P4CONSIST: Towards Consistent P4 SDNs

Apoorv Shukla, Seifeddine Fathalli,
Thomas Zinner, *Member, IEEE,* Artur Hecker, Stefan Schmid, *Member, IEEE*

***Abstract*—The prevailing 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 between the network control and the data plane that can undermine the correct operation at runtime. Through our experiments, we realize that P4 SDNs are no exception, and are prone to similar problems.**

With the aim to verify the control-data plane inconsistency, we present the design and implementation of P4CONSIST, a system to detect the inconsistency between control and data plane in P4 SDNs. P4CONSIST generates active probe-based traffic continuously or periodically as an input to the P4 SDNs to check whether the actual behavior on the data plane corresponds to the expected control plane behavior. In P4CONSIST, the control plane and the data plane generate independent reports which are later, compared to verify the control-data plane consistency. The previous works in the field of monitoring and verification mostly aim to test the P4 programs through static analysis and thus, are insufficient to verify the network consistency at runtime. Experiments with our prototype implementation of P4CONSIST are promising and show that P4CONSIST can verify the control-data plane consistency in the complex datacenter 4-ary fat-tree (20 switches) and multipath grid (4, 9 and 16 switches) topologies with 60k rules per switch within a minimum time of 4 minutes. At the same time, P4CONSIST scales to multiple source-destination pairs to detect control-data plane inconsistency.

***Index Terms*—P4, Software-Defined Networking, Network Verification.**

I. INTRODUCTION

Software-Defined Networks (SDNs) [1] introduced a novel paradigm to networking: programmability. While SDN provides many benefits, ranging from traffic-engineering to simplified centralized network management, its programmability is limited to the remote control plane or the decision element only. With the emergence of P4 [2], [3], the programmability of the network data plane can be enabled without sacrificing line-rate performance. The flexibility of data plane programming allows network administrators to enable complex custom network policies independent of the target devices, e.g., switches. P4 is a domain-specific language that allows the programmer to customize the functionality of the device on the data plane. In addition to quick innovation, this opens up opportunities for novel uses of the network: e.g., in-band network telemetry [4] and in-network caching [5], [6]. The increased capabilities of the programmability, however, are the harbinger of new challenges to the network correctness.

However, just like any other program, P4 programs are prone to software or hardware bugs [7]–[13] or dataplane faults. The faults can manifest in many ways e.g., resulting in abnormal network behavior different from the expected behavior or in other words, the control plane may be completely unaware of the abnormal data plane behavior caused by the faults on the dataplane. Contrary to the prevailing wisdom, such situation violates the premise that the logically centralized control plane has a consistent view of actual data plane state. Thus, it can inflict serious damages to the network infrastructures and business units as it causes incorrect network operation or even a security compromise. This may incur heavy revenue losses [14]–[18] and in addition, network debugging is a costly practice due to manual effort involved in debugging as the existing tools like Ping, Traceroute, SNMP are rendered insufficient in many scenarios.

As important as these issues are, we argue that the correctness crucially depends on the consistency between the control and the data plane. Traditionally, there have been many causes that may trigger inconsistencies at run time, including, switch hardware failures [15], [19], [20], bit flips [21], [22], misconfigurations [16]–[18], [23]–[25], priority bugs [26], [27], control and switch software bugs [28]–[30]. We realize that P4 SDNs are prone to such problems as well. 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.

The recent works in P4 verification [7]–[12], [31] have mostly focused on the static analysis of the P4 programs to detect bugs in the P4 source code. Static analysis, however, tests the program without passing any inputs and thus, is prone to false positives. Unless populated by the control plane at run time, the contents of the Match-Action tables are unknown. The Match-Action rules installed by the control plane at run time and not the P4 program determine the effect of executing these tables on any given packet. The data plane behavior and properties cannot be stated unless we have a deep understanding of the control-data plane interaction and an accurate view of the data plane. Therefore, it is important to verify whether the *actual* data plane behavior corresponds to the *expected* high-level network policy. Since, the target switch does not throw any run time exceptions, detecting bugs is an uphill battle. To elaborate further, different inputs at run time may trigger unknown or abnormal behaviors. The existing related works are, hence, insufficient as they

Apoorv Shukla is with the Technische Universität Berlin (e-mail: apoorv@inet.tu-berlin.de), Seifeddine Fathalli is with the MPI-Informatics (e-mail: fathalli@mpi-inf.mpg.de), Thomas Zinner is with the Department of Information Security and Communication Technology, Norwegian University of Science and Technology, 7491 Trondheim, Norway (e-mail: thomas.zinner@ntnu.no), Artur Hecker is the Director of Future Networks Research at Huawei Technologies, Munich (e-mail: Artur.Hecker@huawei.com), and Stefan Schmid is with the Faculty of Computer Science, University of Vienna (email: stefan_schmid@univie.ac.at).

Thomas Zinner and Seifeddine Fathalli were affiliated with the Technische Universität Berlin while working on this paper.

do not check the control-data plane consistency at runtime. Recently, [13] performed runtime verification of a single P4 switch leveraging reinforcement learning-guided fuzzing. However, such approach cannot detect a P4 network-wide control-data plane inconsistency. Therefore, there is an urgent need for the mechanisms to verify the P4 networks, e.g., P4 SDNs, at runtime as these are missing from the current P4 network verification repertoire.

To debate about consistency in networks, there is a P4-programmed data plane and a programmed control plane. *The challenge is three-fold: 1) elicit the data plane state, 2) gather the control plane state, and 3) compare both of them for detecting inconsistency*. As the data plane faults may not be reflected on the control plane, we need to get independent reports from the control and the data plane and then, compare them. The faults in the P4 programmed-data plane can be due to e.g., a software bug in the P4 program which may cause the path deviation of the input traffic on the actual data plane from the intended network policy or the expected control plane. Such path deviation on the data plane can possess a big threat if it bypasses a waypoint, e.g., firewall and thus, allowing the malicious traffic to attack the critical infrastructure. There can be, however, a plethora of different faults resulting in abnormal behavior of the network. In addition, consistency is hard and complex to ensure as it applies to different packet header space between different source-destination pairs. For instance, in a destination-based routing, the consistency checking may involve testing for 2^{32} possible IPV4 destination addresses or header space between any given source-destination pair.

Problem Statement In this paper, we ask the following question: ***In P4 SDNs, for a given packet, is the control plane consistent with the data plane between a given source-destination pair?***

In this paper, we present a system, P4CONSIST where for a given packet (5-tuple flow) and a source-destination pair, the ***control plane module*** proactively gathers the topology and configuration information in the form of an *expected report*. The ***data plane module*** customizes the INT (In-band Network Telemetry) [4] with MRI (Multi-Hop Route Inspection) [32] to continuously or periodically elicit telemetry data encoding the forwarding behavior of each switch for the input traffic from the input traffic generator in the form of an *actual report*. Finally, the ***analyzer*** compares the two reports for detecting control-data plane inconsistencies using depth-first search (DFS) and symbolic execution. To summarize, on the basis of independent control and data plane reports, we investigate the control-data plane consistency aspects for critical flows (given 5-tuple flows) between any source-destination pair/s.[1]

To evaluate our approach, we built a prototype of P4CONSIST using software switch: Bmv2 (behavioral model version 2) [33] of P4 version 16 (P4-16) connected with the remote network control plane through P4 Runtime API [34]. We conducted experiments on four different topologies and configurations of variable scale to gauge the performance of P4CONSIST. Our results show that the proposed verification process can uncover a broad range of faults such as conflicting rules added through different configuration channels, invalid packet header reads or writes in the P4-programmed data plane. A detailed performance analysis also shows that, although the verification time increases with the size of the network e.g., configurations and devices, pragmatically our approach quickly verifies the control-data plane consistency in various topologies and configurations of multiple scale. Finally, we initiate a discussion on the importance of consistency verification and to what extent we can do the consistency checks.

Our Contributions:

- We identify the need for verifying the control-data plane consistency through example scenarios (§II);
- To address the example scenarios, we present the design of a novel system: P4CONSIST that aims to detect the control-data plane inconsistency by comparing the independent reports from the control and the data plane in an automated manner (§III);
- We develop and prototype P4CONSIST in the software switch behavioral model version 2 (BMv2) of P4 version16 (P4-16), and evaluate it on multiple topologies of variable scale and configurations. Our results show that P4CONSIST can detect control-data plane inconsistency within a minimum time of 4 minutes in the complex datacenter 4-ary fat-tree (20 switches) and multipath grid (4, 9 and 16 switches) topologies with 60k rules per switch (§IV).
- To ensure reproducibility and facilitate the follow up work, we release the P4CONSIST software and experiments at: https://gitlab.inet.tu-berlin.de/apoorv/P4CONSIST.

Organization: In the rest of this paper, we will first give more background on the P4 language and P4 Runtime and then, identify the example scenarios of the control-data plane inconsistency (§II). Next, we present an overview of P4CONSIST system (§III). Then, we give details on the implementation of P4CONSIST and evaluate its performance (§IV). After discussing the limitations and future work (§V), we go through the related work (§VI). Finally, we initiate a discussion on the extent of consistency checks (§VII) to conclude the paper (§VIII).

II. BACKGROUND & MOTIVATION

This section briefly reviews the P4 language, P4 Runtime and P4 SDNs in general and motivates the scenarios of control-data plane inconsistency in P4 SDNs. Based on this motivation, we later present the design (§III) and implementation (§IV) of P4CONSIST to detect such inconsistencies.

A. Background on P4

1) P4 Language: P4 is a domain-specific language [2], [3] that came into existence to enhance SDN as in SDNs, the SDN switch could only be configured and switch chip was fixed-function and supported limited protocols due to fixed parser. With P4, one can program the switch chip as the parser can be programmed, to implement customised protocols. P4 is based

[1]Note we focus on persistent data plane faults which manifest as control-data plane inconsistency. Transient faults are out of the scope of this paper as control plane eventually converges to the correct view of the network.

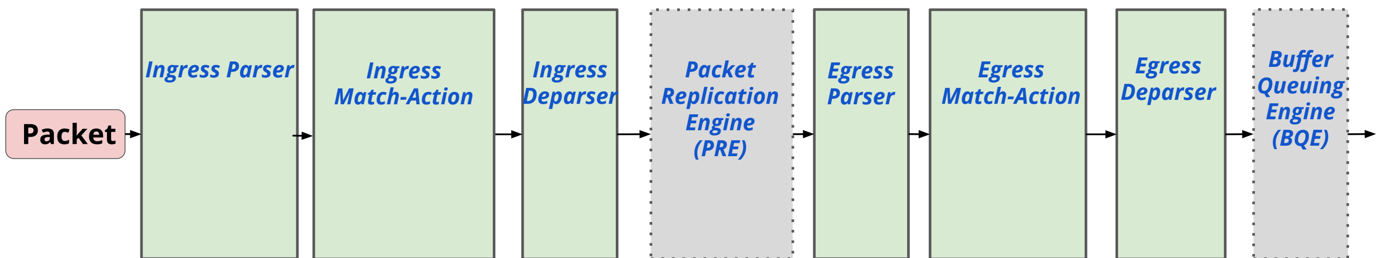


Figure 1: $P4_{16}$ packet processing pipeline. [3]

on an abstract forwarding model called protocol-independent switch architecture (PISA) [35], comprising of packet-processing abstractions, e.g., headers, parsers, tables, actions, and controls. Unless programmed, the P4 device or switch does not understand any protocol and thus, it is protocol-independent. The language has two main versions, $P4_{14}$ and $P4_{16}$. As shown in Figure 1, in $P4_{16}$ packet processing pipeline, there are six programmable blocks in green, namely, `ingress parser`, `ingress match-action`, `ingress deparser`, `egress parser`, `egress match-action`, and `egress deparser`. Note, `packet replication engine (PRE)` and `buffer queuing engine (BQE)` are non-programmable and target dependent (in gray color in Figure 1).

P4 pipeline. The ingress parser transforms the packet from bits into headers according to a parser specification provided by the programmer and decides what packet headers are recognized and in what order. After parsing, an ingress match-action (also called ingress control function) decides how the packet will be processed. Then, the packet is queued for egress processing in the ingress deparser. Upon dequeuing, the packet is processed by an egress match-action (also called egress control function). The egress deparser specification dictates how packets are deparsed from separate headers into a bit representation on output, and finally, the packet leaves the switch.

A P4 program is written in $P4_{14}$ or $P4_{16}$ and then compiled via P4 compiler called p4c [36] to be deployed to run on a P4 switch. At the runtime, via the control plane or an SDN controller or even manually, the match-action rules are populated on the match-action table of a P4 switch. To draw a comparison between P4 and SDN, it is worth noting that in P4, excluding an SDN controller, the usual SDN entities such as OpenFlow and the non-programmable SDN switches, e.g., OpenvSwitch (OvS) get replaced by P4 Runtime [34], [37] and programmable P4 switch platforms, e.g., behavioral model (bmv2) [33], [38], Tofino [39], etc. respectively. Figure 2 illustrates as P4 SDN where P4 Runtime controls the P4 switches on the data plane via SDN Controller. P4 Runtime channel pushes the configuration in the form of forwarding rules from the SDN Controller in the control plane to the P4 switches in the data plane.

B. Motivating Examples

In this section, we motivate the need for P4CONSIST through two practical example scenarios of the control-data plane inconsistency rooted within existing network management practices. These scenarios are easy to reproduce and the inconsistency is undetectable by the existing verification tools in P4.

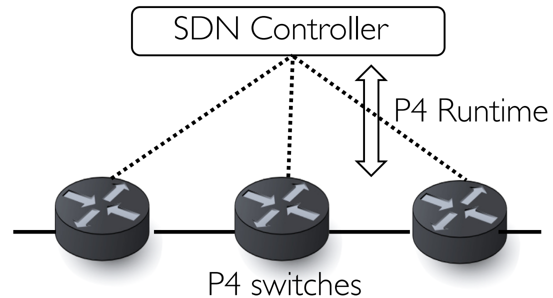


Figure 2: P4 SDN where SDN Controller controls the P4 switches (running the compiled P4 programs) on the data plane via P4 Runtime.

Example Scenario 1 — Problem of multiple configuration channels: Figure 3 illustrates an example scenario of control-data plane inconsistency. The SDN controller installs the P4 program and populates the Match-Action tables with the corresponding rules through the P4 Runtime API. The packets are expected to be forwarded through the *expected path* S1-S3-S2-S4 (shown by the dashed blue arrows). If an attacker maliciously or a network administrator accidentally modifies a rule in S1 through switch CLI (the thrift API debugging channel), it causes the traffic to go through the *actual path* S1-S2-S4 (shown by the dashed red arrows) and thus, bypass the firewall on switch S3. This allows malicious or malformed packets to reach the critical server and inflict serious damages. We realized in such a case, no notification is sent to the SDN controller about the modified rule and thus, the controller stays unaware of the modified rule. Therefore, the state at the data plane is inconsistent with the state at the control plane resulting in inconsistency. *We reproduced such scenario with the P4 software switch bmv2 target:* `simple_switch_grpc` *where we modified a rule through the thrift API debugging channel of the switch without the SDN controller getting notified.* Traditionally, OpenFlow is prone to such problems when rules are modified using `ovs-ofctl` switch CLI [40]. For OpenFlow-based SDNs, [22], [26] have hinted at this problem.

In addition, **Priority faults** [41] 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 [27]. The HP ProCurve switch lacks rule priority

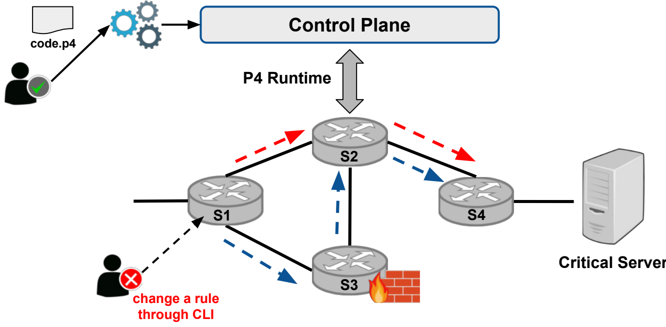


Figure 3: Example scenario where the P4 switches were initially provisioned by the control plane through the P4Runtime channel; installing the P4 program and populating the Match-Action table with the corresponding rules; then maliciously or accidentally the rules were modified through the switch debugging CLI channel (Thrift API) causing control-data plane inconsistency where the packets instead of expected path (in dashed blue arrows) through firewall take a different actual path (in dashed red arrows) bypassing the firewall.

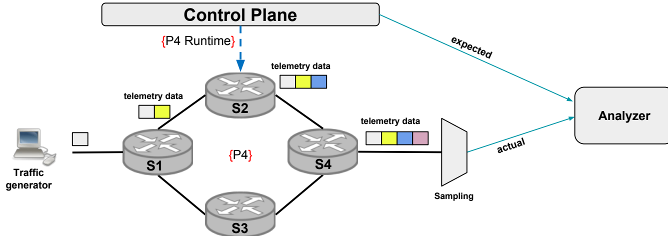


Figure 4: P4CONSIST architecture.

support [26]. Furthermore, priority faults may manifest in many forms e.g., they may cause the trajectory changes or incorrect matches even when the trajectory remains the same. **Action faults** [20] can be another reason where, e.g., a bitflip in the action part of the rule may result in a different trajectory. We have seen priority faults and action faults in OpenFlow-based hardware, however, we expect the similar problems with the P4-based hardware.

Example Scenario 2 — No runtime exceptions in a P4 switch: A P4 switch does not throw any runtime exceptions and, therefore, it is incredibly hard to catch or localize a bug at runtime. There are many issues that can throw runtime exceptions such as if an uninitialized packet header field is read, the P4 switch behavior is unspecified, i.e., an incoming packet may get modified in an unspecified manner. This implies that the packet could either be dropped or a quasi-random value is returned. P4 language specification [3] explains that the behavior of the program implementing firewall may be undefined on certain kinds of packets as reading or writing an invalid header produces an arbitrary result on the data plane which is unknown to the SDN controller and thus, causing inconsistency. Such a scenario is possible when the acl table can correctly match and filter away IPv4 packets sent by external hosts, however, it might incorrectly forward other types of packets, e.g., IPv6. Such scenarios may cause control-data plane inconsistency which can be non-trivial to detect if one relies on the prevailing wisdom, i.e., SDN controller has an accurate view of the network and incorrectly takes the controller view as the ground truth.

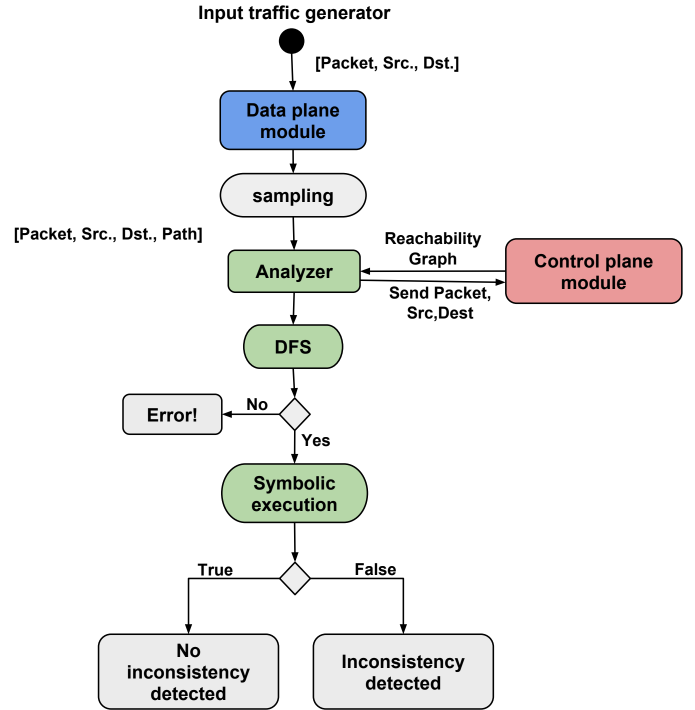


Figure 5: P4CONSIST workflow. Red: Control Plane Module, Blue: Data plane Module, Green: Analyzer and its components: DFS & Symbolic Execution, and Black: Input Traffic Generator.

III. SYSTEM: P4CONSIST

P4CONSIST, derived from P4 CONSISTency, checks the control-data plane consistency between any source-destination pair through the following steps. Figure 4 illustrates the P4CONSIST system architecture. To explain the architecture, it is important to understand the P4CONSIST workflow that describes the roles of each module, i.e., *Control Plane Module*, *Data Plane Module*, *Input Traffic Generator* and the *Analyzer* in the P4CONSIST architecture. Hereby, Figure 5 illustrates the P4CONSIST workflow with the role of each module. First, the input traffic generator (sender) sends the active probe-based traffic comprising of critical flows (5-tuple flows) intended for a destination from a particular source. Second, every switch pushes the telemetry data to the header stack of the packet in real-time. Third, as the traffic reaches the destination, it samples the packet and extracts the corresponding packet containing path- and rule-encoded in the telemetry data to forward them to the analyzer as an *actual report*. Fourth, the control plane module generates the network graph (reachability graph) using the current topology- and configuration-specific information (*expected report*) stored in the control plane module to send it to the analyzer. Fifth, the analyzer traverses the graph to generate all possible paths between the source and destination. Finally, the analyzer generates a symbolic packet with similar header (5-tuple flow) based on the actual report, and simulates its forwarding through the returned paths. The symbolic execution returns the status of every path, so in order to detect inconsistencies, the analyzer checks the status of the actual report with the results.

In particular, it consists of 4 modules on a high-level with

the following functions:

1) **Control Plane Module:** Provides the current SDN controller information, e.g., topology and configurations as an *expected report* comprising reachability graph for a given packet and a source-destination pair to the analyzer. (§III-C)

2) **Input Traffic Generator:** Generates active traffic pertaining to critical flows (5-tuple flow) for testing the network between a given source-destination pair. (§III-A)

3) **Data Plane Module:** For any given packet header (5-tuple) and a source-destination pair, it encodes the path and sequence of forwarding rules to generate a *sampled actual report*. (§III-B)

4) **Analyzer:** Brain of P4CONSIST. It detects inconsistency by comparing the expected from the control plane module with the actual report/s from the data plane module. (§III-D)

Now, we will take a deep dive into the modules of P4CONSIST in a non-sequential manner for the ease of description.

A. Input Traffic Generator

P4CONSIST has no special requirements for the input traffic, just packets that allow the data plane P4 program to stack the telemetry data. Thus, any state of the art traffic generator [42]–[44] would suffice. With a goal to facilitate the tracking and filtering of the packets, avoid generating an overhead on the links in parallel to the production traffic, and covering all critical flows for a pair of source destination, the input traffic generator (sender) allows manipulating and generating special active probe traffic. Note the rate at which packets are required to be generated is 10^3 pps (packets per second) and therefore, it is $\approx 0.05\%$ of a 1 Gbps link which is minimal link overhead.

In principle, it is possible to generate packets belonging to diverse flows via dynamic program analysis techniques such as fuzzing where bits in a packet are mutated in an exhaustive fashion. Such an approach is illustrated in [45]. Furthermore, in this paper, we are interested in detecting the control-data plane inconsistency for the critical flows which are known and can be generated via the Input traffic generator.

One interesting approach is P4pktgen [11]. However, it is limited to localization of errors in the toolchains, e.g., p4c, used to compile and run a P4 program. It uses symbolic execution to create exemplary packets which can execute a selected path in the program. We note that P4pktgen generates exemplary or symbolic packets and not the real packets generated by Input traffic generator that act as an input for a P4 switch on the data plane to detect control-data plane inconsistency during operation.

B. Data Plane Module

The data plane module in P4CONSIST takes advantage of the Inband-Network Telemetry (INT) [4], [46] to collect the telemetry data in real time and offers the flexibility of modification to mirror the data to be checked in the control plane. P4CONSIST generates the *actual report* from the data plane by collecting the telemetry data of every switch traversed in the path. As the traffic starts flowing through the network, every switch parses the packets and based on the packet header (5-tuple) and source-destination pair runs a lookup for the key on the Match-Action table. The recovered action data from the table plays a double role, first, it allows routing and forwarding the packet, second, before deparsing the packet, it defines the required telemetry data for the path to be pushed (e.g. switch ID, matched rule ID). Thus, the telemetry data encodes the rule- and path-related information for a given packet header (5-tuple-flow) between a given source-destination pair and it gets updated on each hop up to the destination. Since, we used INT [4], [46] with Multi-Hop Route Inspection [32] for the implementation, the details are mentioned in the implementation (§IV-A2).

C. Control Plane Module

In principle, we can use the existing control plane modeling mechanisms, including Anteater [47], HSA [48], NetPlumber [49] and APVerifier [50]. Inspired from [47], we modeled the network (topology and configuration) from the control plane. To generate the `expected report`, the control plane module proactively parses the SDN controller-specific topology and configurations. The control plane module automatically models the network as a graph to determine the end-to-end reachability between any source-destination pair for a given packet. We call this graph as *reachability graph*.

The network graph N is modeled as a 3-tuple $N = (D,E,P)$, where D is the set of switches and networking devices (vertices), E is the set of *directed* edges representing connections between vertices. P is a function defined on E to represent forwarding policies. For each edge (υ, ν), $P(\upsilon, \nu)$ is the policy for all packets (5-tuple flows) passing from υ to ν. P is expressed as a boolean formula over a symbolic packet. A packet can traverse an edge if and only if it satisfies the corresponding Boolean formulas. This Boolean function or predicate represents general policies including forwarding and packet filtering[2]. It ensures matching the fields defined in the rules with those encapsulated within the symbolic packet representative of a 5-tuple flow in a packet from the *actual report* between *any* given source-destination pair. For the edges, P4CONSIST defines the network topology with globally unique links as a combination of the switch and port IDs:

$$\{[swID1 : portID1, swID2 : portID2, .., swIDn : portIDn]\}$$

To elaborate further, in P4, the network configuration files (in the form of `JSON` files) provide all necessary information:

1) A distinct configuration file per switch: facilitates enumerating the vertices of the graph.

[2]In principle, we can model packet rewriting but in our current implementation, we do not support it.

2) Switch ID: the table provides a unique switch ID `swID` for every switch.
3) All Match-Action table rules: including the destination IP address and mask, the ingress port ID and the rule ID.

Once the forwarding rules are parsed, they are saved as a dictionary, where the key is the switch ID, and the value is a list of all rules available in the `JSON` configuration file of the switch in question.

The control plane module is decoupled from the data plane module and, thus, due to modular design of P4CONSIST, these are independent of each other and can be changed independently. Therefore, how the control plane manages rules related to each networking device on the data plane, e.g., P4 switch/NetFPGAs/SmartNICs, is orthogonal to the device on the data plane as it can be adapted accordingly.

Indeed, JSON-based configuration files are something specific to Behavioral-Model (BMv2) switches. However, that is just the implementation choice made by us for implementing P4CONSIST and conducting the experiments. The design of P4CONSIST remains general irrespective of the implementation choice. If the control plane uses a different format for saving and storing the forwarding rules, then we just need to adapt the parser at the control plane module. We parse the rules on the control plane to store them as a dictionary irrespective of a JSON or any other format.

To summarize, the control plane module stores the *expected* switch configuration files and topology-specific information. As soon as the packet (5-tuple flow) is received from the data plane through the analyzer, it generates the corresponding reachability graph pertaining to that packet and source-destination pair, sends it to analyzer which executes graph traversal and symbolic execution to detect control-data plane inconsistency.

D. Analyzer

As a packet from the sampled *actual report* from the data plane reaches the analyzer, it starts by first, extracting the telemetry data comprising of the packet header (5-tuple flow), and path taken by the packet. Based on the packet header, it creates a *symbolic packet* (SP) which contains the header corresponding to a 5-tuple flow. Second, it gets the reachability graph from the control plane between a given source-destination pair for that packet in the form of an *expected report*. Third, the analyzer generates all possible paths between the source and the sink (destination) switches in the reachability graph provided by the *expected* from the control plane module; the *source* is the first switch *source* where the packet first accessed the network, and the *sink* is the last hop or destination in the network. This all possible path generation is carried out via graph traversal technique: Depth First Search (DFS) with cutoff (§III-D1). In this traversal, for a given packet between a given source-destination pair, the path length in the actual report is matched with the path length in the expected report by the use of cutoffs. Finally, the analyzer compares the *actual* path with the *expected* paths from the control and the data plane respectively through the *Symbolic Execution* technique (§III-D2) to check the consistency through path conformity.

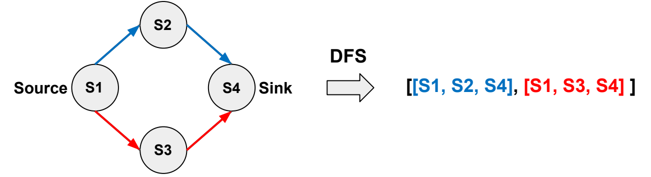


Figure 6: Illustration of graph traversal with the DFS algorithm. It shows that for a given source-destination pair there are two possible *expected* paths: S1-S2-S4 (blue) and S1-S3-S4 (red) from the control plane which will be compared with *actual* data plane path.

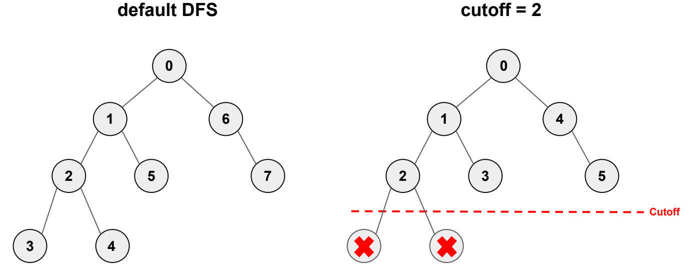


Figure 7: Illustration of a graph traversal through DFS with and without the use of a cutoff. The cutoff approach fixes the path lengths of the expected possible path to the path length of the actual path and thus, filters off unnecessary paths.

In the following, we will explain the graph traversal and symbolic execution in detail.

1) Graph traversal: Depth First Search with cutoff: By definition, a graph traversal is a certain pattern of *visiting* the vertices of a graph [51]. The analyzer implements the graph traversal technique to generate all possible paths between the given source and sink vertices for a given packet header (5-tuple flow). And to simulate the packet forwarding from source to sink nodes through the network, it defines the paths as a `set` of lists of vertices interconnected through edges starting from the source and ending at the sink switches (Figure 6).

$$\{[Path_1], [Path_2], ..., [Path_n]\}$$

The analyzer implements the Depth First Search (DFS) search algorithm for graph or tree traversal [52]. The algorithm starts at the root node and goes continuously down a given branch before backtracking. Note, in the case of a tree-like network topology, for example, the case of fat-tree topology mostly used in the data center networks, DFS may encounter path explosion problem and may require substantial time to generate all paths.

To improve the performance of DFS, we use an algorithm allowing the use of cutoffs. A cutoff is a maximum depth (path-length) that the algorithm can reach within a branch. When the algorithm reaches a cutoff, it breaks and continues with the next branch. The idea is, as a packet is received, the actual path is supposed to be identical to the expected one on the control plane. Thus, it limits the graph traversal to the length of the actual path. Therefore, every time a packet is received from the data plane, the analyzer measures the path length and fixes it as a cutoff for the DFS (Figure 7).

2) Symbolic execution: Once all the *expected* paths between the source and sink (destination) switches are generated, the analyzer conducts the symbolic execution. It simulates

Algorithm 1: Symbolic execution (Consistency checking)

```
Input  : The network configuration Rules, the expected paths between a
         source-destination pair Paths, and the symbolic packet SP.
Output: The consistency status of the actual path Status.
// a path is a list of switches (vertices in the network
   graph)
1  for path ∈ Paths do
2      for switch ∈ path do
           // //check if the switch is the last switch on
              the path.
3          if (switch is the last switch) then
4              for rule ∈ Rules do
                   // check if it is the last rule in the
                      switch
5                  if rule is the last rule then
                       // apply the Boolean policy function
6                      if policy(SP, rule) == True then
                           // path can forward SP
7                          Status(path) ← True
8                          Go to the next path
9                      else
                           // switch and thus, path can not
                              forward the packet
10                         Status(path) ← False
11                         Go to the next path
12                 else
13                     if policy(SP, rule) == True then
                           // path can forward SP
14                         Status(path) ← True
15                         Go to the next path
16                     else
17
18                         Go to the next rule
19         else
20             for rule ∈ Rules do
21                 if rule is the last rule then
                       // apply the Boolean policy function
22                     if policy(SP, rule) == True then
                           // switch can forward SP
23                         Status(switch) ← True
24                         Go to the next switch
25                     else
                           // switch and thus, path can not
                              forward the packet
26                         Status(path) ← False
27                         Go to the next path
28                 else
                       // apply the Boolean policy function
29                     if policy(SP, rule) == True then
                           // switch can forward SP
30                         Status(switch) ← True
31                         Go to the next switch
32                     else
33
34                         Go to the next rule
```

the forwarding of a symbolic packet (SP) belonging to the same flow (5-tuple flow) as the actual packet received from the sampled *actual report* from the data plane through every path in the list of possible paths from the control plane. The consistency checking algorithm (Algorithm 1) applies the Boolean policy function against the symbolic packet and the rules of every switch in the path sequentially (Figure 8). To simulate the behavior of an actual network switch, the symbolic execution has to preserve the first matching order of the rules. In this case, if a rule matches, the switch is marked as forwarding: *True* and the symbolic packet fields are updated.

After, the checking program looks for the next switch and the new switch receives the same checking treatment as the

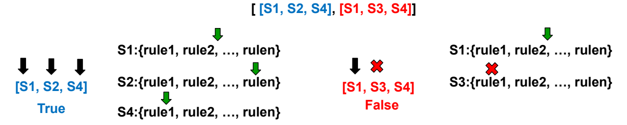


Figure 8: Illustration of the forwarding of the symbolic packet through all possible paths using symbolic execution. Blue shows the correct or consistent path, i.e., S1-S2-S4 and red shows the incorrect or inconsistent path, i.e., S1-S3-S4 respectively.

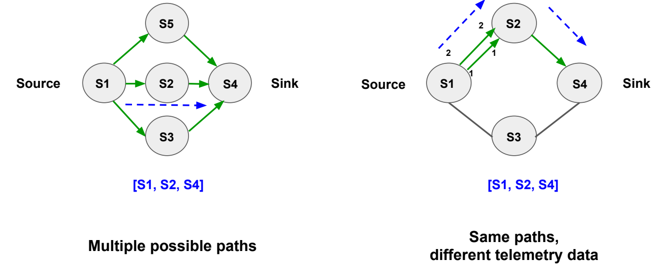


Figure 9: Illustration of multipath scenarios showing the possible cases detectable by the analyzer symbolic execution: multiple possible paths (left) and same paths (i.e., switches) but different ports (right) and thus, different telemetry data for a given packet and a source-destination pair. The blue dashed arrows show the actual path from the actual report.

previous one but with the new updated version of the symbolic packet (new ingress port). If the checking of the rules fail and none of the rules forward the packet, then checking breaks, the current switch is marked as not forwarding: *False* and accordingly the whole path is marked as *False*.

Figure 9 illustrates the multipath scenarios where for a given packet and a source-destination pair, the control plane returns multiple paths for a given packet and a source-destination pair. We show and later evaluate (§IV) that even in the complex case of multipaths (left in Figure 9) or a scenario of same paths in terms of switches but different inports of such switches (right in Figure 9), the symbolic execution can determine the inconsistency as it matches not just `switch IDs` but also the corresponding `inport IDs` and `rule IDs` for any path.

Note blackholes [53] for critical flows can be detected as the analyzer generates an alarm after a chosen time of some seconds if it does not receive any packet pertaining to that flow[3]. For localizing silent random packet drops, MAX-COVERAGE [54] algorithm can be implemented on the analyzer.

Overall, the symbolic execution evaluates all the *expected* paths from the control plane by matching them with the *actual* path from the data plane. The path returning `True` is consistent while the path returning `False` is inconsistent.

IV. Prototype & Evaluation

A. Prototype

This section presents the tools used for the implementation of various modules in the P4Consist prototype. The input traffic generator uses Scapy [55] to forge the packets, the

[3]Blackholes for non-critical flows can be detected and localized through polling of the switches.

```
###[ Ethernet ]###
  dst       = 00:00:00:00:08:10
  src       = 00:00:00:00:08:11
  type      = 0x8100
###[ 802.1Q ]###
            prio      = 0
            id        = 0
            vlan      = 10
            type      = 0x800
###[ IP ]###
            version   = 4
            ihl       = 6
            tos       = 0x0
            len       = 35
            id        = 1
            flags     =
            frag      = 0
            ttl       = 64
            proto     = udp
            chksum    = 0xd235
            src       = 172.16.8.101
            dst       = 172.16.40.10
            \options   \
            |###[ MRI ]###
            |  copy_flag = 0
            |  optclass  = control
            |  option= 31
            |  length  = 4
            |  count   = 0
            |  \swtraces  \
###[ UDP ]###
            sport     = 1234
            dport     = 4321
            len       = 11
            chksum    = 0xf129
###[ Raw ]###
            load      = 'Ok!'
```

Figure 10: Example of an input packet.

data plane module comprising of P4-16 BMv2 (Behavioral-Model version 2) software switches [33] and uses MRI (Multi-Hop Route Inspection) [32] which is a specific version of INT [4] for encoding telemetry data on the input packets, and the analyzer uses Python library NetworkX [56] for graph modeling. Note all of the P4CONSIST modules run in Vagrant [57] development environment.

1) Input Traffic Generator Implementation: Currently, P4CONSIST generates IPv4 based packets, since the MRI [32] P4 program requires the `IPOption` header field to stack the telemetry data. But to facilitate the tracking and filtering of the packets, and avoid generating an overhead, we forge a special UDP active traffic covering all the custom MRI P4 program requirements including: UDP source port 1234 and UDP destination port 4321, IPOption field type 31 and 802.1q header for VLAN tagging. Figure 10 illustrates the input packet header generated via input traffic generator.

Scapy [55] is a Python powerful interactive packet manipulation program largely used among the network community. It enables the user to forge, send, sniff and dissect network packets. By default, it supports a wide number of protocols, which allows construction of tools that can probe, scan or attack networks. We used Scapy to forge the active traffic sent from the sender to the receiver.

2) Data Plane Module Implementation: This section briefly reviews In-band network telemetry (INT), a P4 data plane monitoring tool, in addition to its scaled-down version called Multi-Hop Route Inspection (MRI), and Behavioral Model Version 2 (BMv2) used in the implementation of the data plane module.

INT: In-band Network Telemetry (INT) [4] is a monitoring technique conceived to allow the data plane to collect and report the network state. It introduces header fields in the packets interpreted as “telemetry instructions” by network devices. When a packet with INT instructions is received, the INT-capable device collects the required data and writes it into the packet as it transits the device (Figure 11). The instructions can be embedded in normal data packets or in special probe packets.

MRI: Multi-Hop Route Inspection (MRI) is a scaled-down version of INT introduced by the P4 community [32]. As the packet is forwarded through the network, it collects `switch IDs` and `queue depth` of every switch hop, allowing the users to get an overview of the path and the occupancy of the queues. MRI is implemented as a P4 program, which collects the data and appends it to the header stack of every packet. It is implemented based on a basic L3-forwarding P4 program, extended with an extra MRI header. The default MRI header is constructed with two 32-bit fields (`switch ID, queue depth`) gathered from every switch hop in the path. These headers are carried inside the IP Options header. To indicate the presence of the MRI headers, a special IP Option “type” with the value 31 was defined.

To offer a more rigorous verification, P4CONSIST requires more specific data related to the control plane configuration in the SDN context (e.g. `ingress port ID` and matched `rule ID`). Accordingly, we implement a custom MRI code where we replace the `queue depth` by the `ingress port ID` included in the default P4 metadata. As a result, we offer the possibility to use ingress port-based routing, where the switch is capable of taking routing decisions not only based on the destination IPv4 Address but also the ingress port on which the packet was received. In addition, the ability to check the consistency of this information, when collected by the MRI program.

For rule checking, i.e., if the *actual* matched rule from the switch Match-Action table on the data plane is the same as the *expected* match from the control plane (i.e., the switch configuration). Unfortunately, the design of the P4 language does not allow the P4 code to manipulate the Match-Action table and consequently get access to the `rule ID`. Only the controller has access to this information. P4CONSIST stores this information as an action in the Match-Action table. One can manually define a `rule ID` for every entry in the Match-Action table and the P4 program when the rule is matched will parse the action parameters including `destination MAC address`, `egress port ID` and `rule ID`.

Finally, P4CONSIST design allows verifying packets from different sender hosts by extracting the path between a source-destination pair or pairs and running the verification process. To include the case of multiple senders, sharing the same ingress port on the same source switch when sending to the same destination respectively, we tag the packets in a way that the receiver can uniquely distinguish the senders. To reduce the amount of data pushed inside the MRI header, we opt for implementing the VLAN tagging. It allows adding a 32-bit field between the source MAC address and the `EtherType`

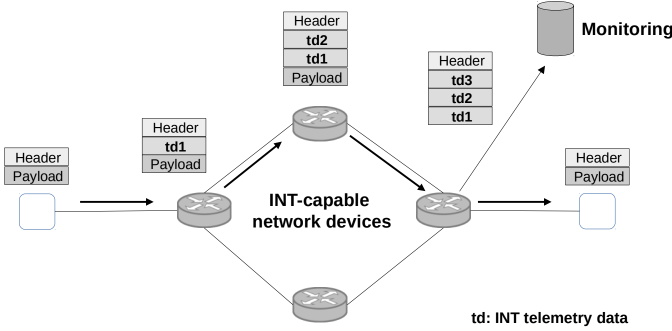


Figure 11: Collection of telemetry data at each switch by INT.

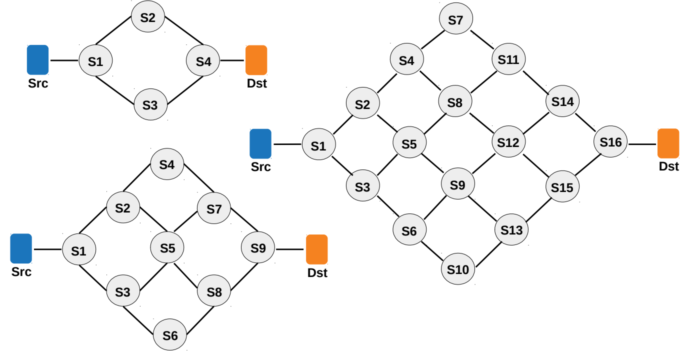


Figure 12: Illustration of 4, 9 and 16 switches grid experimental topologies.

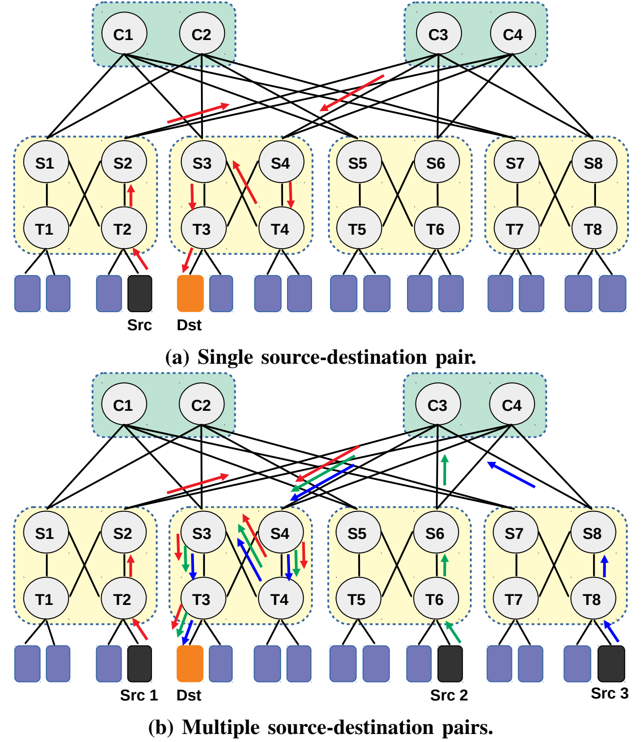
fields of the original frame in the P4 program. It is identified by the `EtherType` value `0x8100`.

BMv2: The Behavioral-Model version 2 (BMv2) [33] is a P4 software switch introduced by the P4 consortium for developers, to facilitate implementing their own P4-programmable architecture. From the available BMv2 P4 software switch targets, the `simple_switch_grpc` is used as it is the only target that implements the official P4 Runtime API, but also supports the default thrift API channel, as a side channel for debugging [38]. The debugging channel is accessible through the command: `runtime_CLI`. Overall, using the `simple_switch_grpc` for the experiments fulfills all the pre-requisites for introducing the data plane faults.

3) Control Plane Module & Analyzer Implementation: For the implementation of the control plane module and analyzer, we used the Python language. For the graph generation, we chose NetworkX [56]: a Python library for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It introduces a standard and suitable programming interface and graph implementation, and enables loading and storing networks in standard and non-standard data formats, generating random and classic networks, analyzing network structure, building network models, designing new network algorithms.

B. Experimental Setup

We evaluate P4CONSIST on 3 grid topologies of 4, 9 and 16 switches (Figure 12), and a datacenter 4-ary fat-tree with 20 switches (Figure 13). The Experiments were conducted with 15k, 30k and 60k rules on each switch, and sFlow [58] sampling rate of 1/100. *Each experiment was conducted* 10 *times*. The CPU machine is an 8 core 2.4GHz Intel-Xeon CPU machine and 64GB of RAM. The controller and the switches run inside vagrant machines configured with 1 CPU core and 1GB of RAM. The receiver hosts are provisioned with 2 CPU cores and 2GB of RAM. As the sender hosts have no special requirements, 1 CPU core and 512MB of RAM is allocated. To automate the active traffic generation, we create pcap files for every source-destination pair, which will be replayed in a loop onto the network using Tcpreplay [59] with a throughput of 10^3 pps (packets per second). For the custom P4 Runtime configuration JSON files, we use a custom script to generate the required number of rules for every scenario. The rules are generated in a way that the P4 program will keep checking



(a) Single source-destination pair.

(b) Multiple source-destination pairs.

Figure 13: The datacenter 4-ary fat-tree experimental topology with single (13a) and multiple (13b) source-destination pairs.

them sequentially till it reaches the rules in question, and the controller is able to populate them in the Match-Action table without errors.

Bug Injection: We randomly inject 20 bugs to be detected by P4CONSIST in every experiment. The bugs are distributed randomly through different hops of the path and cover different rule modifications scenarios, such as different paths or same path but with every time different VLAN IDs, Rule IDs or ingress port IDs.

To automate the triggering of the bugs, we implement custom bash scripts, which while the traffic is flowing through the network will modify the rules in the switches and save the error triggering timestamp in a file. Finally, using a custom Python script, we measure the time-difference between the triggering of the bug and the detection of the same bug. Note each experiment was conducted 10 times.

C. Evaluation Strategy

To evaluate P4CONSIST, our action plan is to parametrize our experiments with different network sizes ranging from 4 to 20 switches allowing different path numbers and maximum path lengths, as the configuration varies from 15k to 60k rules per switch. We aim to evaluate the performance gain of the the depth first search with cutoff during the graph traversal and the symbolic execution time for any given packet. Thus, our metrics of interest are the detection time of the bugs for single and multiple source-destination pairs, the duration of the symbolic execution to check the received packet, the graph traversal time with and without cutoff, and the dataplane overhead.

In particular, we ask the following questions:

Q1: *How much time does* P4CONSIST *take to detect control-data plane inconsistencies between a single source-destination pair?* (§IV-C1)

Q2: *How much time does* P4CONSIST *take to detect control-data plane inconsistencies between multiple source-destination pairs?* (§IV-C2)

Q3: *How much time does* P4CONSIST *take to compute all possible paths at the analyzer and how many paths are available with and without cutoff?* (§IV-C3)

Q4: *How much time does* P4CONSIST *take to execute symbolic execution at the analyzer over different topologies and configurations of varying scale and complexity?* (§IV-C4)

Q5: *How much dataplane overhead does* P4CONSIST *incur?* (§IV-C5)

1) Inconsistency detection time — single source-destination pair: This section discusses the first question (***Q1***) and evaluates the capacity of P4CONSIST to detect errors. To answer this question, we run the experiments with the different topologies and configuration mentioned in §IV-B and measure the time between the bug was triggered and the time when it was detected.

In Figure 14, we plot the cumulative distribution function (CDF) of the bugs detected in the experiments. As expected we were able to detect all 20 bugs for all experiments conditions. The plots illustrate that for the different configurations the detection finishes first in the 4 switches topology, then 9 switches and finally 16 switches. We observe that for 4 and 9 switches 75% of the bugs were detected in a maximum of 4 minutes for all kinds of configuration scales. For 16 switches, considering the number of possible paths and the number of rules, P4CONSIST is able to detect 75% of the bugs were detected in a minimum of 3 minutes (for 30k rules per switch) and a maximum of 13 minutes (for 60k rules per switch). For the 4-ary fat-tree topology (Figure 13a), P4CONSIST performed as expected detecting 75% of the bugs in a minimum of 3 minutes (for 15k rules per switch) and a maximum of 17 minutes (for 60k rules per switch). As the experiment was conducted ten times, the time taken is the mean of the ten values to detect an inconsistency. We omitted confidence intervals as they are small after 10 runs. In all cases, the detection time difference was marginal.

2) Inconsistency detection time — multiple source-destination pair: This section discusses the second question (***Q2***) and evaluates the detection performance and scalability of P4CONSIST for multiple source-destination pairs as compared to single source-destination pair. Hereby, we run the previous experiment for the fat-tree topology with three different sources sending traffic simultaneously as illustrated in Figure 13b. Therefore, we have 3 source-destination pairs. In Figure 15, we plot the CDF of the detection time for a single (Figure 13a) and multiple (Figure 13b) source-destination pairs in 4-ary fat-tree topologies. We expect to detect 20 bugs for single and 60 bugs (20 bugs per source-destination pair) for the multiple source-destination pair case. While the single-source destination pair performed as expected detecting 75% of the bugs in a minimum of 3 minutes (for 15k rules per switch) and a maximum of 17 minutes (for 60k rules per switch), we observe that the analyzer was able to cover the 15k and 30k rules per switch with 3 source-destination pairs, with a minor performance degradation to detect all 60 bugs in $\approx$ 1 hour. In the experiment of 60k rules per switch and 3 source-destination pairs, 50% of the errors detected in almost 30 minutes with a minor performance degradation to detect all 60 bugs in $\approx$ 3 hours. As the experiment was conducted ten times, the time taken is the mean of the ten values to detect an inconsistency. We omitted confidence intervals as they are small after 10 runs. In all cases, the detection time difference was marginal. Therefore, this shows that P4CONSIST scales to more than one source-destination pair. However, since, the traffic is sent from multiple ingress points in parallel so in case of ATPG-like [20] all-pairs analysis, we run out of memory on our Analyzer VM as there is an exponential number of flows to be analyzed at the same time.

Topology	source & destination switches	Number of paths	
		Cutoff	No-cutoff
4 switches	S1-S4 (Figure 12)	2	2
9 switches	S1-S9 (Figure 12)	6	12
16 switches	S1-S16 (Figure 12)	20	184
4-ary fat-tree	T2-T3 (Figure 13a)	20	1360

Table I: This table illustrates the number of possible paths for every topology (Column 1), for every source-destination pair within the topology (Column 2), and if generated with or without the use of a cutoff (Columns 3-4). We observe that as the topologies grow in scale, the cutoff helps to reduce the number of paths and hence, the detection process considerably to improve the overall inconsistency detection time of P4CONSIST.

3) Graph traversal time: This section discusses the third question (***Q3***) about the amount of time that P4CONSIST takes to compute all possible paths and the effect of using cutoff on the number of available paths.

In Figure 16, we measure the graph traversal time with and without using cutoff, between a fixed source and destination switches as explained in the Table I, which also shows the number of possible paths in both scenarios.

Even if the graph traversal takes between 0.1 and 0.7 microseconds for the worst case, we still have a clear difference between using the cutoff and the default algorithm without cutoff. The graph traversal takes less time with cutoff, although this may not directly affect the duration of the global verification process with few microseconds in non-scalable settings, but if we consider that in the case of 16 switches and 4-ary fat-tree, we go down from checking 184

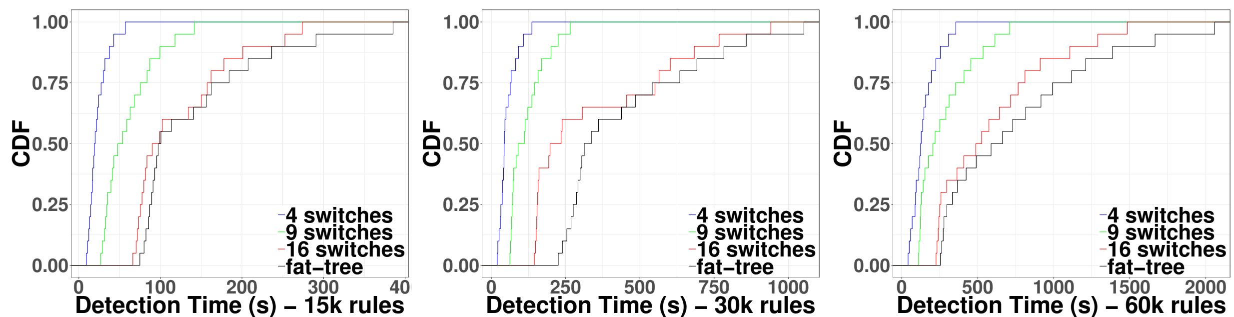


Figure 14: Detection time CDF of 20 bugs measured for the 4 experimental topologies and with 3 different configuration scale. The detection time represents an average over 10 runs.

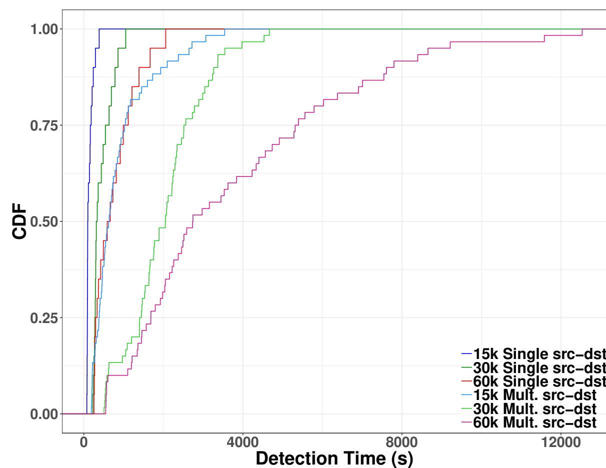


Figure 15: Detection Time CDF for the 4-ary fattree topology with a single and multiple source-destination pairs for different configurations. The detection time represents an average over 10 runs.

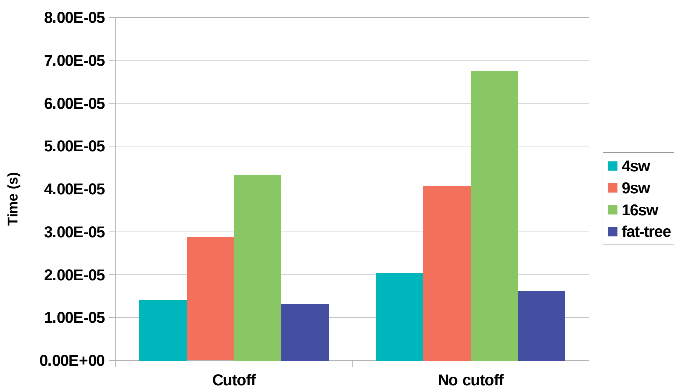


Figure 16: The graph traversal time measured for the 4 experimental topologies with and without cutoff. Each bar represents an average over 10 runs.

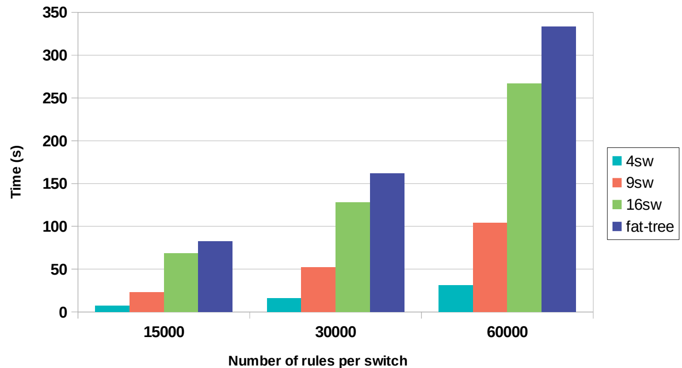


Figure 17: The symbolic execution time measured for the 4 experimental topologies. Each bar represents an average over 10 runs.

and 1360 possible paths respectively to only 20 paths, this will without any doubt reduce the number of checks to be run and consequently make the verification process faster.

Furthermore, we observe that even for the same number of paths in the case of 4 switches, the algorithm performs well because it exits early when it reaches the maximum depth. In addition, because of the tree nature of the fat-tree topology, and despite the size of the topology, the DFS was more effective. Note we omitted confidence intervals as they are small after 10 runs. In all cases, the time difference was marginal.

4) Symbolic execution time: In this section, we discuss the fourth question (***Q4***) about the performance of symbolic execution under different experimental topologies (Figure 12, 13) and configurations of varying scale and complexity.

Figure 17 shows the duration of the symbolic execution of symbolic packet forwarding through the network nodes for all possible paths. For 4 switches and only 2 possible paths, P4CONSIST is able to keep the verification process around 30 seconds for 60k rules/switch. The checking time increases with the number of rules. For 9 and 16 switches (6 and 20 possible paths respectively using cutoff), the time kept increasing with the number of rules as expected and takes less than 5 minutes for the max. number of rules and paths. For the case of the fat-tree topology, although we have the same number of possible paths (20 paths), the checking takes slightly more time than the 16 switches experiment because of the total number of switches in the topology (20 switches). Note we omitted confidence intervals as they are small after 10 runs. In all cases, the time difference was marginal.

5) Dataplane overhead: To answer the fifth question (***Q5***), we realize that the INT packets are strictly processed within the fast path. The information from the data plane is extracted and exported directly without the overhead or scale limitations of a control plane CPU. In addition, the packets are generated at 10^3 pps (packets per second) and therefore, it is $\approx 0.05\%$ of a 1 Gbps link which is a minimal link overhead. Thus, there is no noticeable overhead on the data plane.

V. LIMITATIONS & FUTURE WORK

P4CONSIST opens new avenues for the future work and one of them is the localization of detected faults. Currently, we do not handle transient faults, checking of backup rules, and TCP-based packets which we plan to address in the future. Furthermore, development of an efficient fuzz testing mechanism which generates packets pertaining to different header space by mutation of packet bits belongs to the future work. Lastly, improving multi-threading to cope with the large number of flows efficiently and detection of performance-related faults like throughput, latency and packet-loss lays the groundwork for our future work.

VI. RELATED WORK

We, now will navigate the landscape of related work and compare it to P4CONSIST in terms of the faults which cause inconsistency during runtime (§II-B).

Consistency-based verification tools: [60]–[62] verify the consistency of the control-data plane in SDNs. These tools, however, need sufficient customizations or redesigning to be applicable in the context of P4 SDNs. Such control-data plane consistency is verified continuously or periodically by P4CONSIST. Recently, [45] has proposed a fuzzing methodology to verify the control-data plane inconsistency in OpenFlow-based SDNs. This approach, however, is OpenFlow- and OpenvSwitch-specific and in addition is prone to false positives due to bloom filters used in tagging whereas P4CONSIST uses symbolic execution and DFS with cutoffs to detect inconsistencies in the P4 SDNs. In addition, one recent work [63] has motivated the control-data plane inconsistency in P4-enabled SmartNICs [64], however, it takes into account performance-based properties (i.e., latency, throughput) and not functional properties (i.e., path or rule correctness) like P4CONSIST.

Control- or data plane-based testing tools: The related work in the area of control plane [31], [48]–[50], [65]–[72] 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. The data plane tools either test the rules or the paths whereas P4CONSIST tests both together. In the case of faults where path gets affected, i.e., when the path is same even if different rules are matched, path trajectory tools [61], [73]–[78] fail. The approaches based on active-probing [20], [22], [41], [77], [79]–[81] do not detect the faults caused by hidden or misconfigured rules on the data plane which may only match the active traffic. These tools, however, only generate the probes to test the rules known or synced to the controller. Such faults are detected continuously or periodically by P4CONSIST.

P4-based verification tools: Recently in the area of P4-related verification, a plethora of tools [7]–[12], [31] have surfaced. They, however, majorly debug only P4 programs using static program analysis techniques like symbolic execution or Hoare logic. Runtime bugs and inconsistency which happen once the P4 program is compiled to run on the P4 switch and subjected to input traffic cannot be detected by these tools. Recently, P4RL [13] has performed runtime verification of a single P4 switch via reinforcement learning guided fuzzing. However, it applies to only a single P4 switch and not P4 SDN. In addition, it just can detect the control-data plane inconsistency only in the case of path deviation and not rule deviation. P4CONSIST continuously or periodically detects such runtime faults causing inconsistency.

Related work	Context	Runtime Verification	C-D inconsistency detection
PAZZ [45]	SDN	✓	✓
VeriDP [61]	SDN	✓	(✓)[4]
Cocoon [12]	P4	×	×
Vera [8]	P4	×	×
p4v [7]	P4	×	×
P4-ASSERT [9], [10]	P4	×	×
P4NOD [82]	P4	×	×
P4pktgen [11]	P4	×	×
P4RL [13]	P4	✓	(✓)[5]
P4CONSIST	✓	✓	✓

Table II: Relevant related work in SDN/P4 verification. Note, ✓denotes the capability, (✓) denotes a part of full capability, and × denotes the missing capability. C-D denotes control-data plane.

VII. DISCUSSION

P4 programs direct the packet processing pipeline but not the exact ruleset offered by the control plane that determines the forwarding behavior. Without having a deep understanding of control-data plane interactions, we cannot determine the exact data plane behavior. P4Runtime is one positive effort in standardizing the control-data plane interactions to make the data plane behavior more predictable by removing the shortcomings of OpenFlow [83] and Switch Abstraction Interface (SAI) [84]. It allows the runtime-control of the arbitrary forwarding planes by defining open, standard and silicon-independent API. We realize, however, there is a big space for improvement in verification techniques when it comes to control-data plane interactions as the controller is an independent software from the P4 program and how these two independent programs interplay to enforce a common high-level policy gets tricky as inconsistencies arise. To summarize, the verification of network inconsistencies has become increasingly indispensable.

Now, two important questions come to our mind: *"How frequently to check the inconsistencies?"* and *"how often to check the inconsistencies?"*. To answer the former, we need a P4CONSIST-like continuous testing mechanism as control and data plane programs may evolve over time. Unquestionably, the temporal dimension aggravates the challenges to verify the inconsistencies. To answer the latter, combinatorial complexity caused by path explosion even in simple topologies and configurations may contribute to exponential delays in detecting the inconsistencies. Thus, we need strong semantic foundations while designing the verification methodologies.

Furthermore, detecting inconsistencies is not just about resources to invest but about test traffic as well. Active probing

[4]Detects only path related inconsistency in SDNs.

[5]Can detect only path related inconsistency in a single P4 switch and not a P4 SDN.

is a useful tool in fault detection, however, it has some inherent drawbacks. For instance, one needs to make sure that active traffic depicts a true picture of the network and does not receive any differential treatment as compared to the original production traffic. In addition, the active traffic should not overwhelm or congest the network with multiple probes and thus, leaving no resources for the production or passive traffic. We, however, realise that greedily solving the minimization of test packets, a well-known NP-complete problem for general graphs, obtains sub optimal results.

Lastly, we understand that the abstraction of programmability is not just a harbinger of new capabilities and flexibilities but also new challenges. While the panoply of benefits gained from such flexibility and customizability can improve (cost-)efficiency and unleash tremendous innovation potentials, the challenges get more complex. The networks are increasingly becoming a cocktail of vendor-specific, open source, in-house libraries and thus, exacerbating the challenges to verify the networks. In order to deal with such a dynamic ecosystem, we require a constant reassessment and redesign of the existing network verification methodologies.

VIII. Conclusion

This paper presented P4Consist, a novel network verification system that aims to detect the control-data plane inconsistency in P4 SDNs in an automated manner. In P4Consist, the actual report from the data plane and expected report from the control plane are compared to verify control-data plane consistency. Our evaluation of P4Consist over various network topologies and configurations of different scale showed that P4Consist efficiently detects the faults causing inconsistency while requiring minimal data plane resources (e.g., link bandwidth and switch rules).

IX. Acknowledgement

We thank Anja Feldmann for the initial discussions. We also 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 and also supported by the WWTF project WHATIF (ICT19-045).

References

[1] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[3] P. L. Consortium, "P4 language and related specifications," https://p4.org/specs/, accessed: 2019-03.

[4] 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.

[5] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 35–49.

[6] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 121–136.

[7] J. Liu *et al.*, "P4v: Practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018, pp. 490–503.

[8] R. Stoenescu *et al.*, "Debugging p4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018, pp. 518–532.

[9] L. Freire *et al.*, "Uncovering bugs in p4 programs with assertion-based verification," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: ACM, 2018, pp. 4:1–4:7.

[10] M. Neves *et al.*, "Verification of p4 programs in feasible time using assertions," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: ACM, 2018, pp. 73–85.

[11] A. Nötzli *et al.*, "P4pktgen: Automated test case generation for p4 programs," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: ACM, 2018, pp. 5:1–5:7.

[12] L. Ryzhyk, N. Bjørner, M. Canini, J. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese, "Correct by construction networks using stepwise refinement," in *USENIX NSDI*, 2017.

[13] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid, "Runtime Verification of P4 Switches with Reinforcement Learning," in *ACM SIGCOMM NetAI*, 2019.

[14] "Microsoft: misconfigured network device led to azure outage," http://www.datacenterdynamics.com/content-tracks/servers-storage/microsoft-misconfigured-\\network-device-led-to-azure-outage/68312.fullarticle/.

[15] "France seeks influence on telcos after outage," https://theneteconomy.wordpress.com/2012/07/11/france-seeks-influence-on-telcos-after-outage/.

[16] "Cloud leak: Wsj parent company dow jones exposed customer data," https://www.upguard.com/breaches/cloud-leak-dow-jones/.

[17] "Con-ed steals the 'net'," http://dyn.com/blog/coned-steals-the-net/.

[18] D. Madory, "Renesys blog: Large outage in pakistan," *Blog*, 2011.

[19] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "A Survey Of Network Troubleshooting," Stanford University, Tech. Rep. TR12-HPNG-061012, 2012. [Online]. Available: http://yuba.stanford.edu/~peyman/docs/atpg-survey.pdf

[20] ——, "Automatic test packet generation," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 241–252.

[21] P. Gawkowski and J. Sosnowski, "Experiences with software implemented fault injection," in *Architecture of Computing Systems (ARCS)*, 2007.

[22] P. Perešíni, M. Kuźniar, and D. Kostić, "Monocle: Dynamic, Fine-Grained Data Plane Monitoring," in *Proc. ACM CoNEXT*, 2015.

[23] "The anatomy of a leak: As 9121," https://www.nanog.org/meetings/nanog34/presentations/underwood.pdf.

[24] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding bgp misconfiguration," in *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 4. ACM, 2002, pp. 3–16.

[25] "Cloud leak: How a verizon partner exposed millions of customer accounts," https://www.upguard.com/breaches/verizon-cloud-leak?utm_campaign=Verizon%20Cloud%20Leak&utm_content=57437217&utm_medium=social&utm_source=twitter/.

[26] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *Proc. PAM*, 2015.

[27] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proceedings of the Symposium on SDN Research*. ACM, 2016, p. 6.

[28] J. Rexford, "SDN Applications," in *Dagstuhl Seminar 15071*, 2015.

[29] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A SOFT Way for Openflow Switch Interoperability Testing," in *Proc. ACM CoNEXT*, 2012.

[30] M. Kuzniar, P. Peresini, and D. Kostić, "Providing reliable fib update acknowledgments in sdn," in *Proc. ACM CoNEXT*, 2014, pp. 415–422.

[31] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking Beliefs in Dynamic Networks," in *Proc. USENIX NSDI*, 2015.

[32] P. A. working group, "Mri tutorial," https://github.com/p4lang/tutorials/tree/master/exercises/mri, P4 Language Consortium, June 2018, accessed: 2019-03.

[33] P4.org, "Behavioral model repository," https://github.com/p4lang/behavioral-model, P4 Language Consortium, October 2015, accessed: 2019-03.

[34] P. A. working group, "Initial draft specification for p4 runtime," https://p4.org/p4-runtime/, P4 Language Consortium, October 2017.

[35] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[36] P4 Language Community, "p4c," 2019. [Online]. Available: https://github.com/p4lang/p4c

[37] L. Vicisano and A. Bas, "Announcing p4runtime - a contribution by the p4 api working group," https://p4.org/api/announcing-p4runtime-a-contribution-by-the-p4-api-working-group.html, P4 Language Consortium, July 2017.

[38] P4.org, "Behavioral model targets," https://github.com/p4lang/behavioral-model/blob/master/targets/README.md, P4 Language Consortium, April 2016, accessed: 2019-03.

[39] B. Networks. (2019) Tofino. https://www.barefootnetworks.com/products/brief-tofino.

[40] OpenFlow Spec, "https://www.opennetworking.org/images//openflow-switch-v1.5.1.pdf," p. 51, 2015.

[41] P. Zhang, C. Zhang, and C. Hu, "Fast testing network data plane with rulechecker," in *Network Protocols (ICNP), 2017 IEEE 25th International Conference on*. IEEE, 2017, pp. 1–10.

[42] J. Sommers, H. Kim, P. Barford, and P. Barford, "Harpoon: a flow-level traffic generator for router and network tests," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1. ACM, 2004, pp. 392–392.

[43] N. Bonelli, S. Giordano, G. Procissi, and R. Secchi, "Brute: A high performance and extensible traffic generator," in *Proc. of SPECTS*, 2005, pp. 839–845.

[44] G. Antichi, A. Di Pietro, D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Bruno: A high performance traffic generator for network processor," in *2008 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*. IEEE, 2008, pp. 526–533.

[45] A. Shukla, S. J. Saidi, S. Schmid, M. Canini, T. Zinner, and A. Feldmann, "Towards Consistent SDNs: A Case for Network State Fuzzing," in *IEEE Transactions on Network and Service Management*, 2019.

[46] P4.org, "In-band network telemetry (int) specification v0.5 and 1.0," https://github.com/p4lang/p4-applications/blob/master/docs, P4 Language Consortium, April 2018, accessed: 2019-03.

[47] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 290–301.

[48] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks," in *Proc. USENIX NSDI*, 2012.

[49] P. Kazemian *et al.*, "Real Time Network Policy Checking Using Header Space Analysis," in *Proc. USENIX NSDI*, 2013.

[50] H. Yang and S. S. Lam, "Real-Time Verification of Network Properties Using Atomic Predicates," *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 887–900, April 2016.

[51] T.-Y. Cheung, "Graph traversal techniques and the maximum flow problem in distributed computation," *IEEE Transactions on Software Engineering*, no. 4, pp. 504–512, 1983.

[52] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[53] Y. Zhu *et al.*, "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 479–491.

[54] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren, "Detection and localization of network black holes," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*. IEEE, 2007, pp. 2180–2188.

[55] P. Biondi and the Scapy community, "Scapy-documentation," https://scapy.readthedocs.io/en/latest/, accessed: 2019-03.

[56] A. Hagberg, D. Schult, and P. Swart, "Networkx reference release 2.2," 2018.

[57] HashiCorp, "Introduction to Vagrant," https://www.vagrantup.com/intro/index.html, accessed: 2019-03.

[58] S. Panchen, P. Phaal, and N. McKee, "Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks," 2001.

[59] A. Turner and M. Bing, "Tcpreplay: Pcap editing and replay tools for* nix," *online], http://tcpreplay. sourceforge. net*, 2005.

[60] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks." in *NDSS*, vol. 15, 2015, pp. 8–11.

[61] P. Zhang *et al.*, "Mind the gap: Monitoring the control-data plane consistency in software defined networks," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 2016, pp. 19–33.

[62] A. Shaghaghi, M. A. Kaafar, and S. Jha, "Wedgetail: An intrusion prevention system for the data plane of software defined networks," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 849–861.

[63] N. Gray, A. Grigorjew, T. Hosssfeld, A. Shukla, and T. Zinner, "Highlighting the gap between expected and actual behavior in p4-enabled networks," *IFIP/IEEE International Symposium on Integrated Network Management Demos*, 2019.

[64] "Netronome. open vswitch offload and acceleration with agilio cx smartnics," https://www.netronome.com/media/redactor_files/WP_OVS_Benchmarking.pdf.

[65] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "Buzz: Testing context-dependent policies in stateful networks," in *Proc. USENIX NSDI*, 2016, pp. 275–289.

[66] S. K. Fayaz and V. Sekar, "Testing stateful and dynamic data planes with flowtest," in *Proc. SIGCOMM Workshop HotSDN*. ACM, 2014, pp. 79–84.

[67] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the Data Plane with Anteater," in *SIGCOMM*, 2011.

[68] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 15–27.

[69] 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 *NSDI*, vol. 14, 2014.

[70] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE Way to Test Openflow Applications," in *Proc. USENIX NSDI*, 2012.

[71] C. Scott *et al.*, "Troubleshooting blackbox sdn control software with minimal causal sequences," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 395–406, 2015.

[72] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. D. Millstein, "A general approach to network configuration analysis." pp. 469–483, 2015.

[73] N. Handigol *et al.*, "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks," *Proc. USENIX NSDI*, pp. 71–85, 2014.

[74] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker, "Compiling Path Queries," in *Proc. USENIX NSDI*, 2016.

[75] P. Tammana *et al.*, "CherryPick: Tracing Packet Trajectory in Software-Defined Datacenter Networks," in *SOSR*, 2015.

[76] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 233–248.

[77] K. Agarwal *et al.*, "SDN traceroute : Tracing SDN Forwarding without Changing Network Behavior," *HotSDN 2014*, pp. 145–150, 2014.

[78] P. Zhang, S. Xu, Z. Yang, H. Li, Q. Li, H. Wang, and C. Hu, "Foces: Detecting forwarding anomalies in software defined networks," 2018.

[79] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect sdn forwarding with rulescope," pp. 1–9, 2016.

[80] Y.-M. Ke, H.-C. Hsiao, and T. H.-J. Kim, "Sdnprobe: Lightweight fault localization in the error-prone environment," 2018.

[81] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: Using packets for low latency network programming and visibility," in *ACM SIGCOMM Computer Communication Review*, vol. 44. ACM, 2014, pp. 3–14.

[82] N. Lopes, N. Bjørner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese, "Automatically verifying reachability and well-formedness in p4 networks," *Technical Report, Tech. Rep*, 2016.

[83] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[84] "Switch abstraction interface," https://github.com/opencomputeproject/SAI.