received at an egress port e, then we need to check if it is within the covered area; if it is not we refer to the packet as "unexpected" and then, we have an inconsistency for that packet header space caused by a fault. If a packet from an ingress port is within the expected packet header space of multiple egress ports, we need to check if the sequence of rules *expected to be matched* and path(s) *expected to be taken* by the packet correspond to the actual output port(s) on data plane. This is yet another way of finding inconsistencies caused by faults.

Similar to program testing where negative test cases are used to fuzz test, in networking, we should not only test the network state with "expected" or production traffic, but also with specially crafted probe packets to test corner cases and "fuzz test" the network itself. In principle, there are two ways for testing network forwarding: passive and active. Passive corresponds to using the existing traffic or production traffic while active refers to sending specific probe traffic. The advantage of passive traffic is that it has low overhead and popular forwarding paths are tested repeatedly. However, production traffic may (a) not cover all cases (covers only faults that can be triggered by production traffic only), (b) change rapidly, and (c) have slow fault detection, as the fraction of traffic triggering the faults is intermittent. Indeed, malicious users may be able to inject malformed traffic that may trigger faults. Thus, production traffic may not cover the whole packet header space achievable by active probing.

Furthermore, we should also fuzz test the network state. This is important as we derive our network state from the information of the controller. Yet, this is not sufficient since we cannot presume that the controller state is complete and/or accurate. Thus, we propose to generate packets that are outside of the covered packet header space of an ingress/egress port pair. We suggest doing this by systematically and continuously or periodically testing the header space just outside of the covered header space. For example, if port 80 is within the covered header space, test for port 81 and 79. If `x.0/17` is in the covered header space, test for `x.1.0.0` which is part of the `x.1/17` prefix. In addition, we propose to randomly probe the remaining packet header space continuously or periodically by generating appropriate test traffic. The goal of active traffic generation through fuzzing is to detect the faults identifiable by active traffic only.

Table I shows the existing data plane approaches on the basis of a type of verification or monitoring in addition to the packet header space coverage. We see that existing data plane verification approaches are insufficient when it comes to both path- and rule-based verification in addition to ensuring sufficient packet header space coverage. In this paper, our methodology PAZZ aims to ensure packet header space coverage in addition to path- and rule-based verification to ensure network correctness on the data plane and thus, detecting and localizing persistent inconsistencies.

[2]In this tool, if the packet is received at the expected destination from a source, path is considered to be the same.

[3]In this tool, authors claim that tool may detect match and action faults *without guarantee*.

[4]In this tool, issues in only symmetrical topologies are addressed.

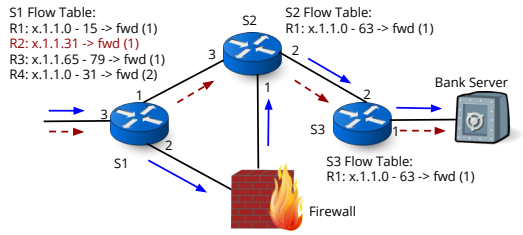


Figure 3: Example misconfiguration with a hidden rule. Expected/actual route ⇔ solid (blue)/dashed (red) arrows.

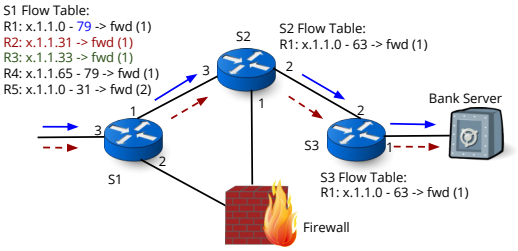


Figure 4: Example misconfiguration with a hidden rule detectable by active probing only. Expected/actual route ⇔ solid (blue)/dashed (red) arrows.

C. Data plane faults manifesting as inconsistencies

1) Faults identifiable by Passive Traffic: Type-p: To highlight the type of faults, consider a scenario shown in Figure 3. It has three OpenFlow-based SDN switches (S1, S2, and S3) and one firewall (FW). Initially, S1 has three rules R1, R3, and R4. R4 is the least specific rule and has the lowest priority. R1 has the highest priority. Note the rules are written in priority order.

Incorrect packet trajectory: We start by considering a *known* fault [20], [21], [24]–**hidden rule/misconfiguration**. For this, the rule R2 is added to S1 via the switch command line utility. The controller will remain unaware of R2 since R2 is a non-overlapping flow rule. Thus, it is installed without notification to the controller [43]. [4], [12] have hinted at this problem. As a result, traffic to IP `x.1.1.31` bypasses the firewall as it uses a different path.

Priority faults are another reason for such incorrect forwarding where either rule priorities get swapped or are not taken into account. The Pronto-Pica8 3290 switch with PicOS 2.1.3 caches rules without accounting for rule priorities [13]. The HP ProCurve switch lacks rule priority support [12]. Furthermore, priority faults may manifest in many forms: e.g., they may cause trajectory changes or incorrect matches even when the trajectory remains the same. **Action faults** can be another reason where bitflip in the action part of the flow rule may result in a different trajectory.

Insight 1: *Typically, packet trajectory monitoring tools only monitor the path.*

Correct packet trajectory, incorrect rule matching: If we add a higher priority rule in a similar fashion where the path does not change, i.e., the match and action remains the same as in the shadowed rule, then previous work will be unable to detect it and, thus, it is *unaddressed*.[5] Even if the packet trajectory is correct but a wrong rule is matched, it can create inconsistencies. Misconfigs, hidden rules, priority faults, match faults (described next) may be the reason for incorrect matches.

Next, we focus on **match faults** where an anomaly in the match part of a forwarding flow rule on a switch causes the packets to be matched incorrectly. We again highlight *known* as well as *unaddressed* cases starting with a known scenario. In Figure 3, if a bitflip,[6] e.g., due to hardware problems, changes R1 from `x.1.1.0/28` to match from `x.1.1.0` up to `x.1.1.79`, then traffic pertaining to `x.1.1.17` is now forwarded based on R1 rather than R4 and thus, bypasses the firewall. This may still be detectable, e.g., by observing the path of a test packet [20]. However, the bitflip in R1 also causes an overlap in the match of R1 and R3 in switch S1 and both rules have the same action, i.e., forward to port 1. Thus, traffic to `x.1.1.66` (supposed to be matched by R3) will be matched by R1. If later, the network administrator removes R3, the traffic pertaining to R3 still flows accordingly. This violates the network policy. For example, packets can be put into different queues or classes of service, leading to unexpected performance variations for some flows, possibly violating SLAs. It also leads to updating the wrong counters at the switches. In this paper, we categorize the faults detectable by the production traffic as *Type-p faults*.

Insight 2: *Even if the packet trajectory remains the same, the matched rules need to be monitored.*

2) Faults identifiable by Active Traffic only: Type-a: To highlight this class of faults, we focus on a hidden or misconfigured rule R3 (in green) in Figure 4. This rule matches the traffic corresponding to `x.1.1.33` on switch S1 and reaches the confidential bank server; however, the expected traffic or production traffic does not belong to this packet header space [38], [39], [44]. Therefore, we need to generate probe packets to trigger such rules and thus, detect their presence. This requires generating and sending traffic corresponding to the packet header space that is not expected by the control plane. We call this traffic as *fuzz traffic* in the rest of the paper since it tests the network behavior with unexpected packet header space. We categorize the faults detectable by only the active or fuzz traffic as *Type-a faults*.

Insight 3: *The tools that test rules check only rules "known" to the control plane (SDN controller) by generating active traffic for "known" flows.*

Insight 4: *Typically, active traffic for certain flows checks behavior only if the path remains the same even when matched rules may be different on the data plane.*

III. PAZZ METHODOLOGY

Motivated by our insights gained in §II-C about the Type-p and Type-a faults on the data plane resulting in inconsistencies, we aim to take consistency checks further. To this end, our methodology, PAZZ compares forwarding rules, topology, and paths of the control and the data plane, using top-down

[5] We validated this via experiments. OpenFlow specifications [43] state that if a non-overlapping rule is added via the switch command-line, the controller is not notified.

[6] A previously unknown firmware bug in HP 5406zl switch.

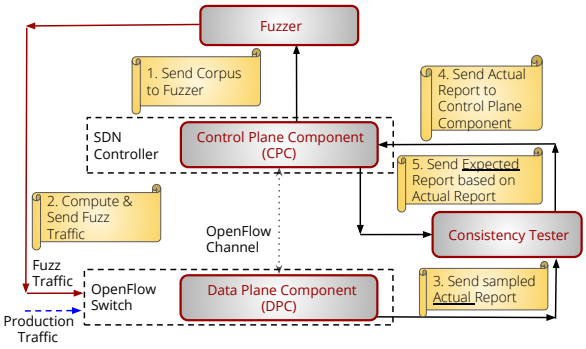


Figure 5: PAZZ Methodology.

and bottom-up approaches, to detect and localize data plane faults.

PAZZ, derived from *P*assive and *A*ctive (fu*ZZ* testing), takes into account both production and probe traffic to ensure adequate packet header space coverage. PAZZ checks the matched forwarding flow rules as well as the links constituting paths of various packet headers (5-tuple microflow) present in passive and active traffic. To detect faults, PAZZ collects state information (in terms of *reports*) from the control and the data plane: PAZZ compares the "expected" state reported by the control to the "actual" state collected from the data plane. Figure 5 illustrates the PAZZ methodology. It consists of four components:

1) **Control Plane Component (CPC):** Uses the current controller information to proactively compute the packets that are reachable between any/every source-destination pair. It then sends the corpus of seed inputs to Fuzzer. For any given packet header and source-destination pair, it reactively generates an expected report that encodes the paths and sequence of rules. (§III-B)

2) **Fuzzer:** Uses the information from CPC to compute the packet header space not covered by the controller and hence, by production traffic. It generates active traffic for fuzz testing the network. (§III-C)

3) **Data Plane Component (DPC):** For any given packet header and source-destination pair, it encodes the path and sequence of forwarding rules to generate a *sampled* actual report. (§III-A)

4) **Consistency Tester:** Detects and later, localizes faults by comparing the expected reports from the CPC with the actual reports from the DPC. (§III-D)

Now, we discuss in detail all components. To ease description, we follow a bottom-up order.

A. Data Plane Component (DPC)

To record the actual path of a packet and the rules that are matched in forwarding it, we rely on tagging the packet contained in active and production traffic. In particular, we propose the use of a shim header that gives us sufficient space

Algorithm 1: Data Plane Tagging

Input : (p, s, i, o, r) for each incoming packet p and switch with ID s let i be the inport ID and o the outport ID for packet p, r is the flow rule used for forwarding.
Output: Tagged packet p if necessary with the *Verify* shim header.

```
// Is there already a shim header, e.g., (s,i) is not
   an entry point or source port
if (p has no shim header) then
    // Add shim header with "Ethertype" 2080,
       initialize tag values- Verify_Port: entry point
       hash, Verify_Rule: 1.
    p.push_verify;
// Determine u_p ID from switch ID s and port ID i
u_p = s || i;
// Bloom filter
p.Verify_Port ← bloom(hash(u_p));
// Determine u_r ID from rule ID r of table ID t
u_r = s || r || t;
// Binary hash chain
p.Verify_Rule ← hash(p.Verify_Rule, u_r);
// Shim header has to be removed if (s,o) is exit point
if ((s, o) is exit point) then
    sample;
    if (p has no shim header) then
        // For traffic injected between a
           source-destination pair
        p.push_verify;
    Generate_report((s, o), p.Verify_Port, p.Verify_Rule,
      p.header); p.pop_verify;
```

even for larger network diameters or large flow rule sets. Indeed, INT [45] can be used for data plane monitoring; however, it is only applicable for P4 switches [46]. Unlike [20]–[24], we use a custom shim header for tagging; therefore, tagging is possible without limiting forwarding capabilities. To avoid adding extra monitoring rules on the scarce TCAM, which may also affect the forwarding behavior [22], [47], we augment OpenFlow with new actions. Between any source-destination pair, the new actions are used by all rules of the switches to add/update the shim header if necessary for encoding the sequence of inports (path) and matched rules. To remove the shim header, we use another custom OpenFlow action. To trigger submitting the actual report to the Consistency Tester, we use *sFlow* [48] sampling. Indeed, we can use any other sampling tool. Note sFlow is a *packet sampling* technique so it samples packets (and not flows) based on sampling rate and polling interval. For a given source, the report contains the packet header, the shim header content, and the egress port of the exit switch (destination).

Even with a shim header such as *Verify*, however, it is impractical to encode the required information about each port and rule in the path on the packet. Therefore, we rely on a combination of Bloom filters [49] and binary hash chains. For scalability, sampling is used before sending a report to the Consistency Tester.

Data Plane Tagging: To limit the overhead, we insert *Verify* shim header at layer-2. *VerifyTagType* is EtherType for *Verify* header, *Verify_Port* (Bloom filter) encodes the local inport in a switch, and *Verify_Rule* (binary hash chain) encodes the local rules in a switch. Thus, the encoding is done with the help of the Bloom filter and binary hash chain, respectively. To take actions on the proposed *Verify*

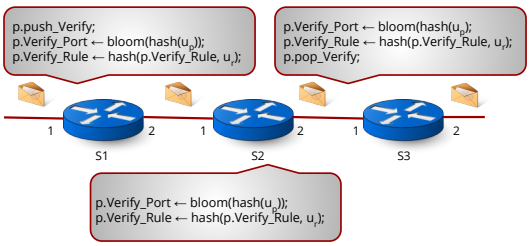


Figure 6: Data plane tagging using Bloom filters and hashing.

shim header and to save TCAM space, we propose to extend OpenFlow with four new actions: two for adding (*push_verify*) and removing (*pop_verify*) the *Verify* shim header and two for updating the *Verify_Port* (*set_Verify_Port*) and *Verify_Rule* (*set_Verify_Rule)* header fields, respectively. Since the header size of the *Verify_Port* and *Verify_Rule* and tagging actions are implementation-specific, we explain them in section §IV-A1, §IV-A2, respectively. Algorithm 1 explains the data plane tagging algorithm between a source-destination pair. For each packet either from production or active traffic (§III-C) entering the source inport, a *Verify* shim header will be added automatically by the switch. For each switch on the path, the tags in the packet: *Verify_Port* and *Verify_Rule* fields, get updated automatically. Figure 6 illustrates the per-switch tagging approach. Once the packet leaves the destination outport, the resulting report known as the *actual report* is sent to the Consistency Tester (§III-D). Note that if there is no *Verify* header, the *Verify* shim header is pushed on the exit switch to ensure that any traffic injected at any switch interface between a source-destination pair gets tagged. To reduce overheads at the Consistency Tester as well as on the switch, we employ sampling at the egress port. We continuously or periodically test the network as the data plane is dynamic due to reconfigurations, link/switch/interface failures, and topology changes. Note, periodic testing can also be executed by implementing timers.

B. Control Plane Component (CPC)

In principle, we can use the existing control plane mechanisms, including HSA [28], NetPlumber [29] and APVerifier [36]. In addition to experiments in [36], our independent experiments show that Binary Decision Diagram (BDD)-based [50] solutions like [36] perform better for set operations on headers than HSA [28] and NetPlumber [29]. In particular, we propose a novel BDD-based solution that supports rule verification in addition to path verification (APVerifier [36] takes into account only paths). Specifically, CPC performs two functions: a) *Proactive* reachability and corpus computation, and b) *Reactive* tag computation.

***Proactive* Reachability & Corpus Computation:** We start by introducing an abstraction of a single switch configuration called *switch predicate*. In a nutshell, a switch predicate specifies the forwarding behavior of the switch for a given set of incoming packets, and is defined in turn by the *rule predicates*. More formally, the general configuration abstraction of a SDN switch s with ports 1 to n can be described by switch predicates: $S_{i,j}$ where $i \in \{1, 2, ..., n\}$ and $j \in \{1, 2, ..., n\}$ where n denotes the number of switch ports. The packet headers satisfying predicate $S_{i,j}$ can be forwarded from port i to port j only. The switch predicate is defined via rule predicates: $R_{i,j}$ which are given by the flow rules belonging to the switch s and a flowtable t. Each rule has an identifier that consists of a $unique\_id$ and $table\_id$ representing the flowtable t in which the rule resides, in_port array representing a list of inports for that rule, out_port array representing a list of outports in the action of that rule and the rule priority p. Based on the rule priority p, in_port in the match part and out_port in the action part of a flow rule, each rule has a list of rule predicates (BDD predicates) that represent the set of packets that can be matched by the rule for the corresponding inport and forwarded to the corresponding outport.

Similar to the plumbing graph of [29], we generate a dependency graph of rules (henceforth, called rule nodes) called *reachability graph* based on the topology and switch configuration which computes the set of packet headers between any source-destination pair. There exists an edge between the two rules a and b, if (1) out_port of rule a is connected to in_port of b; and (2) the intersection of rule predicates for a and b is non-empty. For computational efficiency, each rule node keeps track of higher priority rules in the same table in the switch. A rule node computes the match of each higher priority rule, subtracting it from its own match. We refer to this as the *same-table dependency* of rules in a switch. In the following, by slightly abusing the notation, we will use switch predicates $S_{i,j}$ and rule predicates $R_{i,j}$ to denote also the set of packet headers: $\{p_1, p_2, ..., p_n\}$ they imply. Disregarding the ACL predicates for simplicity, the rule predicates in each switch s representing packet header space forwarded from inport i to outport j is given by $R^{fwd}_{i,j}$. The switch predicates are then computed as: $S_{i,j} = \cup R^{fwd}_{i,j}$

More specifically, to know the reachable packet header space (set of packet headers) between any source-destination pair in the network, we inject a fully-wildcarded packet header set h from the source port. If the intersection of the switch predicate $S_{i,j}$ and the current packet header p is non-empty, i.e., $S_{i,j} \cap \{p\} \neq \phi$, the packet is forwarded to the next switch until we reach the destination port. Thus, we can compute reachability between any/every source-destination pair. For caching and tag computation, we generate the inverse reachability graph simultaneously to cache the traversed paths and rules matched by a packet header p between every source-destination pair. After the reachability/inverse reachability graph computation, CPC sends the current switch predicates of the entry and exit switch pertaining to a source-destination pair to Fuzzer as a corpus for fuzz traffic generation (§III-C).

In case of a FlowMod action, the reachability/inverse reachability graph and new corpus are re-computed. Recall every rule node in a reachability graph keeps track of high-

priority rules in a table in a switch. Therefore, only a part of the affected reachability/inverse reachability graph needs to be updated in the event of rule addition/deletion. In the case of rule addition, the same-table dependency of the rule is computed by comparing the priorities of new and old rules before it is added as a new node in the reachability graph. If the priority of a new rule is higher than any rules and there is an overlap in the match part, the new switch predicate: $S'_{i,j}$ as per the new rule predicate: $R'^{fwd}_{i,j}$ is computed as:
$S'_{i,j} = R'^{fwd}_{i,j} \cup (R^{fwd}_{i,j} - R'^{fwd}_{i,j})$

***Reactive* Tag Computation:** For any given data plane report corresponding to a packet header p between any source-destination pair, we traverse the pre-computed inverse reachability graph to generate a list of sequences of rules that can match and forward the actual packet header observed at a destination port from a source port. Note, there can be multiple possible paths, e.g., due to multiple entry points and per-packet or per-flow load balancing. For a packet header p, the appropriate *Verify_Port* and *Verify_Rule* tags are computed as per Algorithm 1. The expected report is then sent to the Consistency Tester (§III-D) for comparison. Note we can generate expected reports for any number of source-destination pairs.

C. Fuzzer

Inspired by the code coverage-guided fuzzers like LibFuzz [51], we design a mutation-based fuzz testing component called Fuzzer. Fuzzer receives the corpus of seed inputs in the form of the switch predicates of the entry and exit switch from the CPC for a source-destination pair. In particular, the switch predicates pertaining to the inport of the entry switch (source) and the outport of the exit switch (destination) represent *expected* covered packet header space containing the set of packet headers satisfying those predicates. Fuzzer applies mutations to the corpus (Algorithm 2).

Where Can Most Faults Hide? Before explaining Algorithm 2, we present a scenario to explain the packet header space area where potential faults can be present. Consider the example topology illustrated in Figure 2. Due to a huge header space in IPv6 (128-bit), we decide to focus on the destination IPv4 header space (32-bit) in a case of destination-based routing. In principle, it is possible to consider IPv6 header space as the fuzzing mechanism remains the same. We use S_i, S_1 and S_e to represent covered packet header space (switch predicates) of switches $S0$, $S1$ and $S3$ between *i-e* (source-destination pair). Note there can be multiple paths for the same packet header *p*. Now, assume there is only a single path: $S0 \rightarrow S1 \rightarrow S3$, the reachable packet header space or net covered packet header space area is given by $S_i \cap S_1 \cap S_e$. Note this area corresponds to the control plane perspective so there may be more or less coverage on the data plane. The production traffic is generated in the area S_i which depends on the *expected* rules of $S0$ at an ingress port *i* for a packet header *p* destined to *e*. In principle, production traffic will cover the packet header space area S_i. Now, active traffic should be generated for the uncovered area $U - S_i(entryswitch)$ where U represents the universe of all possible packet header space which is 2^0 to 2^{32} for a destination IPv4 header space. As stated in §II-B, we need to start with active traffic generation on the boundary of the net covered packet header space area between a source-destination pair as there is a maximum possibility of faults in this area due to bitflips or "fat-finger" errors while configuring the network. A packet will reach e from i iff all rules on the switches in a path match it; else it will be dropped either midway or on the first switch. Therefore, for an end-to-end reachability, the ruleset on $S0$ and $S3$ should match the packet p contained in production traffic belonging to the covered packet header space: $S_i \cap S_e$. This implies that we need to first generate active traffic in the area: $S_e - S_i$ and then randomly generate in the leftover area. Traffic can, however, be also injected at any switch on any path between a source-destination pair, thus the checking needs to be done for different source-destination pairs.

Algorithm 2: Fuzzer

Input : Switch predicates of entry (S_i) and exit S_e switch for a source-destination pair i-e
Output: $fuzz\_sweep\_area, random\_fuzz\_area$

```
// Generate fuzz traffic in the difference of covered
   packet header space area between entry and exit
   switch
1 fuzz_sweep_area ← S_e − S_i;
2 residual_area ← U − fuzz_sweep_area − S_i;
3 random_fuzz_area ← Φ;
// Generate fuzz traffic in completely uncovered packet
   header space area randomly
4 while random_fuzz_area ≠ residual_area do
5     fuzz ← random_choose(residual_area);
6     random_fuzz_area ← random_fuzz_area ⋃ fuzz;
```

We now explain the Algorithm 2 in the context of Figure 2. For active or fuzz traffic generation, if there is a difference in the covered packet header space areas of S0 and S3, we first generate traffic in the area, i.e., $S_e - S_i$ denoted by $fuzz\_sweep\_area$ (Line 1). Recall, there is a high probability that there may be hidden rules in this area as the header space coverage of the exit switch may be bigger than the same at the entry switch. Later, we generate traffic randomly in the residual packet header space area, i.e., $U - fuzz\_sweep\_area - S_i$ denoted by $residual\_area$ (Lines 2-6). We generate traffic randomly in the area as this is mostly, a huge space and fault/s can lie anywhere. The fuzz traffic generated randomly is given by the completely uncovered packet header space area denoted by $random\_fuzz\_area$. Thus, the fuzz traffic that the Fuzzer generates belongs to the packet header space area given by $fuzz\_sweep\_area$ and $random\_fuzz\_area$. It is worth noting that not all of the packets generated by the fuzz traffic are allowed in the network due to a default drop rule in the switches. Therefore, if some packets in the fuzz traffic are matched, the reason can be attributed to either the presence of faulty rules, wildcarded rules or hardware/software faults to match such traffic. This also highlights that the fuzz traffic may not cause network congestion as there is no forwarding rule to

match most of the fuzz traffic. As discussed previously, there is another scenario where the traffic gets injected at one of the switches on the path between a source-destination pair and may end up getting matched in the data plane. Verify header is pushed at the exit switch if it is not already present (§III-A, §IV-A2) and thus, the packets still get tagged in the data plane to be sent in the actual report. However, the CPC may generate empty *Verify_Rule* and *Verify_Port* tags as the traffic is unexpected. In such cases, the fault is still detected but may not be localized *automatically* (§III-D). Furthermore, Fuzzer can be positioned to generate traffic at different inports to detect more faults in the network between *any/every* source-destination pair. If production traffic does not cover all expected rules at the ingress or entry switch, Fuzzer design can be easily tuned to also generate traffic for critical flows. Our evaluation confirms that an exhaustive active traffic generator, which randomly generates traffic in the uncovered area, performs poorly against PAZZ in real world topologies (§IV-E). Note if the network topology or configuration changes, the CPC sends the new corpus to the Fuzzer and Algorithm 2 is repeated. We can continuously or periodically test with fuzz traffic for any changes.

D. Consistency Tester

After receiving an actual report from the data plane, the Consistency Tester queries the CPC for its expected report for the packet header and the corresponding source-destination pair in the actual report. Once Consistency Tester has received both reports, it compares both reports as per Algorithm 3 for fault detection and the localization. To avoid confusion, we use $Verify\_Port_a$, $Verify\_Rule_a$ tags for the actual data plane report and $Verify\_Port_e$, $Verify\_Rule_e$ tags for the corresponding expected control plane report respectively. If *Verify_Rule* tag is different for a packet header and a pair of ingress and egress ports, then the fault is detected and reported (Lines 3-4). Note that we avoid the Bloom filter false positive problem by first matching the hash value for the *Verify_Rule* tag. Therefore, the detection accuracy is high unless a hash collision occurs in *Verify_Rule* field (§IV-A3). Once a fault is detected, Consistency Tester uses the *Verify_Port* Bloom filter for localization of faults where the actual path is different from the expected path, i.e., the $Verify\_Port_a$ Bloom filter is different from the $Verify\_Port_e$ Bloom filter (Lines 5-10). Therefore, $Verify\_Port_a$ is compared with the per-switch hop *Verify_Port* in the control plane or $Verify\_Port_{e^i}$ for the i^{th} hop starting from the source inport to the destination outport. This hop-by-hop walkthrough is done by traversing the reachability graph at the CPC hop-by-hop from the source port of the entry switch to the destination port of the exit switch. As per the Algorithm 3, the bitwise logical AND operation between the $Verify\_Port_a$ and the $Verify\_Port_{e^i}$ is executed at every hop. It is, however, important to note that if actual path was same as expected path, i.e., $Verify\_Port_e = Verify\_Port_a$ even when actual rules matched were different on the data plane, i.e., $Verify\_Port_e \neq Verify\_Port_a$ (Lines 12-15), the localization of faulty R_f gets tricky as it can be either a case of Type-p match faults (e.g., bitflip in match part) (Lines 12-13) or Type-a fault (Lines 14-15). Hereby, it is worth noting that there will be no expected report from the CPC in the case of *unexpected* fuzz traffic. Therefore, Consistency Tester checks if $Verify\_Rule_e \neq 0$ (Line 12). If true, localization can be done through hop-by-hop manual inspection of *expected* switches or manual polling of *expected* switches (Lines 12-13) else the Type-a fault *may* be localized to the entry switch as it has a faulty rule that allows the unexpected fuzz traffic in the network (Lines 14-15). There is another scenario where the actual rules matched are same as expected but the path is different (Lines 16-22). This is a case of Type1-action fault (e.g., bitflip in the action part of the rule). In this case, the expected and actual *Verify_Port* Bloom filter can be compared and thus, Type-p action fault is detected and localized. Note action fault is Type-p as it is caused in production traffic.

Algorithm 3: Consistency Tester (*detection, localization*)

Input : Actual and expected report containing the $Verify\_Rule_a$, $Verify\_Port_a$ and $Verify\_Rule_e$, $Verify\_Port_e$ tags respectively for a packet p pertaining to a flow (5-tuple)
Output: Detected and localized faulty switch S_f or Faulty Rule R_f.

1 **if** *No Actual report received in x seconds* **then**
    // Blackhole reported as no actual report was generated in x seconds for p.
2 Report blackhole by generating alarm
// Different rules were matched on data plane.
3 **else if** $(Verify\_Rule_a \neq Verify\_Rule_e)$ **then**
    // Fault is detected and reported.
4 Report fault
    // Different path was taken on data plane.
5 **if** $(Verify\_Port_a \neq Verify\_Port_e)$ **then**
        // Localize the fault.
6 **for** $i \leftarrow 0$ **to** n **by** 1 **do**
7 **if** $Verify\_Port_a \cap Verify\_Port_{e^i} = Verify\_Port_{e^i}$ **then**
8 No problems for this switch hop
            // Previous switch wrongly routed the packet.
9 **else**
10 $S_f \leftarrow S_{i-1}$
    // Path consistent even rules matched are different.
11 **else**
        // Localize Type-p match fault.
12 **if** $Verify\_Rule_e \neq 0$ **then**
13 Go through the different switches hop-by-hop to find R_f
        // Localize Type-a fault.
14 **else**
15 R_f lies in $S0$ else go through the different switches hop-by-hop
// Different path was taken on data plane.
16 **else if** $(Verify\_Port_a \neq Verify\_Port_e)$ **then**
    // Type-p action fault is detected and reported.
17 Report fault
    // Localize Type-p action fault.
18 **for** $i \leftarrow 0$ **to** n **by** 1 **do**
19 **if** $Verify\_Port_a \cap Verify\_Port_{e^i} = Verify\_Port_{e^i}$ **then**
20 No problems for this switch hop
            // Previous switch wrongly routed the packet.
21 **else**
22 $S_f \leftarrow S_{i-1}$
23 **else**
24 No fault detected

Binary hash chain in *Verify_Rule* gives PAZZ better accuracy, however, we lose the ability to *automatically* localize the Type-p match faults where the path remains the same and rules matched are different as the *Verify_Port* Bloom filter remains the same. To summarize, detection will happen always, but localization can happen automatically only in the case of two conditions holding simultaneously: a) when traffic is production; and b) when there is a change in path since *Verify_Port* Bloom filter will be different. In most cases, fuzz traffic is not permitted in the network. Recall active traffic can be injected from any switch in between a source-destination pair. In such cases, the R_f will still be detected and can be localized by either manual polling of the switches or hop-by-hop traversal from source to destination. Blackholes for critical flows can be detected as Consistency Tester generates an alarm after a chosen time of some seconds if it does not receive any packet pertaining to that flow (Lines 1-2). For localizing silent random packet drops, MAX-COVERAGE [52] algorithm can be implemented on Consistency Tester which can be co-located with the CPC or can be a standalone module.

IV. PAZZ PROTOTYPE AND EVALUATION

A. Prototype

1) DPC: Verify Shim Header: We decided to use a 64-bit (8 Byte) shim header on layer-2: *Verify*. To ensure sufficient space, we limit the link layer MTU to a maximum of 8,092 Bytes for jumbo frames and 1,518 Bytes for regular frames. *Verify* has three fields, namely:

- ***VerifyTagType*:** 16-bit EtherType to signify *Verify* header.
- ***Verify_Port*:** 32-bit encoding the local inport in a switch.
- ***Verify_Rule*:** 16-bit encoding the local rule/s in a switch.

We use a new EtherType value for *VerifyTagType* to ensure that any layer-2 switch on the path without our OpenFlow modifications will forward the packet based on its layer-2 information. The *Verify* shim header is inserted within the layer 2 header after the source MAC address, just like a VLAN header. In presence of VLAN (802.1q), the *Verify* header is inserted before the VLAN header. Note *Verify* header is backward compatible with legacy L2 switches, and transparent to other protocols.

2) DPC: New OpenFlow Actions: The new actions ensure that there is no interference in forwarding as no extra rules are added. To ensure efficient use of the shim header space, we use a Bloom filter to encode path-related information in the *Verify_Port* field and binary hash chains to encode rule-level information in the *Verify_Rule* field. A binary hash chain adds a new hash-entry to an existing hash-value by computing a hash of the existing hash-value and the new hash-entry and then storing it as the new value. The *Verify_Port* field is a Bloom filter which will contain all intermediate hash results of the path including the first (ingress switch) and last value (exit switch). This ensures that we can test the initial value as well as the final path efficiently.

- ***set_Verify_Port*:** Computes hash of the unique identifier (u_p) of the current switch ID and its inport ID and adds the result to the Bloom filter in the *Verify_Port* field.
- ***set_Verify_Rule*:** Computes hash of the globally unique identifier (u_r) of the current flow rule, i.e., switch ID and rule ID (uniquely identifying a rule within a table), flow table ID with the previous value of the *Verify_Rule* to form a binary hash chain.
- ***push_verify*:** Inserts a *Verify* header if needed, initializes the value in *Verify_Rule* to 1 and the value of *Verify_Port* is the hash of u_p. It is immediately followed by *set_Verify_Rule* and *set_Verify_Port*. If there is no *Verify* header, *push_verify* is executed at the entry and the exit switch between a source and destination pair.
- ***pop_verify*:** Removes the *Verify* header from the tagged packet.

push_verify should be used, if there is no *Verify* header for a) all packets entering a source inport or b) all packets leaving the destination outport (in case, if any traffic is injected between a source-destination pair) just before a report is generated to the Consistency Tester. For packets leaving the destination outport, *pop_verify* should be used only after a *sampled* report to the Consistency Tester has been generated. To initiate and execute data plane tagging, the actions *set_Verify_Port* and *set_Verify_Rule* are prepended to all flow rules in the switches as first actions in the OpenFlow "action list" [43]. On the entry and exit switch, action *push_verify* is added as the first action. On the exit switch, *pop_verify* is added as an action once the report is generated. Recall, our actions do not change the forwarding behavior per se as the match part remains unaffected. However, if one of the actions gets modified unintentionally or maliciously, it may have a negative impact but gets detected and localized later. Notably, *set_Verify_Rule* encodes the priority of the rule and flow table number in the *Verify_Rule* field and thus, providing support for rule priorities/cascaded flow tables.

3) Bloom filter & Hash Function: We use *Verify_Port* Bloom filter for the localization of detected faults. In an extreme case from the perspective of operational networks like datacenter networks or campus networks, for a packet header and a pair of ingress and egress port, if CPC computes i different paths with n hops in each of the paths, the probability of a collision in Bloom filter and hash value simultaneously will be given by: $(0.6185)^{m/n} \times p(i)$

In our case, $(0.6185)^{m/n} = (0.6185)^{32/n}$ using the Bloom filter false positive formula [49] as $m = 32$ (Bloom filter size) of the *Verify_Port* field and n is the network diameter or the number of switches in a path. $p(i)$ is the probability of collision of the hash function computed using a simple approximation of the birthday attack formula:
$p(i) = i^2/2H = i^2/2^{17}$

i is the number of different paths, H is 2^{16} for 16-bit *Verify_Rule* field hash.

For the 16-bit *Verify_Rule* hash operation, we used Cyclic Redundancy Check (CRC) code [53]. For the 32-bit *Verify_Port* Bloom filter operations, first, three hashes are com-

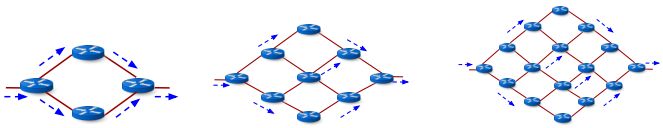


Figure 7: Left to right: Experimental grid topologies with 4, 9 and 16 switches respectively where fuzz and production traffic enter from the leftmost switch and leave at the rightmost switch in each topology.

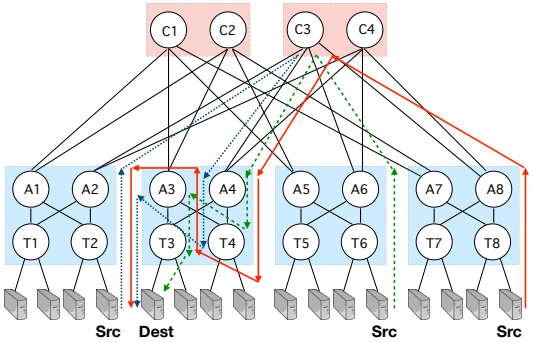


Figure 8: 4-ary fat-tree (20 switches) experimetal topology with expected paths in dotted blue line from three different sources (Src) to Destination (Dst). T< x >: ToR (Top-of-Rack), A< x >: Aggregate and C< x >: Core switches respectively. Paths for each source-destination pair: T2 A2 C3 A4 T4 A3 T3 (dotted blue), T6 A6 C3 A4 T4 A3 T3 (dashed green), T8 A8 C3 A4 T4 A3 T3 (solid red).

puted as: $g_i(x) = h1(x) + i \cdot h2(x)$ for $i = 0, 1, 2$ where $h1(x)$ and $h2(x)$ are the two halves of a 32-bit Jenkins hash of x. After, we use the first 5 bits of $g_i(x)$ to set the 32-bit Bloom filter for $i = 0, 1, 2$ [49].

PAZZ Components: We implemented DPC atop Open vSwitch (OvS) Version 2.6.90 software switches. The customized OvS switches and the fuzz/production traffic generators run in vagrant VMs. Currently, the prototype supports both OpenFlow 1.0 and OpenFlow 1.1. We chose Ryu SDN controller. Python-based Consistency Tester, Java-based CPC, and Python-based Fuzzer communicate via Thrift.

B. Experiment Setup

We evaluate PAZZ on 4 topologies: a) 3 grid topologies (Figure 7) of 4, 9 and 16 switches respectively with varying complexities to ensure diversity of paths, and b) 1 datacenter fat-tree (4-ary) topology of 20 switches with multipaths (Figure 8). Experiments were conducted on an 8 core 2.4GHz Intel-Xeon CPU machine and 64GB of RAM. For scalability purposes, we modified and translated the Stanford backbone configuration files [54] to equivalent OpenFlow rules as per our topologies, and installed them at the switches to allow multi-path destination-based routing. We used our custom script to generate configuration files for the four experimental topologies. The configuration files ensured the diversity of paths for the same packet header. Columns 1-4 in Table II illustrate the parameters of the four experimental topologies.

We randomly injected faults on randomly chosen OvS switches in the data plane where each fault belonged to different packet header space (in 32-bit destination IPv4 address space) either in the production or fuzz traffic header space. In particular, we injected Type-p (match/action faults) and Type-a faults. `ovs-ofctl` utility was used to insert the faults in the form of high-priority flow rules on random switches. Therefore, we simulated a scenario where the control plane was unaware of these faults in the data plane. We made a pcap file of the production traffic generated from our Python-based script that crafts the packets. In addition, we made a pcap of the fuzz traffic generated from the Fuzzer. The production and fuzz traffic pcap files were collected using Wireshark and replayed in parallel and continuously at the desired rate using Tcpreplay [55] with infinite loops to test the network continuously. To execute PAZZ periodically instead of continuously, there is an API support to implement timers with desired timeouts as per the deployment scenario. For sampling, we used sFlow [48] with a polling interval of 1 second and a sampling rate of 1/100, 1/500 and 1/1000. Note, the polling interval and sampling rate can be modified as per the deployment scenario. The sampling was done on the egress port of the exit switch in the data plane so the sampled actual report reaches the Consistency Tester and thus, avoids overwhelming it. Note each experiment was conducted for ten times for a randomly chosen source-destination pair.

Topology	#Rules	#Paths	Path Length	Reachability graph computation time	Fuzzer Execution Time
4 switches (grid)	~5k	~24k	2	0.64 seconds	~1 ms
9 switches (grid)	~27k	~50k	4	0.91 seconds	~1.2 ms
16 switches (grid)	~60k	~75k	6	1.13 seconds	~3.2 ms
4-ary fat-tree (20 switches)	~100k	~75k	6	1.15 seconds	~7.5 ms

Table II: Columns 1-4 depict the parameters of four experimental topologies. Column 5 depicts the reachability graph computation time by the CPC for the experimental topologies proactively by the CPC. Represents an average over 10 runs. Column 6 depicts the Fuzzer execution time to compute the packet header space for generating the fuzz traffic for the corresponding experimental topologies. Represents an average over 10 runs.

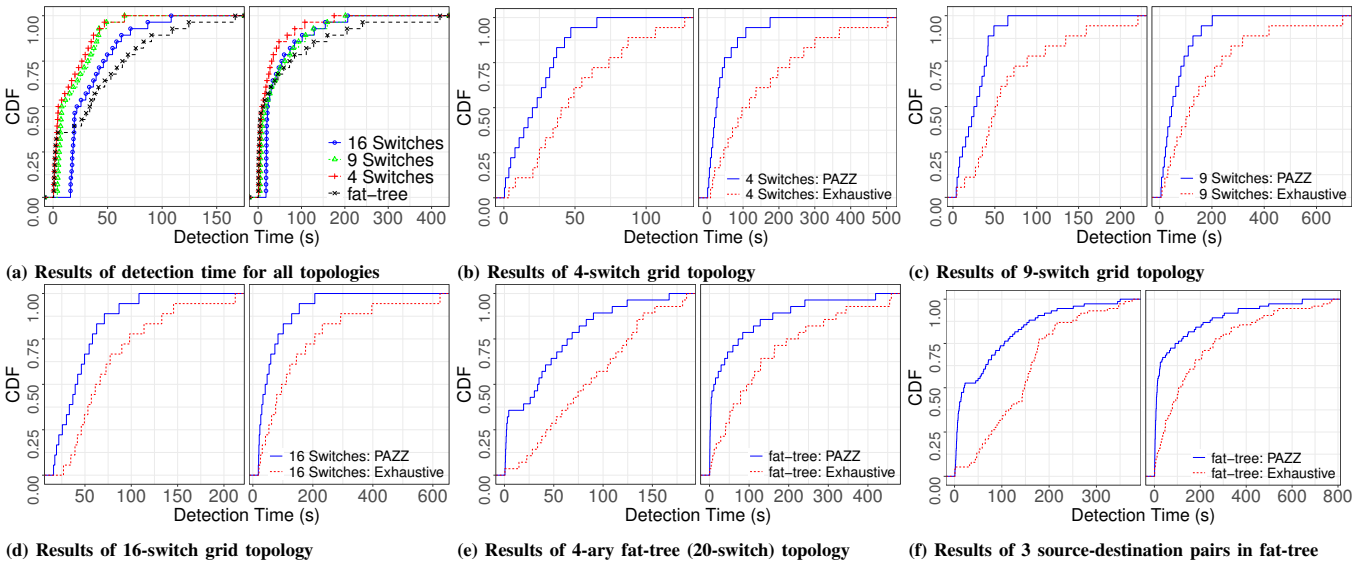
Workloads: For 1 Gbps links between the switches in the 4 experimental topologies: 3 grid and 1 fat-tree (4-ary), the production traffic was generated at 10^6 pps (packets per second). In parallel, fuzz traffic was generated at 1000 pps, i.e., 0.1% of the production traffic.

C. Evaluation Strategy

For a source-destination pair, our experiments are parameterized by: (a) size of network (4-20 switches), (b) path length (2-6), (c) configs (flow rules from 5k-100k), (d) number of paths (24k-75k), (e) number (1-28) and kind of faults (Type-p, Type-a), (f) sampling rate (1/100, 1/500, 1/1000) with polling interval (1 sec), and (g) workloads, i.e., throughput (10^6 pps for production and 1000 pps for fuzz traffic). Note, due to space constraints, we removed the results of 1/500 sampling rate. Our primary metrics of interest are fault detection with localization time, and comparison of fault detection/localization time in PAZZ against the baseline of exhaustive traffic generation approach and Header Space Analysis (HSA) [28].

In particular, we ask these questions:

Q1. *How does* PAZZ *perform under different topologies and configs of varying scale and complexity?* (§IV-D)



(a) Results of detection time for all topologies

(b) Results of 4-switch grid topology

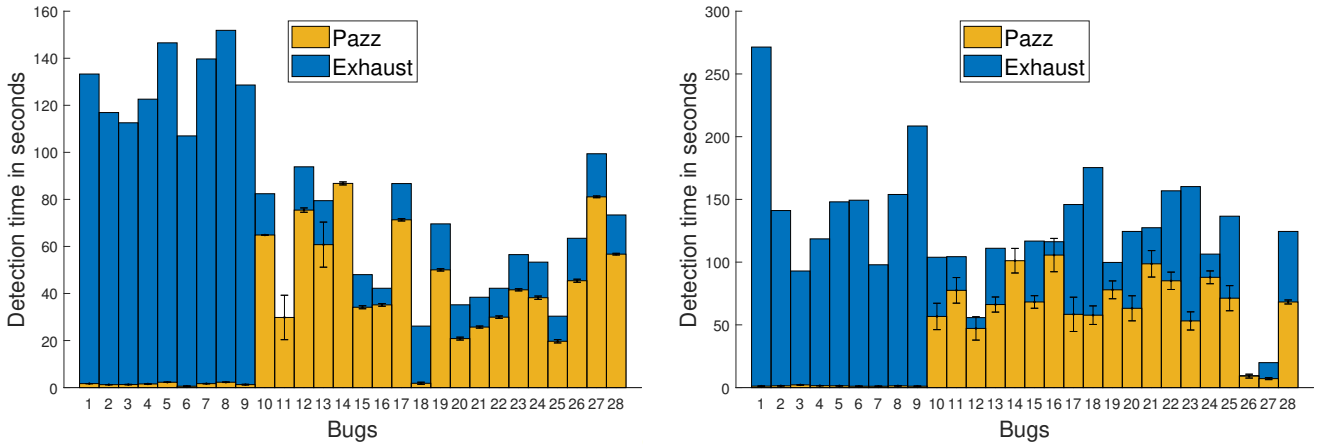
(c) Results of 9-switch grid topology

(d) Results of 16-switch grid topology

(e) Results of 4-ary fat-tree (20-switch) topology

(f) Results of 3 source-destination pairs in fat-tree

Figure 9: a) For a source-destination pair, CDF of Type-p and Type-a fault detection time by PAZZ in all 4 experimental topologies for sampling rate of 1/100 (left), and 1/1000 (right) respectively.
b), c), d), e) For a source-destination pair, comparison of fault detection time by PAZZ (blue) and exhaustive packet generation approach (red) in all 4 experimental topologies (4-switch, 9-switch, 16-switch and 4-ary fat-tree respectively). In each figure, left to right illustrates sampling rate of 1/100 (left), 1/1000 (right) respectively.
f) For 3 source-destination pairs, comparison of fault detection time (in seconds) by PAZZ (blue) and exhaustive packet generation approach (red) in 4-ary fat-tree topology.



(a) Results of 4-ary fat-tree (20-switch) with 1/100 sampling rate

(b) Results of 4-ary fat-tree (20-switch) with 1/1000 sampling rate

Figure 10: a) Median fault detection time in PAZZ with quartiles vs baseline for a single source-destination pair in 4-ary fat-tree topology with sampling rate of 1/100.
b) Median fault detection time in PAZZ with quartiles vs baseline for a single source-destination pair in 4-ary fat-tree topology with sampling rate of 1/1000.

Q2. *How does* PAZZ *compare to the strawman case of exhaustive random packet generation for a single and multiple source-destination pairs?* (§IV-E)
Q3. *How much time does* PAZZ *take to compute reachability graph as compared to Header Space Analysis [28]?* (§IV-F)
Q4. *How much time does* PAZZ *take to generate active traffic for a source-destination pair and how much overhead does* PAZZ *incur on the links?* (§IV-G)
Q5. *How much packet processing overhead does* PAZZ *incur on varying packet sizes?* (§IV-H)

D. PAZZ *Performance*

Figure 9a illustrates the cumulative distribution function (CDF) of the Type-p and Type-a faults detected in the four different experimental topologies with the parameters mentioned in Table II. As expected, in a grid 16-switch topology with 60k rules and 75k paths, PAZZ takes only 25 seconds to detect 50% of the faults and 105 seconds to detect all of the faults in case of sampling rate 1/100 and polling interval of 1 second (left in Figure 9a). For the same sampling rate of 1/100, in the case of 4-ary fat-tree topology with 20 switches containing 100k rules and 75k paths, PAZZ detects 50% of the faults in 40 seconds and all faults in 160 seconds. As the production traffic was replayed at 10^6 pps in parallel with the fuzz traffic replayed at 1000 pps, the Type-p faults in the production traffic header space (35% of total faults) were detected faster in a maximum time of 24 seconds for all four topologies as compared to the Type-a faults (65% of total faults) in the fuzz traffic header space which were detected in a maximum time of 420 seconds.As the experiment was conducted ten times, time taken is the mean of the ten values to detect a fault pertaining to a packet header space. Overall,

the detection time difference was marginal.

Localization Time: As per Algorithm 3, the production traffic-specific faults after detection were automatically localized within a span of 50 μsecs for all four experimental topologies. The localization of faults pertaining to fuzz traffic was manual as there was no expected report from the CPC. Hereby, the localization was done for two cases: a) when the fuzz traffic entered at the ingress port of the entry switch and b) when the fuzz traffic entered in between a pair of ingress and egress ports. For the first case, the localization of each fault happens in a second after the fault was detected by the Consistency Tester as the first switch possessed a flow rule to allow such traffic in the network. For the second case, i.e., where fuzz traffic was injected from between the pair of ingress and egress ports took approx. 2-3 minutes after detection for manual localization as the path was constructed after hop-by-hop inspection of the switch rules.

To compute the CPU usage in PAZZ, we measured the CPU time required for computing expected paths, test and detect inconsistency per sFlow sample analyzed by the Consistency Tester (brain of PAZZ) for the 4-ary fat-tree scenario. Our single-threaded implementation required 4.271 ms (mean), and 4.397 ms (median) with a minimum of 0.259 ms, and a maximum of 7.310 ms for each sample on a single core of the Intel Xeon E5-2609 2.40GHz CPU. For the memory usage, topology and configuration are the main factors. The fat-tree scenario with 100k rules requires the maximum resident set size (RSS) of 129MB. See §IV-F for memory usage of CPC.

E. Comparison to Exhaustive Packet Generation

We compare the fault detection time of PAZZ which uses Fuzzer against exhaustive packet generation approach. For a fair comparison, the exhaustive packet generation approach generates the same number of flows *randomly* and at the same rate like PAZZ. Figures 9b, 9c, 9d and 9e illustrate the fault detection time CDF in 4-switch, 9-switch, 16-switch and 4-ary fat-tree (20-switch) experimental topologies respectively. Two figures for each experimental topology illustrates the results for two different sampling rates of 1/100, and 1/1000 (left and right) respectively. The solid blue line indicates PAZZ which uses Fuzzer and the dotted red line indicates exhaustive packet generation approach. As expected, we observe that PAZZ performs better than exhaustive packet generation approach. PAZZ provides an average speedup of 2-3 times. We observe in all cases, 50% of the faults are detected in a maximum time of ~50 seconds or less than a minute by PAZZ. Note we excluded the Fuzzer execution time (§IV-G) in the plots. It is worth mentioning that PAZZ will perform much better if we compare against a fully exhaustive packet generation approach which generates 2^{32} flows in all possible destination IPv4 header space. Hereby, the detected faults are Type-a as they require active probes in the *uncovered* packet header space. Since, PAZZ relies on production traffic to detect the Type-p faults and thus, we get rid of the exhaustive generation of all possible packet header space.

Multiple Source-Destination Pairs & Header Coverage: We observed the similar results for different source-destination pairs when we placed Fuzzer and production traffic generator at different sources in parallel for a different header space coverage. Figure 9f shows the fault detection time comparison of PAZZ against exhaustive packet generation approach for 3 different source-destination pairs in the 4-ary fat tree topology (Figure 8). Note, PAZZ can be extended to Layer-2 and Layer-4 headers.

Figures 10a, 10b illustrate the median fault detection time with quartiles in PAZZ against the baseline for 4-ary fat-tree topology (20 switches) with 100k rules for a single source-destination pair for sampling rates: 1/100 and 1/1000 respectively. Note, 1-10 bugs belong to the Type-p category, i.e., detected by production traffic replayed at 10^6pps and 11-28 belong to the Type-a category, i.e., detected by fuzz traffic replayed at 1000 pps. We observe that in only 1 or 2 bugs out of total 28 bugs, PAZZ performs similar to the exhaust baseline as the detection is dependent on the sFlow sampling. Furthermore, the single-threaded implementation of Consistency Tester can analyze only a single flow at any point of time. Overall, we outperform the exhaust baseline in all four experimental topologies.

F. Reachability Graph Computation vs Header Space Analysis (HSA) [28]

Table II (Column 5) shows the reachability graph computation in all experimental topologies by the CPC for an egress port. To observe the effect of evolving configs, we added additional rules to various switches randomly. We observe that CPC always computes the reachability graph in < 1s.

We compare CPC against the state-of-the-art Header Space Analysis (HSA) [28] that can be used to generate reachability graphs to generate test packets and compute covered header space. HSA is also used by NetPlumber [29] and ATPG [2]. We observe that HSA library has scalability issues due to poor support for set operations on wildcards unlike BDDs [50]. Indeed, we reconfirm this by running a simple reachability experiment between a source-destination pair in the fat-tree topology of 20 switches with 100k rules in total. In this experiment, we compare the resource usage of the Python implementation of HSA library with the CPC of PAZZ which uses BDDs. HSA requires in excess of 206GB of memory to compute the 10 reachable paths between two ports, while the CPC requires only 1.5GB of memory.

G. Fuzzer Execution Time & Overhead

Execution Time: Table II (Column 6) illustrates the time taken by Fuzzer to compute the packet header space for fuzz traffic in the four experimental topologies after it receives the covered packet header space (corpus) from the CPC. Since, we considered destination-based routing hence, the packet header space computation was limited to 32-bit destination IPv4 address space in the presence of wildcarded rules. When

some of the rules were added to the data plane, the CPC recomputed the corpus, the new corpus was sent to the Fuzzer which recomputed the new fuzz traffic within a maximum time of 7.5 milliseconds.

H. DPC Overhead

We generated different packet sizes from 64 bytes to 1500 bytes at almost Gbps rate on the switches running the DPC software of PAZZ and the native OvS switches. We added flow rules on our switches to match the packets and tag them by the *Verify* shim header using our *push_verify*, *set_Verify_Rule* and *set_Verify_Port* actions in the flow rule. Then, we measured the average throughput over 10 runs.

We observe that the *Verify* shim header and the tagging mechanism incurred negligible throughput drop as compared to the native OvS. However, a throughput drop of 1.1.% is observed only on the entry switch/exit switch as *push_verify* actions happen on the entry switch/exit switch to insert the *Verify* shim header. Furthermore, sFlow sampling is done at the exit switch only. On the contrary, *set_Verify_Port* and *set_Verify_Rule* just set the *Verify_Port* and *Verify_Rule* fields respectively and have no observable impact. Thus, PAZZ introduces minimal packet processing overheads atop OvS.

V. Related Work

In addition to the related work covered in §I that includes the existing literature based on [17] and Table I, we now will navigate the landscape of related works and compare them to PAZZ in terms of the Type-p and Type-a faults which cause inconsistency (§II). The related work in the area of control plane [26]–[36] either check the controller-applications or the control-plane compliance with the high-level network policy. These approaches are insufficient to check the physical data plane compliance with the control plane. As illustrated in Table I, we navigate the landscape of the data plane approaches and compare them with PAZZ based on the ability to detect Type-p and Type-a faults. It is worth noting that the approaches either test the rules or the paths whereas PAZZ tests both together. In the case of Type-p match faults (§II-C1) when the path is consistent even if a different rule is matched, path trajectory tools [20]–[25], [41] fail. The approaches based on active-probing [2], [4], [18], [19], [25], [42] do not detect the Type-a faults (§II-C2) caused by hidden or misconfigured rules on the data plane which only match the fuzz traffic. These tools, however, only generate the probes to test the rules known or synced to the controller. Such Type-a faults are detected by PAZZ.

VI. Discussion: How far to go?

Motivated by our analysis of existing faults which cause inconsistency in SDNs and the fact that network consistency is only as good as its weakest link, we have argued that we need to take consistency checks further. Using an analogy to program testing, we proposed a novel "bottom-up" methodology, making the case for network state fuzzing.

However, the question of "how far to go?" in terms of consistency checks is a non-trivial one and needs further

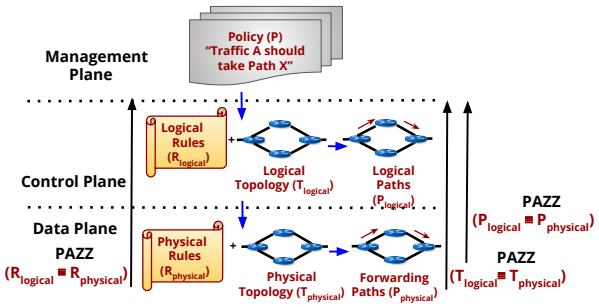


Figure 11: PAZZ provides coverage on the forwarding flow rules, topology and paths.

attention. In particular, while methodologies such as PAZZ provide a framework to systematically find (and subsequently remove) faults, *in theory*, they raise a question of *how much resources to invest*. Indeed, we observe that providing full coverage of forwarding flow rules, topology, and forwarding paths is inherently expensive: Consider a simplistic linear network of $n = 4$ switches with two links between each switch. Assume IP traffic enters from a source and exits on a destination, and each switch stores a simple forwarding rule: switch S_i for $i = \{1, 2, 3...n\}$ bases its forwarding decisions solely on the i^{th} least significant bit of the destination IP address: packets of even bits are forwarded to the upper link (*up*) and packets of odd bits to the lower (*lo*) link. While such a configuration requires only a small constant number of rules per switch, the number of possible paths is *exponential* in n: the set of paths is given by $\{up, lo\}^{n-1}$.

In addition to the combinatorial complexity of testing static configurations, we face the challenge that both the (distributed) control plane and the data plane state (as well as possibly the network connecting the controllers to the data plane elements) evolve over time, possibly in an asynchronous (or even stateful [26]) and hard to predict manner. This temporal dimension exacerbates the challenge of identifying faults, raising the question "how *frequently* to check".

Finally, the question of finding faults can go beyond deciding on the right amount of resources to invest: as any traffic-based testing introduces the inherent "Schrödinger's cat" problem that tagged traffic may not be subject to the same treatment as the untagged one.

We note that current trends towards programmable data planes [46] and hardware [56] may aid in solving some of the issues this paper raises. INT [45] in P4 specifically tries to solve the problem of network monitoring but is yet to be made scalable for the whole network and is limited to the P4 switches. Still, leveraging programmability, future SDNs will encompass even more possibilities of faults with a mix of vendor-code, reusable libraries, and in-house code. Yet, there will no longer be a common API, i.e., OpenFlow. As such the general problem will persist and we will have to explore how to extend the PAZZ methodology to programmable networks.

Overall, as illustrated in Figure 11, PAZZ checks consistency at all levels between control and the data plane, i.e., $P_{\text{logical}} \equiv P_{\text{physical}}$, $T_{\text{logical}} \equiv T_{\text{physical}}$, and $R_{\text{logical}} \equiv R_{\text{physical}}$.

VII. Conclusion & Future Work

This paper presented PAZZ, a novel network verification methodology that automatically detects and localizes the data plane faults manifesting as inconsistencies in SDNs. PAZZ periodically fuzz tests the packet header space and compares the expected control plane state with the actual data plane state. The tagging mechanism tracks the paths and rules at the data plane while the reachability graph at the control plane tracks paths and rules to help PAZZ in verifying consistency. Our evaluation of PAZZ over real network topologies and configurations showed that PAZZ efficiently detects and localizes the faults causing inconsistencies while requiring minimal data plane resources (e.g., switch rules and link bandwidth).

Currently, we do not handle transient faults, packet rewriting and checking of backup rules which lay the groundwork for our future work. For improving the performance of PAZZ further, we consider multithreading at the Consistency Tester will allow multiple flows to be checked in parallel. For DPC, Data Plane Development Kit (DPDK) or P4 can improve DPC performance. Distributed SDN controller and P4-specific hardware [46], [56] design and implementation will also be a part of our future work.

VIII. Acknowledgement

We thank Georgios Smaragdakis and our anonymous reviewers for their helpful feedback. This work and its dissemination efforts were conducted as a part of Verify project supported by the German Bundesministerium für Bildung und Forschung (BMBF) Software Campus grant 01IS17052.

References

[1] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "A Survey Of Network Troubleshooting," Stanford University, Tech. Rep. TR12-HPNG-061012, 2012.
[2] ——, "Automatic test packet generation," in *ACM CoNEXT*, 2012.
[3] P. Gawkowski and J. Sosnowski, "Experiences with software implemented fault injection," in *Architecture of Computing Systems (ARCS)*, 2007.
[4] P. Perešíni, M. Kuźniar, and D. Kostić, "Monocle: Dynamic, Fine-Grained Data Plane Monitoring," in *ACM CoNEXT*, 2015.
[5] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," in *ACM SIGMETRICS Performance Evaluation Review*, 2009.
[6] Z. Al-Ars, A. J. van de Goor, J. Braun, and D. Richter, "Simulation based analysis of temperature effect on the faulty behavior of embedded drams," in *Proceedings International Test Conference 2001 (Cat. No. 01CH37260)*. IEEE, 2001.
[7] NetworkWorld, "Cisco says router bug could be result of 'cosmic radiation' ... Seriously?" https://www.networkworld.com/article/3122864/cisco-says-router-bug-could-be-result-of-cosmic-radiation-seriously.html, 2016.
[8] Y. Yankilevich and G. Malchi, "Tcam soft-error detection method and apparatus," 2017, US Patent App. 15/390,465.
[9] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding bgp misconfiguration," in *ACM CCR*, 2002.
[10] UpGuard, "Cloud leak: Wsj parent company dow jones exposed customer data," https://www.upguard.com/breaches/cloud-leak-dow-jones/.
[11] Dyn.com, "Con-ed steals the 'net'," http://dyn.com/blog/coned-steals-the-net/.
[12] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *The Passive and Active Measurement (PAM) conference*, 2015.
[13] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *ACM SOSR*, 2016.
[14] J. Rexford, "SDN Applications," in *Dagstuhl Seminar 15071*, 2015.
[15] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A SOFT Way for Openflow Switch Interoperability Testing," in *ACM CoNEXT*, 2012.
[16] M. Kuzniar, P. Peresini, and D. Kostić, "Providing reliable fib update acknowledgments in sdn," in *ACM CoNEXT*, 2014.
[17] B. Heller, J. McCauley, K. Zarifis, P. Kazemian *et al.*, "Leveraging SDN layering to systematically troubleshoot networks," in *ACM HotSDN*, 2013.
[18] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li *et al.*, "Is every flow on the right track?: Inspect SDN forwarding with RuleScope," in *IEEE INFOCOM*, 2016.
[19] P. Zhang, C. Zhang, and C. Hu, "Fast testing network data plane with rulechecker," in *Network Protocols (ICNP)*. IEEE, 2017.
[20] P. Zhang, H. Li *et al.*, "Mind the gap: Monitoring the control-data plane consistency in software defined networks," in *ACM CoNEXT*, 2016.
[21] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks," in *USENIX NSDI*, 2014.
[22] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker, "Compiling Path Queries," in *USENIX NSDI*, 2016.
[23] P. Tammana, R. Agarwal, and M. Lee, "CherryPick: Tracing Packet Trajectory in Software-Defined Datacenter Networks," in *ACM SOSR*, 2015.
[24] ——, "Simplifying datacenter network debugging with pathdump," in *USENIX OSDI*, 2016.
[25] K. Agarwal, J. Carter, and C. Dixon, "SDN traceroute : Tracing SDN Forwarding without Changing Network Behavior," in *ACM HotSDN*, 2014.
[26] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "Buzz: Testing context-dependent policies in stateful networks," in *USENIX NSDI*, 2016.
[27] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the Data Plane with Anteater," in *SIGCOMM*, 2011.
[28] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks," in *USENIX NSDI*, 2012.
[29] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. Mckeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis," in *USENIX NSDI*, 2013.
[30] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks," in *USENIX NSDI*, 2014.
[31] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE Way to Test Openflow Applications," in *USENIX NSDI*, 2012.
[32] C. Scott, A. Wundsam, B. Raghavan, A. Panda *et al.*, "Troubleshooting blackbox sdn control software with minimal causal sequences," in *ACM CCR*, 2015.
[33] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide Invariants in Real Time," in *USENIX NSDI*, 2013.
[34] A. Fogel, S. Fung *et al.*, "A general approach to network configuration analysis." in *USENIX NSDI*, 2015.
[35] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking Beliefs in Dynamic Networks," in *USENIX NSDI*, 2015.
[36] H. Yang and S. S. Lam, "Real-Time Verification of Network Properties Using Atomic Predicates," in *IEEE/ACM Transactions on Networking*, 2016.
[37] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: whitebox fuzzing for security testing," in *Comm. of the ACM*, 2012.
[38] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, "How to detect a compromised sdn switch," in *Network Softwarization (NetSoft), 2015*. IEEE, 2015.
[39] G. Pickett, "Staying persistent in software defined networks," in *Black Hat*, 2015.
[40] G. Varghese, "Technical perspective: Treating networks like programs," in *CACM*, 2015.
[41] P. Zhang, S. Xu, Z. Yang, H. Li, Q. Li, H. Wang, and C. Hu, "Foces: Detecting forwarding anomalies in software defined networks," in *ICDCS*. IEEE, 2018.
[42] Y. Ke, H. Hsiao, and T. H. Kim, "SDNProbe: Lightweight Fault Localization in the Error-Prone Environment," in *ICDCS*. IEEE, 2018.
[43] OpenFlow Spec, "https://www.opennetworking.org/images//openflow-switch-v1.5.1.pdf," 2015.
[44] M. Antikainen, T. Aura, and M. Särelä, "Spook in your network: Attacking an sdn with a compromised openflow switch," in *Nordic Conference on Secure IT Systems*. Springer, 2014.
[45] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.
[46] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," in *ACM CCR*, 2014.
[47] H. Zhang, C. Lumezanu, J. Rhee, N. Arora, Q. Xu, and G. Jiang, "Enabling Layer 2 Pathlet Tracing through Context Encoding in Software-Defined Networking," in *ACM HotSDN*, 2014.
[48] S. Panchen, P. Phaal, and N. McKee, "Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks," in *RFC 3176*, 2001.
[49] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: building a better bloom filter," in *ESA*. Springer, 2006.
[50] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," in *IEEE Trans. Comput.*, 1986.
[51] LLVM.org, "Llvm Compiler Infrastructure: libfuzzer: a library for coverage-guided fuzz testing," http://llvm.org/docs/LibFuzzer.html, 2019.
[52] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren, "Detection and localization of network black holes," in *IEEE INFOCOM*, 2007.
[53] G. Castagnoli, J. Ganz, and P. Graber, "Optimum cycle redundancy-check codes with 16-bit redundancy," in *IEEE Transactions on Communications*. IEEE, 1990.
[54] P. Kazemian, "Hassel-public," https://bitbucket.org/peymank/hassel-public/, 2016.
[55] A. Turner and M. Bing, "Tcpreplay: Pcap editing and replay tools for* nix," http://tcpreplay.sourceforge.net, 2005.
[56] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *USENIX NSDI*, 2018.