

P4 Switch Code Data Flow Analysis: Towards Stronger Verification of Forwarding Plane Software

Karine Birnfeld, Diogo Campos da Silva, Weverton Cordeiro
Institute of Informatics (INF)
Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
{first.last}@inf.ufrgs.br

Breno Bernard Nicolau de França
Institute of Computing (IC)
University of Campinas (UNICAMP)
Campinas, Brazil
breno@ic.unicamp.br

Abstract—With the advent of languages like POF and P4 and the ability to write code for redefining switch behavior, software verification becomes paramount to avoid network and service disruption due to buggy implementations. Various techniques like symbolic execution, annotations, and assertions have been recently used to ensure bug-free switch code. In spite of the potentialities, they are limited in the classes of errors they can capture. More importantly, existing proposals for switch program verification often require a programmer to write custom verification code (e.g., annotations), an error-prone approach in itself. In this paper, we present the design and implementation of p4-data-flow, a practical tool which uses data flow analysis for verification of switch programs. We focus on the P4 language, and present experiments showing that data flow analysis may reveal defects from classes not yet covered by existing work, without demanding further programmer effort.

Index Terms—P4, programmable network, programmable data plane, software testing, data flow testing

I. INTRODUCTION

The concept of programmable forwarding planes has experienced a renewed research interest since the advent of Software Defined Networking (SDN) [1], [2]. Domain-specific languages like POF [3] and P4 [4] now allow network operators to redefine how forwarding devices parse and process packets, thus enabling faster provisioning of novel and/or home-brewed protocols, and unleashing innovation in the forwarding plane.

In a world where network operators can redefine switch behavior, by writing their code to implement some protocol specification, proper verification and validation (V&V) of written switch code becomes critical for adequate network operations & management and, therefore, business continuity. In 2017, a faulty router forced Southwest Airlines to cancel 2,300 flights over four days, resulting in \$74 million loss [5]. The networking community has been keen to investigate solutions to tackle software defects before they cause such damages. Approaches like syntactic meta-data, symbolic execution, assertions, and functional testing have been applied to data plane software testing. While promising, existing methods have limited coverage in terms of classes of defects, also demanding additional programmer intervention (which can be faulty as well) for verification.

Take as an example the P4 code excerpt shown in Fig. 1 of a simple NAT and ACL. Even a simple missing instruction like `ck.update(hdr.ipv4);` (in `TopDeparser` control) cannot be caught during development time without extra information provided by the programmer (in this case, that output IPv4 packets must have a valid checksum). There are more complex cases, however. Liu et al. [6] described a hypothetical case (inspired on a real-world situation of a Cisco product feature update [7]) in which a faulty router was not enforcing ACL rules correctly because they were applied *after* NAT rules (not *before*). This case is also illustrated in Fig. 1, in the `apply` section of `TopPipe` control.

Examples like the one in Fig. 1 illustrate cases of software defects by *omission* (when a protocol specification is not fully implemented in the switch code) and *incorrect fact* (when switch code behavior does not comply with its specification) [8], respectively. Solutions like p4v [6] and ASSERT-P4 [9] are only able to catch them by means of programmer provided syntactic meta-data. The other classes of defects, *ambiguity*, *inconsistency*, and *extraneous information* [8], are only partially covered without need for extra programmer input, like in Vera [10] and p4pktgen [11]. In the remainder of this paper, we review the state-of-the-art on forwarding plane software verification and validation (V&V) and elaborate further on the argument that existing techniques *do not* properly cover the classes of software defects mentioned earlier. From the literature review, we have also identified opportunities in the solution space which have not been explored by previous investigations. In particular, we argue that data flow analysis is a promising building block for designing V&V tools that do not rely on extra programming effort to find switch bugs.

In this paper, we explore the potential of data flow testing to identify defects of the classes inconsistency, omission, incorrect fact, ambiguity, and extraneous information research direction. To this end, we devise a solution that enumerates possible execution paths within a P4 switch program specification and analyzes the order of read/write operations performed on header fields and local variables. We identify potential bug situations, for example, header fields read without a prior write operation, and then generate packets that attempt to exploit them. In case an incorrect packet response is received (for example, a packet that should have been dropped being

<pre>#include <core.p4> #include "vss.p4" header ethernet_t { bit<48> dst_addr; bit<48> src_addr; bit<16> ether_type; } header ipv4_t { bit<4> version; bit<4> ihl; bit<8> diffserv; bit<16> totalLen; bit<16> id; bit<3> flags; bit<13> fragOffset; bit<8> ttl; bit<8> protocol; bit<16> checksum; bit<32> src_addr; bit<32> dst_addr; } struct headers { ethernet_t eth; ipv4_t ipv4; }</pre>	<pre>parser TopParser(packet_in pkt, out headers hdr) { state start { hdr.extract(pkt.eth); transition select(pkt.eth.ether_type) { 0x800: parse_ipv4; default: accept; } } state parse_ipv4 { hdr.extract(pkt.ipv4); transition accept; } control TopDeparser(inout headers hdr, packet_out pkt) { Ck16() ck; apply { pkt.emit(hdr.eth); if (hdr.ipv4.isValid()) { ck.clear(); hdr.ipv4.checksum = 16w0; hdr.ipv4.checksum = ck.get(); } pkt.emit(hdr.ipv4); } } }</pre>	<pre>control TopPipe(inout headers hdr, in error parseError, <in InControl inCtrl, out OutControl outCtrl) { action allow() { } action deny() { outCtrl.outputPort = DROP_PORT; } action rewrite(bit<32> src_addr, PortId port) { hdr.ipv4.src_addr = src_addr; outCtrl.outputPort = port; } table acl { actions = { allow; deny; } key = { hdr.ipv4.src_addr: lpm; } } table nat { actions = { rewrite; } key = { hdr.ipv4.dst_addr: lpm; } } apply { if (hdr.ipv4.isValid()) { acl.apply(); nat.apply(); } } } VSS(TopParser(), TopPipe(), TopDeparser()) main;</pre>
--	---	--

Fig. 1: Simple NAT and ACL P4 code for the very simple switch (VSS) model (code adapted from [6]).

forwarded), a bug is then successfully uncovered.

To assess the technical feasibility of our approach, we developed p4-data-flow, a prototypical implementation of our solution, and evaluated it using popular P4 switch codes publicly available. From our experiments, we confirmed the potentialities of using data flow analysis as a useful resource to catch bugs in switch implementations without requiring any input/effort/knowledge from the switch developer, in contrast to existing solutions. We also discuss the set of bugs detected and potential implications to the switch operations. In summary, we make the following contributions:

- A novel approach, based on data flow analysis, for verification of switch programs written in P4, for uncovering bugs related to inconsistency, omission, incorrect fact, ambiguity, and extraneous information classes of defects;
- An analysis of software defects identified in popular switch implementations widely discussed in the literature.

The remainder of this paper is organized as follow. In Section II, we briefly cover data flow testing aspects that are central to our work and discuss unexplored V&V directions in the solution space. In Section III, we present our solution for data plane verification based on data flow testing, whereas in Section IV we discuss real-world use cases. Finally, we review related work in Section V, and close the paper in Section VI with concluding remarks and directions for future work.

II. BACKGROUND

Software testing is “the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items” [12]. Mainly, there are three types of software testing techniques: black-box, white-box, and defect-based testing techniques. Black-box testing, also known as functional testing, assesses the compliance of a system with the specified functional requirements, regardless of the internal functioning, the code itself, focusing only on outputs that are generated according to inputs and conditions provided. Unlike black-box testing, white-box testing, also known as structural testing,

evaluates the internal functioning of the system using the source code for test case generation. Defect-based technique is a technique that uses known defects to design test cases and to reproduce the defect behavior.

To detect improper use of data values due to coding mistakes, data flow testing can be used as a structural test criterion [13]. Rapps and Weyuker proposed Def-Use Graph, which consists of an extension of the Control Flow Graph (CFG) [14]. In this proposal, information is added to the CFG about the program data flow, which identifies the associations in which a value is assigned to a variable (called a variable definition) and where this value is read (called a variable use). Data flow tests are generated from these associations. According to the data flow model defined in [13], whenever a value is stored in a memory location the definition of the variable is occurring, such as when the variable is on the left side of a command assignment or input command or procedure calls as an output parameter.

To generate data flow tests, all sub-paths are mapped between assigning a variable (definition) to the points at which the variable is used (use). There are two ways a variable can be used: by computing the variable (c-uses) where a value is used in a computation or output statement or by using predicates (p-uses) that occurs whenever a value is used in a predicate statement. The notation for representing these patterns is [14]:

- d – defined, initialized
- u – used
 - c – used in a computation
 - p – used in a predicate

Three possibilities exist for the first occurrence of a variable through a program path. The $\sim$ symbol is used to denote that before this the variable did not exist [14]:

- 1) $\sim d$ – variable does not exist, then it is defined (d)
- 2) $\sim u$ – variable does not exist, then it is used (u)
- 3) $\sim k$ – variable does not exist, then it is destroyed (k)

Of these three possibilities, only the first one is correct, where the variable did not exist and then it is defined. The

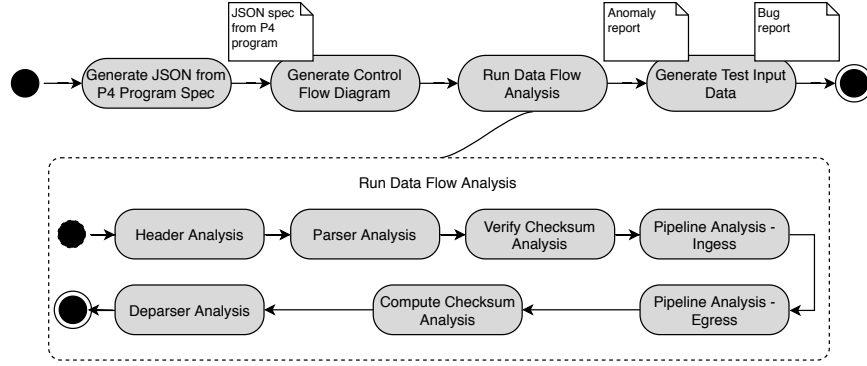


Fig. 2: P4 switch code data flow analysis overview.

second is incorrect because you cannot make a safe use of a variable unless it has been defined before and the third is probably incorrect as well, because a variable is being destroyed before it is created. In P4 language, killing or destroying variables is not a language construct.

Def-use paths (also called du-paths) is an ordered pair (d, u) , where a statement called d contains a definition of a variable v , which is used in a statement u in a program [14]. Table I lists usage combinations and corresponding consequences.

TABLE I: Testing Anomalies

Anomaly		Explanation
dd	Defined and defined again	Not invalid but suspicious. Potential bug.
du	Defined and used	Allowed. Normal case.
ud	Used and defined	Allowed.
uu	Used and used again	Allowed.

Identifying an anomaly in the data flow test does not always represent an incorrect result in the execution of the application. Although it could be a harmless anomaly, it is worth investigating because it often represents a sign of programmer mistake or bad coding practices.

A method for detecting the data flow anomalies has been developed by Fosdick and Osterweil [15]. The basic idea is to compute the so-called path expressions in a flow graph by making use of data flow analysis algorithms developed. A path expression describes all actions that happen to a variable along the paths that have been mapped. Anomalies in the data stream can be detected by the sequence of definitions and uses that occur with each variable along the way.

III. P4 DATA FLOW ANALYSIS

In this section, we present our approach for using data flow analysis to uncover bugs in P4 switch programs, using Fig. 2 as basis. Our approach uses a JSON specification of the switch code, therefore the first step in our verification process is the generation of a JSON specification from a P4 program. The JSON file we use is the one expected by BMv2 behavioral model, software switch popularly used to evaluate the functionalities of a P4 program specification.

Based on the JSON specification, we then generate the control flow graph of the P4 program. This graph contains

every possible execution path within the P4 specification. To illustrate, the control flow graph depicted in Fig. 3, from the basic.p4 switch code¹, shows eight possible execution paths.

For each possible path, we run data flow analysis. This process (detailed in the lower part of Fig. 2) generates a report indicating path expressions on each variable and header fields, as well as identified anomalies on path expressions according to the theory of data flow analysis. Fig. 4 shows the path expressions obtained for the execution path highlighted in dotted lines and gray ellipses in Fig. 3. To illustrate, take the path expression of field `ipv4.ttl`: DUDUU. The first define (D) occurs during `packet.extract(hdr.ipv4)`; header extraction. Then a use (U) followed by define occur on `hdr.ipv4.ttl = hdr.ipv4.ttl - 1`; . Finally, two uses occur for compute checksum and packet deparsing.

There are cases of variable used but never defined, which are reported as bugs. For other potential bugs reported, these are used as input for generating test packets. In this step, we deploy the JSON specification on BMv2 software switch, and inject test packets that attempt to exploit the anomalous path expressions. The test packets are carefully design to explore the execution flow causing the anomalous path expression, and exercise it. In case an abnormal switch behavior occurs (e.g., a packet that should have been dropped is forwarded, or a packet is silently dropped), then a bug is revealed.

Having provided an overview of our solution, next we describe in more detail the data flow analysis process and the automated generation of test packets. In our work, we assume without loss of generality the use of the V1Switch model, which is composed of a parser, verify checksum, ingress, egress, compute checksum, and deparser blocks.

A. P4 JSON Data Flow Analysis

The analysis of each execution path within a P4 switch code comprises processing each of the switch components over that path: header definitions, parser, and controls, as shown in the tasks depicted in the lower part of Fig. 2. The process carried out in each task is described in the algorithms discussed next.

¹The basic.p4 switch source code was obtained from the tutorials available at the official P4 language github repo. A copy of the file can be found at <https://github.com/diogocampos/p4-data-flow/blob/master/examples/basic.p4>

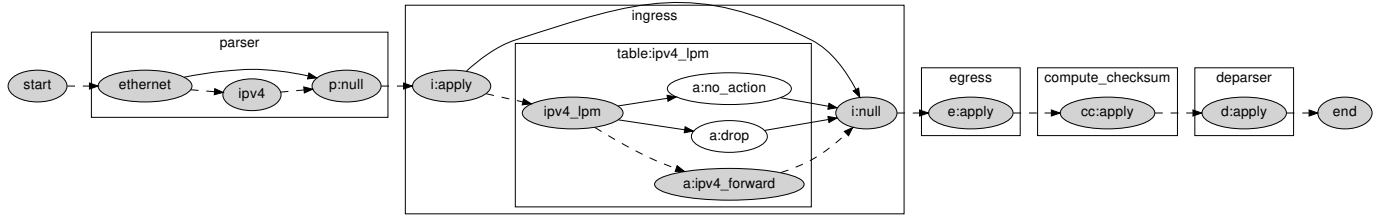


Fig. 3: Control flow graph for the basic.p4 switch program.

```

Execution Path:
0 -> parsers/start -> parsers/parse_ipv4 -> parsers/
    ↪ null -> ingress/node_2 -> ingress/MyIngress.
    ↪ ipv4_lpm -> ingress/MyIngress.ipv4_lpm/
    ↪ MyIngress.ipv4_forward -> ingress/null ->
    ↪ egress/null -> compute_checksum -> deparsers

DF_Table:
standard_metadata
  egress_spec: D
ethernet
  dstAddr: DUDU
  srcAddr: DDU
  etherType: DUU
ipv4
  version: DUU
  ihl: DUU
  diffserv: DUU
  totalLen: DUU
  identification: DUU
  flags: DUU
  fragOffset: DUU
  ttl: DUDUU
  protocol: DUU
  hdrChecksum: DDU
  srcAddr: DUU
  dstAddr: DUUU
  $valid$: UU
MyIngress.ipv4_forward
  dstAddr: PU
  port: PU

```

Fig. 4: An example of path expressions obtained for the basic.p4 switch code, for the path highlighted in Fig. 3. In the excerpt above, P means parameter passing. Note that the execution path above follows the JSON notation generated by the p4c compiler for bmv2 simple switch model.

Algorithm 1 Header Analysis

- 1: Fetch headers declaration in JSON file
- 2: **for** each declared header **do**
- 3: Find header in the JSON header_type specification
- 4: Store in DF_Table each field from the header

Algorithm 1 shows the routine for header analysis. In summary, we fetch the headers declaration from the JSON file and, for each header, we extract its fields and store them in a definition table (DF_Table). This is a global table we use to check path expressions in each header and metadata field.

After the extraction of the header definitions, the next step is processing the switch parser. The routine, depicted in Algorithm 2, is responsible for applying the effects of the JSON parser definitions into extracted header fields. Given that the switch parser is a finite state machine, the first step in the algorithm is finding the initial parser state. While next

state is not null, we search for the state operation (lines 4-24) and transition (lines 25-34). The state operation could be:

Algorithm 2 Parser Analysis

- 1: Fetch parsers declaration in JSON file
- 2: Find init_state in the JSON parse_states tuple
- 3: **repeat**
- 4: Find parser_ops
- 5: **for** each declared parser_ops **do**
- 6: **if** "op" : "extract" **then**
- 7: Find parameters in tuple parser_ops
- 8: **if** "type" : "regular" **then**
- 9: Append 'D' to all fields of value structure
- 10: **else if** "op" : "set" **then**
- 11: Find first item of tuple parameters
- 12: Append 'D' to the value field
- 13: Find second item of tuple parameters
- 14: **if** "type" : "field" **then**
- 15: Append 'U' to the value field
- 16: **else if** "type" : "expression" **then**
- 17: **for** each value tuple item **do**
- 18: **if** "type": "field" or "type": "runtimedata" **then**
- 19: Append 'U' to the value field
- 20: **else if** "op" : "verify" **then**
- 21: Find first item of tuple parameters
- 22: **for** each value tuple item **do**
- 23: **if** "type": "field" or "type": "runtimedata" **then**
- 24: Append 'U' to the value field
- 25: Find transition_key in tuple parser_states
- 26: **for** each declared transition_key **do**
- 27: **if** "type" : "field" **then**
- 28: Append 'U' to the value field
- 29: Find transitions in tuple parser_states
- 30: **for** (each declared transitions **do**
- 31: **if** "type" : "hexstr" **then**
- 32: **if** match **then**
- 33: **if** "next_state" != null **then**
- 34: find next_state in tuple parser_states
- 35: **until** next_state != null

- **extract (lines 6-9):** Applied to a packet, it populates a header with the next sizeof header bits from the packet stream. An example from the basic.p4 code is `packet.extract(hdr.ethernet);`
- **set (lines 10-19):** The `set()` method is an assignment, written with the `=` sign. It first evaluates its left sub-expression to an l-value, then its right sub-expression to a value, and finally copies the value into the l-value;
- **verify (lines 20-24):** The `verify()` statement provides a simple form of error handling. Verify can only be

invoked within a parser. If the first argument is true, then executing the statement has no side-effect. However, if the first argument is false, it causes an immediate transition to reject, which causes immediate parsing termination; at the same time, the `parserError` associated with the parser is set to the value of the second argument.

Algorithm 3 Ingress Analysis

```

1: Fetch ingress in pipelines declaration in JSON file
2: if "init_table" != null then
3:   Find init_table in the JSON tables tuple
4:   repeat
5:     if table found then
6:       Find key in tables tuple
7:       if "match_type" != "valid" then
8:         Append 'U' to the target field
9:       if match_type true then
10:        Select one action in ingress tuple
11:        Find action in actions tuple
12:        Store in DF_Table each runtime_data item
13:        Append 'D' to each runtime_data item
14:        for each primitives in action tuple do
15:          if "op" : "assign" then
16:            Find second item in parameters tuple
17:            Append 'U' to the value field
18:            Find first item in parameters tuple
19:            Append 'D' to the value field
20:          else if "op" : "drop" then
21:            Append 'D' to the egress_spec field
22:          else if match_type false then
23:            Find default_entry in tables tuple
24:            Find action_id in actions tuple
25:            Store in DF_Table each runtime_data item
26:            Append 'D' to each runtime_data item
27:            for each primitives in action tuple do
28:              if "op" : "assign" then
29:                Find second item in parameters tuple
30:                Append 'U' to the value field
31:                Find first item in parameters tuple
32:                Append 'D' to the value field
33:              else if "op" : "drop" then
34:                Append 'D' to the egress_spec field
35:          else if table not found then
36:            Find init_table in conditionals tuple
37:            if left != null in expression then
38:              Append 'U' to the value field
39:            else if left : null in expression then
40:              Find right in expression
41:              Append 'U' to the value field
42:            if conditional = true then
43:              Find true_next state in the JSON tables tuple
44:            else
45:              Find false_next state in the JSON tables tuple
46:   until next_state != null

```

After processing the parser, we proceed to the analysis of the verify checksum control. For the sake of brevity, we omit this routine in the paper. In summary, it checks for calculations and then populates the `DF_Table` with respective usage and definitions. We then move to the ingress analysis, whose routine is depicted in Algorithm 3. The routine starts by fetching the ingress in the JSON pipeline declaration (line 1). Then, it fetches the initial table and its keys (lines 3-6). For

each matching case (line 9), we process each of the possible actions defined within that table (lines 10-22). We also process the default case, in which there is no match (lines 23-34).

In case a table is not directly found (line 35), we fetch a conditional that could be present (lines 37-41), and then explore the next state that can be reached. In case the conditional cannot be evaluated based on the `DF_Table` state, we explore both states from the conditional.

Algorithm 4 Deparser Analysis

```

1: Fetch deparsers declaration in JSON file
2: for each declared order do
3:   Append 'U' to all values field of this package

```

Following the pipeline depicted in the lower part of Fig. 2, we proceed to the analysis of the compute checksum control, whose algorithm, omitted for the sake of brevity, follows a logic similar to the verify checksum discussed earlier. Finally, we move to the analysis of the deparser control. The routine is depicted in Algorithm 4. In summary, for each deparser declaration in the JSON file, we process it by updating the `DF_Table` with each usage definition found.

B. Test Packet Generation

There are a few potential bugs that may need further verification to be confirmed. For exercising such cases, we use the P4 packet test framework (PTF)². In summary, we deploy the switch code using `bmv2`, populate its tables with a minimal set of rules that exercise the path to be explored, and send a set of test packets, fixing to zero/undefined such suspicious cases. In case the packet is not processed according to the expected outcome (e.g., forwarded when it should have been dropped), then the bug is confirmed.

It is important to emphasize that only a small subset of bugs reported need this closer inspection using such white-box testing technique. These are the cases, for example, of variables which are defined twice without a use in between (i.e., its path expression has a DD sub-sequence). The rationale is that the occurrence of a variable that is written twice could indicate an issue in the switch programming logic.

IV. EVALUATION

We implemented a prototype of our solution, `p4-data-flow`, using python 3.7. Our prototype has around 450 lines of code, and is available on GitHub³. In our evaluation, we considered many switch implementations publicly available, mainly from the official P4 repo⁴. In Table II we present a summary of some of the switches tested and bugs found. Our verification times were measured on a ultra-book with Intel Core i7-8550U CPU @ 1.80GHz and 16 GB of RAM, running Ubuntu 16.04.

Since many of these implementations were available on P4_14, we used the P4 `p4test` tool to convert them to P4_16. We then used `p4c-bm2-ss` for P4_16 to JSON code conversion.

²PTF GitHub repo: <https://github.com/p4lang/ptf/>

³p4-data-flow GitHub repo: <https://github.com/diogocampos/p4-data-flow/>

⁴Official P4 GitHub repo: <https://github.com/p4lang/>

TABLE II: Switch programs used in our evaluation and bugs found

Program	Size (LOC)	Version	Number of Execution Paths	Verification Time (sec)	Omission	Incorrect Fact
simple_nat	363	P4_14	6,300	3.64	X	X
load_balance	226	P4_16	60	0.02	X	
flowlet_switching	163	P4_16	1,458	3.50		X
checksum	118	P4_14	18	0.12		X

The original switch codes tested, their code converted P4_16, as well as generated JSON files and verification output, are also available in our repository. Finally, we used bmv2, ptf, and scapy for exercising and confirming potential bugs.

A. Simple NAT

Our first experiment is the simple_nat program, which implements a NAT box with IPv4 support. Using p4-data-flow, we found three bugs in its code. Fig. 5 shows execution paths (and respective path expressions of header fields) related to two of these bugs. The first one refers to the possibility of packets without a IPv4 header being processed by the ipv4_lpm table. Observe in the first execution path that after parsing the Ethernet header (parsers/ethernet), the parser goes to the exit state (parsers/null). This is the case of the path default: accept; in the transition select (hdr.ethernet.etherType), as one can see in the simple_nat-16.p4 code available in our repo. Later in the execution path (ingress/node_4 -> ingress/ipv4_lpm), a buggy conditional allows the packet without a valid IPv4 header to be processed by the ipv4_lpm table.

The faulty code in this case is if (meta.meta.do_forward == 1w1 && hdr.ipv4.ttl > 8w0). From the P4_16 specification⁵, header fields that are not defined prior to use could have an undefined value and therefore lead the switch to an abnormal behavior. In fact, after exercising the bug using PTF, we found that a packet without IPv4 header that should have been dropped was actually forwarded by the switch (software defect due to incorrect fact). To prevent it, the authors of the simple_nat switch code should have checked the validity of the IPv4 header, using the method hdr.ipv4.isValid().

The second bug is related to TCP header fields not extracted but NATed anyway. Note in the second execution path in Fig. 5 that the IPv4 header is parsed, but not the TCP header (parsers/parse_ethernet -> parsers/parse_ipv4 -> parsers/parse_null). Then, in egress/send_frame, action do_rewrites is triggered without a check if the TCP header is valid (a software defect by omission). Finally, the third bug found is related to a use of the IPv4 TTL field without a prior definition (incorrect fact), causing a wrong ipv4_lpm table apply.

B. Load Balance

Our second experiment is the load_balance switch from the official P4 tutorial. In Fig. 6 we present one execution

```

0 -> parsers/start -> parsers/parse_cpu_header -> parsers/
    ↪ parse_ethernet -> parsers/null -> ingress/if_info
    ↪ -> ingress/if_info/set_if_info -> ingress/nat ->
    ↪ ingress/nat/nat_hit_ext_to_int -> ingress/node_4
    ↪ -> ingress/ipv4_lpm -> ingress/ipv4_lpm/set_nhops
    ↪ -> ingress/forward -> ingress/forward/set_dmac ->
    ↪ ingress/null -> egress/node_9 -> egress/
    ↪ send_to_cpu -> egress/send_to_cpu/do_cpu_encap ->
    ↪ egress/null -> compute_checksum -> deparsers
ethernet
  dstAddr: DDU
ipv4
  version: UU
  ihl: UU
  diffserv: UU
  totalLen: UU
  identification: UU
  flags: UU
  fragOffset: UU
  ttl: UUUUU
  protocol: UUU
  srcAddr: UUU
  dstAddr: UUU
tcp
  srcPort: UU
  dstPort: UU
  seqNo: UU
  ackNo: UU
  dataOffset: UU
  res: UU
  flags: UU
  window: UU
  urgentPtr: UU
0 -> parsers/start -> parsers/parse_cpu_header -> parsers/
    ↪ parse_ethernet -> parsers/parse_ipv4 -> parsers/
    ↪ null -> ingress/if_info -> ingress/if_info/
    ↪ set_if_info -> ingress/nat -> ingress/nat/
    ↪ nat_hit_int_to_ext -> ingress/node_4 -> ingress/
    ↪ ipv4_lpm -> ingress/ipv4_lpm/set_nhops -> ingress/
    ↪ forward -> ingress/forward/set_dmac -> ingress/
    ↪ null -> egress/node_9 -> egress/send_frame ->
    ↪ egress/send_frame/do_rewrites -> egress/null ->
    ↪ compute_checksum -> deparsers
ethernet
  dstAddr: DDU
ipv4
  hdrChecksum: DDU
tcp
  seqNo: UU
  ackNo: UU
  dataOffset: UU
  res: UU
  flags: UU
  window: UU
  urgentPtr: UU

```

Fig. 5: Execution paths with faulty behavior in the simple_nat switch implementation.

path with a faulty code. Before applying the ecmp_group table, the authors do not check if the packet has a valid TCP header. In action set_ecmp_select, a hash function is applied over srcAddr, dstAddr, and protocol fields of the ipv4

⁵P4_16 specification: <https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html>

```

0 -> parsers/start -> parsers/parse_ipv4 -> parsers/null
    ↳ -> ingress/node_2 -> ingress/MyIngress.ecmp_group
    ↳ -> ingress/MyIngress.ecmp_group/MyIngress.
    ↳ set_ecmp_select -> ingress/MyIngress.ecmp_nhop ->
    ↳ ingress/MyIngress.ecmp_nhop/MyIngress.set_nhop ->
    ↳ ingress/null -> egress/MyEgress.send_frame ->
    ↳ egress/MyEgress.send_frame/MyEgress.rewrite_mac ->
    ↳ egress/null -> compute_checksum -> deparsers
ethernet
  dstAddr: DDU
  srcAddr: DDU
ipv4
  hdrChecksum: DDU
tcp
  srcPort: U
  dstPort: U

```

Fig. 6: Faulty behavior in the load_balance switch code.

header, and srcPort and dstPort fields of the tcp header; the result of the hash function is stored in meta.ecmp_select. Since hdr.tcp.srcPort and hdr.tcp.dstPort are undefined in the execution path shown in Fig. 6, the result of the hash, and therefore variable meta.ecmp_select, becomes undefined. The meta.ecmp_select is later used to determine which path the packet should follow. In our experiment, the packets were forwarded to a same switch.

C. Flowlet Switching and Checksum P4

We also evaluated the flowlet switching [16] and checksum implementations available in the official P4 GitHub repo. In the flowlet switching execution path shown in Fig. 7, there are path expressions in which ipv4 and tcp header fields are used but never defined (tcp path expressions omitted for brevity).

The problem in the flowlet switching program (which also applies to the checksum program) is an unverified ipv4 and tcp header validity before applying a hash function. As a result, it becomes unstable. Similarly to the load balance, a simple solution would be testing isValid() before applying any tables that access those header fields, like the flowlet, ecmp_group, and ecmp_nhop tables in flowlet switching.

D. Discussion on Limitations and Applicability

Our solution scales well with the number of execution paths within the switch program, as shown in the experiments considered. However, for “branchier” codes, higher verification times could be observed. Note also that our solution does not uncover any possible types of switch bugs, being therefore a complement to existing tooling that does not require programmer intervention, like Vera [10].

We performed experiments using P4 programs available from well-know repositories and publications in prestigious venues in the field. Although more straightforward compared to real switch code, they still represent an essential benchmark to analyze the effectiveness of p4-data-flow compared to the state-of-the-art. Also, the size and complexity of these programs provide a more controllable testing environment, which makes it easier to rule out false-positives. We are working on a broader validation of p4-data-flow using more sophisticated programs to validate function correctness and scalability.

```

0 -> parsers/start -> parsers/parse_ethernet -> parsers/
    ↳ null -> ingress/ingress.flowlet -> ingress/ingress/
    ↳ .flowlet/ingress.lookup_flowlet_map -> ingress/
    ↳ node_3 -> ingress/ingress.new_flowlet -> ingress/
    ↳ ingress.new_flowlet/ingress.update_flowlet_id ->
    ↳ ingress/ingress.ecmp_group -> ingress/ingress.
    ↳ ecmp_group/ingress.set_ecmp_select -> ingress/
    ↳ ingress.ecmp_nhop -> ingress/ingress.ecmp_nhop/
    ↳ ingress.set_nhop -> ingress/ingress.forward ->
    ↳ ingress/ingress.forward/ingress.set_dmac ->
    ↳ ingress/null -> egress/egress.send_frame -> egress/
    ↳ /egress.send_frame/egress.rewrite_mac -> egress/
    ↳ null -> compute_checksum -> deparsers
ipv4
  version: UUU
  ihl: UUU
  diffserv: UUU
  totalLen: UUU
  identification: UUU
  flags: UUU
  fragOffset: UUU
  ttl: UUUUU
  protocol: UUU
  hdrChecksum: DDU
  srcAddr: UUU
  dstAddr: UUUU

```

Fig. 7: Faulty behavior in the flowlet switching code.

A current limitation of our prototype is that some P4 primitives like resubmit and recirculate are not yet implemented, and further analysis of the codes evaluated is still required. With regard to test packet generation, our current focus is exploring data flow analysis for P4 program verification. We are investigating heuristics to automate test case generation based on the outcome of potential bugs, possibly using Header Space Analysis (HSA) or symbolic execution.

V. RELATED WORK

The topic of network verification has seen some intense research activity in the past few years. Existing work have either approached static analysis techniques, like symbolic execution and model checking (e.g., p4v [6], Vera [10], and ASSERT-P4 [9]), and dynamic analysis (e.g., p4pktgen [11]). Static analysis techniques are those that do not involve running any code, and may be used to identify defects before an executable version of the system is available [17].

One example in the category of static analysis is Network Optimized Datalog (NoD) [18]. NoD is a network verification tool in which operators may express beliefs about the network using Datalog (high-level invariants, like “the print server can only be accessed from the intranet”), and verify them through analysis of forwarding tables and ACLs. In another direction, Dobrescu and Argyraki [19] have used symbolic execution with S2e [20] to analyze implementations of Click modular router elements [21]. One problem to these approaches is the possibility of path explosions during verification.

There are also solutions that rely on code annotations for static verification. ASSERT-P4 [9] enables developers to write assertions into P4 code that specify network correctness properties. The program and assertions are then translated into C models and verified using symbolic execution. p4v [6] also proposes that developers annotate programs with Hoare logic

clause (pre and post conditions) to enable static verification. In common to ASSERT-P4 and p4v is the fact that extra developer effort is required to express the aspects that must be verified, an error-prone approach. More importantly, if some critical network property is not specified through an assertion in the code, it will be left unverified.

Vera [10] is another static verification tool that, unlike p4v and ASSERT-P4, does not require annotations. Instead, it uses symbolic execution and SEFL [22] (language designed for network verification) to analyze P4 code. Since it relies on symbolic execution, it may not scale well for “branchy” codes.

In the realm of dynamic analysis, p4pktgen [11] is a relevant example. It also uses symbolic execution to generate test packets and predict the expected output, and uses bmv2 for assessing the behavior of a P4 program. This approach may even uncover bugs in the compiler and the software switch. On the downside, it requires switch deployment for testing and may not scale well for complex programs.

Another important drawback common to the solutions above is that they cannot cover defects due to omission and any other that does not cause runtime exceptions (e.g., a hash computed over an undefined header field) – unless additional code is written by the developer, in the case of assertion-based solutions. We therefore argue that a important contribution of our approach is covering a relevant part of the verification space, without human intervention or extra developer effort.

VI. FINAL CONSIDERATIONS

The possibility of defining switch behavior brought by programmable forwarding planes mandates novel approaches to switch development and testing. In this paper, we reviewed the state-of-the-art on verification of programmable data planes, and discussed the potentialities of using data flow analysis as a resource to detect issues in P4 switch implementations. To this end, we devised p4-data-flow, a python program that analyzes path expressions in switch metadata variables and header fields and detects potential bugs in the P4 switch code. While we focused on P4 (mostly due to the availability toolkit and popular switch implementations), our approach could be easily generalized to address switch programs written in other domain specific languages, like POF [3].

We emphasize that our approach does not replace, but complements existing work in the field, like Vera [10], p4v [6], and ASSERT-P4 [9]. While they cannot catch defects that fall in classes like *omission*, we envisage that a fully-fledged verification tool must incorporate a data flow analysis based approach as well as contributions of previous investigations.

In spite of the progress achieved, much work remains. One research avenue for future investigation is the optimization of the P4 code analysis using heuristics to prune the search space. For example, not all execution paths need to be evaluated as they might be unfeasible (i.e., would not be reached during normal switch operation). We also envisage embedding our solution with symbolic execution to tackle the need for white box testing for potential but unconfirmed bugs after analysis.

ACKNOWLEDGEMENTS

This work has been supported in part by funding from National Science Foundation (NSF Award Number 1740791), Brazilian Education and Research Network (RNP), and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001.

REFERENCES

- [1] N. Feamster, J. Rexford, and E. Zegura, “The road to sdn: An intellectual history of programmable networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014.
- [2] W. Cordeiro, J. Marques, and L. Gaspary, “Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management,” *J. Netw. Syst. Manage.*, vol. 25, no. 4, pp. 784–818, Oct. 2017.
- [3] H. Song, “Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane,” in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’13. New York, NY, USA: ACM, 2013, pp. 127–132.
- [4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, “P4: Programming protocol-independent packet processors,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
- [5] S. Carey, “Why a single failed router can ground a thousand flights,” *The Wall Street Journal*. [Online]. Available: <https://www.wsj.com/articles/why-a-single-failed-router-can-ground-a-thousand-flights-1489743001>
- [6] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcal, N. McKeown, and N. Foster, “P4v: Practical verification for programmable data planes,” in *ACM SIGCOMM 2018*. New York, NY, USA: ACM, 2018, pp. 490–503.
- [7] P. Kazemian, “Network path not found?” *Forward Networks Blog*, February 2017. [Online]. Available: <https://www.forwardnetworks.com/network-path-not-found-how-to-use-an-army-of-tests-to-understand-and-diagnose-your-network/>
- [8] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, “Detecting defects in object-oriented designs: Using reading techniques to increase software quality,” *SIGPLAN Not.*, vol. 34, no. 10, pp. 47–56, Oct. 1999.
- [9] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos, “Verification of p4 programs in feasible time using assertions,” in *CoNEXT ’18*. New York, NY, USA: ACM, 2018, pp. 73–85.
- [10] R. Stoicescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, “Debugging p4 programs with vera,” in *ACM SIGCOMM 2018*. New York, NY, USA: ACM, 2018, pp. 518–532.
- [11] A. Nötzi, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, “P4pktgen: Automated test case generation for p4 programs,” in *Proceedings of the Symposium on SDN Research*. ACM, 2018, p. 5.
- [12] IEEE, “IEEE standard glossary of software engineering terminology,” *IEEE Std 610.12-1990*, Dec. 1990.
- [13] J. C. Maldonado *et al.*, “Critérios potenciais usos: Uma contribuição ao teste estrutural de software,” (*Publicação FEE*), 1991.
- [14] S. Rapps and E. J. Weyuker, “Selecting software test data using data flow information,” *IEEE Trans. Softw. Eng.*, vol. 11, no. 4, pp. 367–375, Apr. 1985. [Online]. Available: <http://dx.doi.org/10.1109/TSE.1985.232226>
- [15] L. D. Fosdick and L. J. Osterweil, “Data flow analysis in software reliability,” in *Engineering of Software*. Springer, 2011, pp. 49–85.
- [16] S. Sinha, S. Kandula, and D. Katabi, “Harnessing tcps burstiness with flowlet switching,” in *Hot Topics in Networks (HotNets)*. ACM, 2004.
- [17] I. Sommerville *et al.*, *Software engineering*. Addison-wesley, 2007.
- [18] N. P. Lopes, N. Björner, P. Godefroid, K. Jayaraman, and G. Varghese, “Checking beliefs in dynamic networks,” in *NSDI*, 2015, pp. 499–512.
- [19] M. Dobrescu and K. Argyraki, “Software dataplane verification,” in *NSDI 14*, 2014, pp. 101–114.
- [20] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.
- [21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.
- [22] R. Stoicescu, M. Popovici, L. Negreanu, and C. Raiciu, “Symnet: Scalable symbolic execution for modern networks,” in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 314–327.