UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

KARINE BIRNFELD

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

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Weverton Luis da Costa
Cordeiro
Coadvisor: Prof. Dr. Breno Bernard Nicolau de
França

Porto Alegre
July 2020

CIP — CATALOGING-IN-PUBLICATION

Birnfeld, Karine

P4 Switch Code Data Flow Analysis: Towards Stronger Verification of Forwarding Plane Software / Karine Birnfeld. – Porto Alegre: PPGC da UFRGS, 2020.

55 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Weverton Luis da Costa Cordeiro; Coadvisor: Breno Bernard Nicolau de França.

1. P4. 2. Programmable Network. 3. Programmable Data-plane. 4. Software Testing. 5. Data Flow Testing. I. Cordeiro, Weverton Luis da Costa. II. de França, Breno Bernard Nicolau. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof^{a}. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof^{a}. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. João Luiz Dihl Comba
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"The only way to do great work

is to love what you do."

— STEVE JOBS

ACKNOWLEDGMENTS

First of all, I would like to thank God for giving me a healthy and fulfilling life to complete this journey.

I am especially grateful to professors Dr. Weverton Cordeiro and Dr. Breno França for all the support, trust, contributions, professionalism in guiding this dissertation, and for understanding / conducting with great patience the difficult moments we faced.

I would like to thank my husband Diego for his companionship, understanding, and patience. For the unconditional support for me to attend the master's degree and make this dream come true. For believing in me and supporting me in the moments I didn't think I could.

I thank my children who are my source of inspiration and courage to continue on this journey. For understanding the times I was absent to carry out this work.

Thanks to my parents for all the teaching and especially for the constant encouragement to study throughout my life.

ABSTRACT

The advent of Domain Specific Languages (DSL) like POF and P4 has enabled for the first time network operators to quickly redefine how forwarding plane devices (*e.g.*, switches) parse and process packets in a Software-Defined Network (SDN). In this context, proper verification of home-brewed forwarding plane software 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 forwarding plane code. In spite of the potentialities, they are limited in the classes of errors they can capture. More importantly, existing proposals for verifying forwarding plane software often require a programmer to write additional verification code (*e.g.*, annotations), an error-prone approach in itself. In this dissertation, 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.

Keywords: P4. Programmable Network. Programmable Dataplane. Software Testing. Data Flow Testing.

Análise do Fluxo de Dados de Código Switch P4

RESUMO

O advento de linguagens específicas de domínio como POF e P4 permitiram pela primeira vez que operadores de rede pudessem prontamente redefinir como os dispositivos de encaminhamento de dados (por exemplo, *switches*) interpretam e processam pacotes em uma Rede Definida por Software (SDN). Nesse contexto, a verificação de *software* para o plano de dados desenvolvido “em casa” se torna fundamental para evitar interrupções na rede e no serviço devido a problemas de implementação. Várias técnicas, como execução simbólica, anotações e asserções, foram usadas recentemente para garantir códigos de planos de dados programáveis livres de erros. Apesar das potencialidades, elas são limitadas nas classes de erros que podem capturar. Mais importante, as propostas existentes para verificação do programa de switch geralmente exigem que o programador escreva um código de verificação personalizado (por exemplo, anotações), uma abordagem propensa a erros. Nesta dissertação, apresentamos o design e a implementação do P4-DATA-FLOW, uma ferramenta prática que utiliza a análise de fluxo de dados para verificação de programas de switch. Nós nos concentramos na linguagem P4 e apresentamos experimentos mostrando que a análise do fluxo de dados pode revelar defeitos de classes ainda não cobertas pelos trabalhos existente, sem exigir mais esforço do programador.

Palavras-chave: Rede programável, Plano de dados programável, Teste de Software, Teste de fluxo de dados.

LIST OF ABBREVIATIONS AND ACRONYMS

ASICs	Application-specific integrated circuit
BMv2	Behavioral Model version 2
CFG	Control Flow Graph
FPGA	Field Programmable Gate Array
HSA	Header Space Analysis
JSON	JavaScript Object Notation
NAT	Network Address Translation
NoD	Network Optimized Datalog
OF	OpenFlow
P4	Programming Protocol-Independent Packet Processors
PISA	Protocol Independent Switch Architecture
POF	Protocol Oblivious Forwarding
PTF	Packet Test framework
SEFL	Symbolic Execution Friendly Language
SDN	Software Defined Networking
SLR	Systematic Literature Reviews
SMS	Systematic Mapping Studies
TCP	Transmission Control Protocol
V&V	Verification and Validation

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The concept of programmable forwarding planes has experienced a renewed research interest since the advent of Software Defined Networking (SDN) (FEAMSTER; REXFORD; ZEGURA, 2014; CORDEIRO; MARQUES; GASPARY, 2017). Domain-specific languages like POF (SONG, 2013) and P4 (BOSSHART et al., 2014) 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 (CAREY, 2017). The networking community has been keen to investigate solutions to tackle software defects before they cause such damages. Approaches like static checking (LOPES et al., 2016), syntactic meta-data and assertions (FREIRE et al., 2018; NEVES et al., 2018; LIU et al., 2018), symbolic execution (STOENESCU et al., 2018; FREIRE et al., 2018; NEVES et al., 2018; LIU et al., 2018), functional testing (NöTZLI et al., 2018; ZHOU et al., 2019), and machine learning (SHUKLA et al., 2019) 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. This scenario demands complementary techniques to consolidate a broader V&V strategy for data plane software.

Take as an example the P4 code excerpt shown in Fig. 1.1 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. (LIU et al., 2018) described a hypothetical case inspired on a real-world situation of a Cisco product feature update (KAZEMIAN, 2017), in which a faulty router was not enforcing ACL rules correctly. The cause of the problem was a change in the order of application of ACL rules and NAT rules from one product version to another, which made ACL rules to be applied *after* NAT rules (instead of applying them *before*, as it occurred in the previous version),

Figure 1.1: Simple NAT and ACL P4 code for the very simple switch (VSS) model (Adapted from (LIU et al., 2018)).

```
#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;
}
```

```
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);
  }
}
```

```
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;
```

thus breaking the correct application of configured network policies. This case is also illustrated in Fig. 1.1, in the `apply` section of `TopPipe` control.

Examples like the one in Fig. 1.1 illustrate cases of software defects by *omission*, *i.e.*, 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 (TRAVASSOS et al., 1999), respectively. Solutions like p4v (LIU et al., 2018) and ASSERT-P4 (NEVES et al., 2018) are only able to catch them by means of programmer-provided syntactic meta-data. The other classes of defects, *ambiguity*, *inconsistency*, and *extraneous information* (TRAVASSOS et al., 1999), are only partially covered without need for extra programmer input, like in Vera (STOENESCU et al., 2018) and p4pktgen (NöTZLI et al., 2018). In the remainder of this dissertation, 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 dissertation, we explore the potential of data flow testing to identify defects of the different classes (see details in Section 2.2). 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 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 and limitations of using data flow analysis as a resource to catch bugs in switch implementations without requiring any input/effort/knowledge from the switch developer, when compared 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 dissertation is organized as follow. In Chapter 2 we briefly cover data flow testing aspects that are central to our work and discuss unexplored V&V directions in the solution space. We review related work in Chapter 3. In Chapter 4 we present our solution for data plane verification based on data flow testing, whereas in Chapter 5 we discuss real-world use cases. We close the dissertation in Chapter 6 with concluding remarks and directions for future work.

2 BACKGROUND

This chapter presents the theoretical foundations used for developing this work. In the Section 2.1 we describe the main concepts of the P4 language. Next, in the Section 2.2 we present Software Testing concepts and techniques and, finally, in the Section 2.3 we describe the definition of Data Flow Testing.

2.1 Software Defined Networking and P4

In the past decade, network practitioners experienced a revolution in the way they operate and manage networked systems, with the advent of Software Defined Networking (SDN) (KREUTZ et al., 2015). Before SDN, a common practice was dealing with a multitude of forwarding devices like switches, routers, and middleboxes in general, each from a specific vendor and having proprietary management interfaces, supporting a specific set of supported protocols, and frequently lacking interoperability. Each device had to be configured independently for implementing global networking policies for traffic engineering, routing, security, etc. Combined, these aspects severely hampered effective and efficient network operations and management, making them a daunting and error-prone task (KREUTZ et al., 2015).

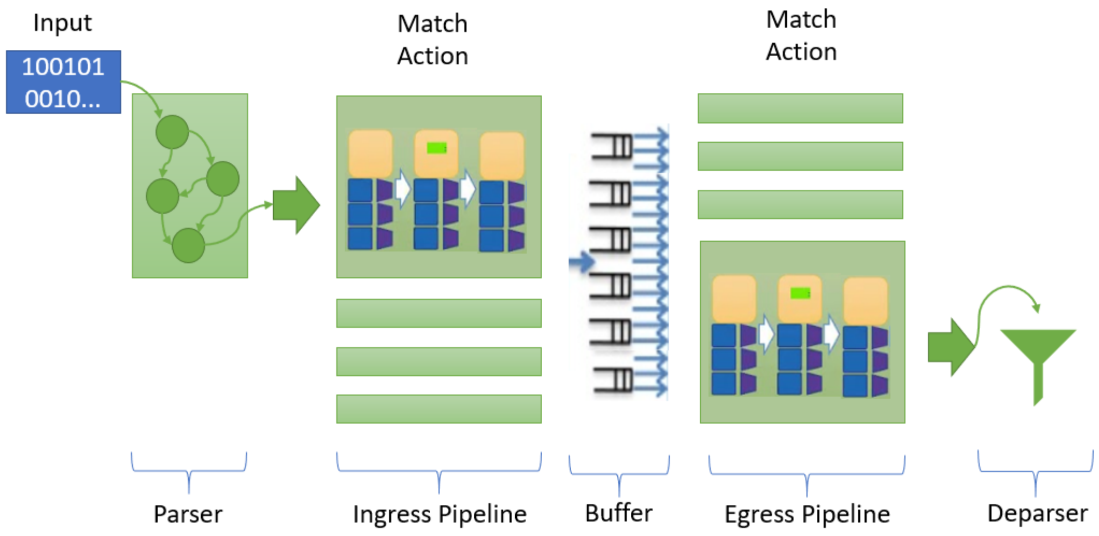
The emergence of OpenFlow (MCKEOWN et al., 2008) changed dramatically the state of affairs. OpenFlow provided a standard and open interface that network operators could use to configure networking devices with flow forwarding actions based on packet field matching. OpenFlow also enabled decoupling the network intelligence from the forwarding device, thus fostering a logical reorganization of the network between a *control plane* (where networking intelligence dwells) and a *data plane* (where packet handling occurs) (HALEPLIDIS et al., 2015).

The logical separation of the network between a control and a data plane fostered further research to enable forwarding devices to achieve higher levels of programmability. Using OpenFlow as a common, open, and provider-independent interface, the control layer proved indispensable to the data plane. However, to enable existing devices to expose more of their resources to the controller, the OpenFlow specification has become increasingly complex, adding fields and stages of rules (BOSSHART et al., 2014).

Making the data plane programmable required a novel language for expressing the forwarding plane behavior, *i.e.*, how it should parse and process packets. By 2013, renewed ideas for data plane programmability emerged with the proposal of Reconfigurable Match-Action tables (BOSSHART et al., 2013) and a high-level domain specific language for redefining Match-Action tables in forwarding devices (SONG, 2013; BOSSHART et al., 2014). One of the proposed languages was P4 (BOSSHART et al., 2014), acronym for Programming Protocol-Independent Packet Processors. The language took shape with the release of the first technical specification, P4_14[1]. In 2016, a novel specification was released, P4_16[2]. P4 is a language that describes how a packet should be processed by the data plane of a programmable forwarding device. Through the P4 language, it is possible to abstract the data plane making the programming of the network equipment behavior simpler and performed through a high-level programming language.

The P4 language is based on the protocol-independent switch architecture (PISA) architecture. Through PISA, we have a new hardware paradigm based on a configuration pipeline using the Match-Action model. Thus, unlike traditional ASICs, the architecture provides great flexibility without compromising performance. Figure 2.1 presents an overview of the process of an equipment using PISA (BOSSHART et al., 2014).

Figure 2.1: The abstract forwarding model. Source (BOSSHART et al., 2014)



As shown in Figure 2.1, packets are first handled by the parser. The parser recognizes and removes fields from the header and thus defines the protocols supported by the switch. The extracted fields are passed to the match-action tables, divided between input (ingress) and output (egress) stages. Although both can modify the header, match-action ingress determines the egress ports and the queue in which the packet is placed. Based on this processing, the packet can be forwarded, replicated, dropped or even sent

[1]https://p4.org/p4-spec/p4-14/v1.0.2/tex/p4.pdf
[2]https://p4.org/p4-spec/docs/P4-16-v1.2.0.pdf

to the control plane for further inspection. The egress match-action often performs modifications to the packet header, but may also determine the output port. The action tables represent a stream through which a network operator may track the frame-to-frame state and metadata information that offer details of the steps traversed by packets.

The design of a P4 program requires filling in five main PISA switch constructs (BOSSHART et al., 2014), enumerated next:

- Headers: declaration of packet header fields, and specification of the order in which they must be parsed. The header definition describes the sequence and structure of a series of bit fields defined by the programmer. It may include, for example, header field width specifications, types, and value restrictions;
- Parser: it specifies how to identify valid headers or header sequences in packets. Also, it is responsible for analyzing and extracting packet fields;
- Tables: mechanism that performs the packet processing. Within the tables, there are the matches and the actions to be performed;
- Actions: set of primitives used for a certain custom function, which may contain primitives such as: copy_header, remove_header, add, etc. P4 supports construction of complex actions built using simple, protocol independent primitives;
- Control: last part of program P4, it establishes the flow control of the tables;

The P4 switch architecture defines how the P4 programmable blocks are composed (*e.g.*, parser, ingress control flow, egress control flow, deparser, etc.) and define how to interact with non-programmable elements. P4 programs are written for a specific switch architecture. The `simple_switch` architecture is considered to be the main architecture for most users as it is equivalent to the "abstract switch model" as described in the P4_14 specification. The `v1model` architecture is designed to be identical to the P4_14 switch architecture allowing direct automatic translation of P4_14 programs to P4_16 version programs that use the v1model architecture. In version P4_16, the language is designed so that a number of different architectures can be considered for each different device. In addition, the P4_16 language now also has a Portable Switch Architecture (PSA) defined in its own specification. There is also the `SimpleSumeSwitch` architecture, currently defined for programming a NetFPGA SUME.

BMv2 is the implementation of the Behavioral Model of the PISA architecture, and this one implements a simple switch model on the specific compiler backend: p4c-bm. Once the P4 program has been compiled, the backend will be responsible for programming or configuring the appropriate data plan. For example, in the case of BMv2, the P4 program will be transformed into a JSON description as a backend output. The JSON file, in turn, will be the entry for the BMv2 switch that will make the segmentation of the packages at the entrance, following the guidance of what is specified in the JSON configuration file, and then will populate the reference PISA switch pipeline tables.

The goal of the research work carried out in the scope of this dissertation is the verification of P4 programs to identify potential bug situations. Software verification is in the context of software testing that is described in Section 2.2.

2.2 Software Testing

Software testing is part of a broader process of software verification and validation (V & V). Verification and validation processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the customer. Verification and validation are different processes. Verification is usually a more technical activity to check that the software meets its stated functional and non-functional requirements, checking that we are building the product right. Validation usually depends on domain knowledge to ensure that the software meets the customer's expectations. Aims to validate that we are building the right product.(AMMANN; OFFUTT, 2016)

Software testing is one of the quality assurance activities designed to verify that the product under development meets its specification (DELAMARO M. E.; JINO, 2007). The IEEE Glossary (IEEE, 1990) define Software testing as *"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"*.

Before describe how to detect software defects, we need to have some knowledge of the different kinds of defects to be sought, as defined in Table 2.1 (TRAVASSOS et al., 1999). This taxonomy classifies defects by identifying related sources of information, which are relevant for the system being built.

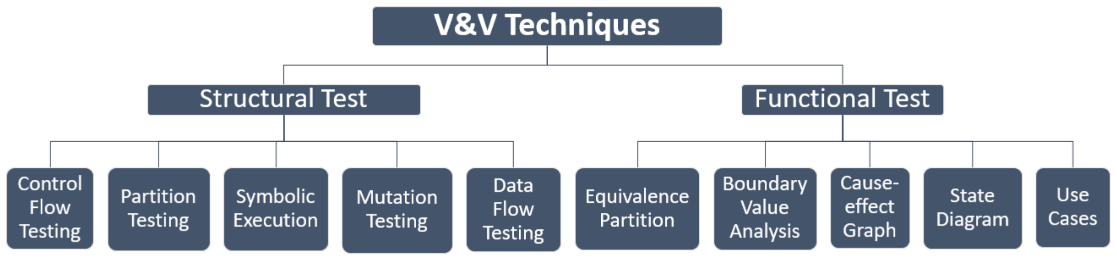
Different types of tests can be performed to verify if a program behaves as specified and, for each type, the definition of test requirements is defined by the type of information used to perform the test. Mainly, there are two types of software testing techniques: structural test and functional test as illustrated in 2.2. Functional testing, also known as black-box testing, assesses the compliance of a system with the specified func-

Table 2.1: Defect Classes (Source: (TRAVASSOS et al., 1999))

Defect Class	General Description
Omission	Necessary information about the system has been omitted from the software
Incorrect Fact	Some information in the software artifact contradicts information in the requirements document or the general domain knowledge
Inconsistency	Information within one part of the software artifact is inconsistent with other information in the software artifact
Ambiguity	Information within the software artifact is ambiguous, i.e. any of a number of interpretations may be derived that should not be the prerogative of the developer
Extraneous Information	Information is provided that is not needed or used

tional requirements, regardless of the internal functioning, the code itself, focusing only on outputs that are generated according to inputs and conditions provided. Unlike functional testing, structural testing, also known as white-box testing, evaluates the internal functioning of the system using the source code for test case generation. In this work, we focus on the use of a structural testing technique.

Figure 2.2: V&V Techniques.



The structural test technique is based on the internal paths, structure and implementation of the program under test, that is, it requires knowledge of the code of the program under test to be applied (AMMANN; OFFUTT, 2016). Each technique is seen as complementary to other existing testing techniques as it covers distinct classes of defects (DELAMARO M. E.; JINO, 2007). The information obtained by applying structural technique can assist in code maintenance and debugging activities (AMMANN; OFFUTT, 2016), because, unlike the functional technique, the test results enable analysis directly related to the source code of the software under test.

Structural test techniques are based on different types of program concepts and components to determine test requirements. The first structural program testing technique used were based solely on the program control flow, the best known being: the *all nodes* criterion; the *all branches* criterion; the *all paths* criterion. The *all nodes* criterion requires that all commands be executed at least once; The second, *all-branch* criterion, one of the most widely used criteria, requires that all transfer of control between command blocks be exercised at least once. The *all paths* criterion, in turn, requires that all possible program paths be exercised (HOWDEN, 1978; WOODWARD M. R.; HENNEL, 1980; KING, 1976).

Subsequently, the technique based on data flow analysis of the program emerged. Data flow analysis (HECHT, 1977) has been widely used for code optimization by compilers and detection of program anomalies through static program analysis. In general, it classifies each occurrence of a variable in the program as a **definition** or as a **use**. These technique therefore use program data flow information to derive test requirements.

The main purpose of introducing data flow based technique is to provide a criteria hierarchy between the all branches and all paths criteria and to make the structural test more rigorous. From the point of view of testing embedded functions in a program, technique that use data flow information to derive requirements and tests are better suited than those that use only control flow information, as the former identifies data dependency and therefore require functional segments (HOWDEN, 1986; URAL; YANG, 1988).

2.3 Data Flow Testing

To detect improper use of data values due to coding mistakes, data flow testing can be used as a structural test criterion (MALDONADO et al., 1991). Rapps and Weyuker proposed Def-Use Graph, which consists of an extension of the Control Flow Graph (CFG) (RAPPS; WEYUKER, 1985). 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 (MALDONADO et al., 1991), 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 base on (RAPPS; WEYUKER, 1985):

- d – defined, initialized
- u – used

Three possibilities exist for the first occurrence of a variable through a program path. The ∼ symbol is used to denote that before this the variable did not exist (RAPPS; WEYUKER, 1985):

1. ∼d – variable does not exist, then it is defined (d)
2. ∼u – variable does not exist, then it is used (u)
3. ∼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 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 (RAPPS; WEYUKER, 1985). Table 2.2 lists usage combinations and corresponding consequences.

Table 2.2: Testing Anomalies (Source: (RAPPS; WEYUKER, 1985))

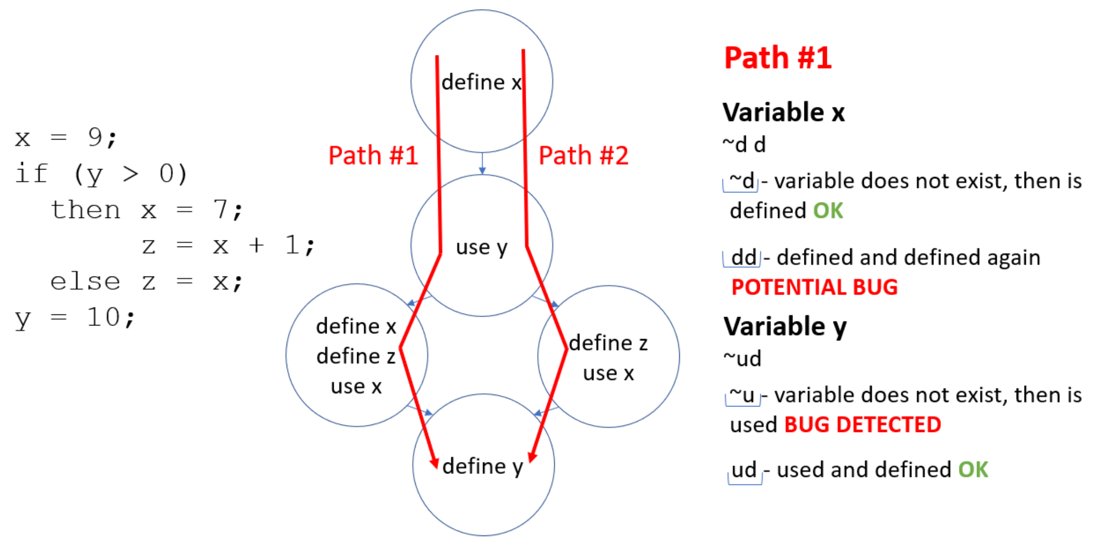
Anomaly		**Explanation**
dd	Defined and defined again	Not invalid but suspicious. Potential bug.
du	Defined and used	Allowed. Normal case.
ud	Used and defined	Allowed.
uu	Used and used again	Allowed.

Identifying an anomaly using data flow testing 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 (FOSDICK; OSTERWEIL, 2011). 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 performed over a variable along the paths mapped. Anomalies in the data stream can be detected by the sequence of definitions and uses that occur with each variable along the way.

The Figure 2.3 shows an example of how data flow analysis is performed. To the left of the image we have an example code. The data flow graph that is generated based on this code is illustrated in the center of the image, that is, the possible paths are mapped and each node has the use (reading) and definition (writing) annotation that happens with each variable. Then, the data flow analysis for each path is performed. We can see on the right side of the image the analysis of the definition and usage sequence that happens for each variable identified in path 1. In pairs, each occurrence of this sequence is analyzed and verified if it can be a possible defect or not, according to what was previously described about the possible first occurrences of the variables and the possibilities of pairs shown in Table 2.2.

Figure 2.3: Example Data Flow Analysis.



```
x = 9;
if (y > 0)
   then x = 7;
        z = x + 1;
   else z = x;
y = 10;
```

Path #1

Variable x

~d d

~d - variable does not exist, then is defined OK

dd - defined and defined again POTENTIAL BUG

Variable y

~ud

~u - variable does not exist, then is used BUG DETECTED

ud - used and defined OK

It is important to note that the data flow test technique is most commonly applied for unit tests, that is, tests of a program unit. However, there are already works that extend the data flow test to integration tests (HORGAN; LONDON, 1991). For different test levels (unitary, integration, regression) different data flow test criteria are applied, which limit in some way the number of paths explored to identify defects. In addition, there are also works that extend the application of this test technique to the testing of OO programs and components. New data flow testing applications can be developed and adapted for different programming languages. For example, procedural language and object-oriented language are much different in the construction of def-use pairs.

3 RELATED WORK

Network verification has shown 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 (LIU et al., 2018), Vera (STOENESCU et al., 2018), and ASSERT-P4 (NEVES et al., 2018)), and dynamic analysis, *e.g.*, p4pktgen (NöTZLI et al., 2018). Static analysis techniques are those that do not involve running the software under evaluation, and may be used to identify defects before an executable version of the system is available (SOMMERVILLE et al., 2007).

In order to identify relevant work in the literature, we conducted a systematic mapping study to capture the state-of-the-art on forwarding plane software verification and validation. We reviewed techniques according to their verification and validation capabilities, and identified research opportunities towards more reliable and secure forwarding plane software.

3.1 Systematic Mapping Study

Systematic Literature Reviews (SLR) and Systematic Mapping Studies (SMS) have been adopted as a research method in mature scientific fields such as Medical and Social Sciences (KITCHENHAM; BUDGEN; BRERETON, 2015). Recently, its use has grown in Computer Science, particularly in Software Engineering. Besides, publications appear in other research areas including Computer Networks (PATEL et al., 2013). The so-called secondary studies (SLR and SMS) benefits from using a research protocol what promotes an unbiased, more auditable, and reproducible review.

With a narrower objective, SLRs aim at synthesizing evidence for a particular research question using results from several primary studies or experiments. On the other side, SMSs aim at producing a broad map of a particular area and informing the existing research and challenges (gaps).

In this dissertation, we report part of an ongoing Systematic Mapping Study effort for the area of programmable data plane, concentrating on V&V issues. In this way, the activities of the study of definition of the research goals and questions, planning the study and executing the review refer to the broader scope of the study which is a map of the program area of the data plane. The study activities of the analysis of findings and reporting detailed in this dissertation refer only to one of the research questions that is related to the verification and validation of P4 programs.

The process for this study follows five activities: (1) definition of the research goals and questions; (2) planning the study, which results in the protocol described in this section; (3) executing the review, encompassing the search and selection of sources, as well as the extraction of information; (4) analysis of findings; and (5) reporting.

As result of the activity (1), the main goal of this study is to *analyze* scientific publications, technical reports, and white papers *for the purpose of* characterizing them, *with respect to* the existing research, opportunities and open challenges on programmable data plane, *from the viewpoint of* researchers *in the context of* academic and industrial reports. More specifically, we aim at classifying and analyzing the literature on programmable data plane to map the phenomenon, providing a solid overview for the state-of-the-art, and investigating the scientific evidence and identify areas suitable for further research.

From this research goal, we defined several research questions, but in this dissertation we concentrate on "*Which are the proposed approaches to verify and validate data plane programs*?".

3.1.1 Search Strategy and Procedure

We performed the automated search using two digital libraries, Scopus and Google Scholar. The first provides a wide-range of scientific peer-reviewed papers (including IEEE and ACM venues), and the latter includes relevant technical reports and white papers. To define the search string, we adopt terms standing for the programmable data planes domain and its applications, according to the procedure below:

- Step 1: Identification of keywords and synonyms for terms used in the research questions.
- Step 2: Formulation of a search string combining the terms through the boolean operators AND (main concepts) and OR (join synonyms).
- Step 3: Compare the search results against the set of 16 control papers, previously identified in an *ad-hoc* literature review. The search string should be able to return the whole set of control papers (100% coverage). Due to distinct features of search engines, the search strings for the different libraries are semantically equivalent, but syntactically different (see Table 3.1).

In the selection procedure, every document should be reviewed by, at least, two

Table 3.1: Search String x Digital Library

Digital Library	Search String	Filters	Criteria
Scopus	TITLE-ABS-KEY ("programmable network*" OR "programmable data plane" OR "data plane language" OR (P4 AND (network* OR language OR program*)))	Title, Abstract, Keywords	All results
Google Scholar	"programmable network*" OR "programmable data plane*" OR "data plane languages" OR (P4 AND (network* OR language OR program*))	All fields	All results until achieving 20 negative results in a row, Use of private navigation, Chrome Plugin (NoCountryRedirect)

reviewers. The third reviewer could be used in case no consensus is reach. The procedure is iterative and the level of agreement among reviewers increases as the number of iterations grow.

3.1.2 Selection Criteria

For selecting relevant literature, we established the following set of *a priori* inclusion (I) and exclusion (E) criteria:

I1 The source is in the area of Networks or Computer Science;

I2 Title and/or abstract has to explicitly mention the programmable data plane;

I3 The source is a journal or conference paper, thesis, technical report or white paper;

E1 The source is not in English;

E2 The source is an editorial or proceedings introduction;

E3 The source is not fully accessible;

E4 Duplicated sources;

After executing the search, we firstly applied these criteria to title, abstract, and keywords. Secondly, on the remaining set, we applied them to introduction. Finally, in the extraction phase, we applied them to the whole document.

3.1.3 Information Extraction

The extraction procedure is organized into rounds, in which every extraction was reviewed by another reviewer. The iterative procedure allowed us to reach a common understanding of what should be extracted for each field. The information extracted from the selected sources include: (1) Paper identification (title, authors, venue, year, and publication type); (2) Addressed research problems; (3) V&V approaches (proposed and/or used) for programmable data plane; (4) Applied research methods (controlled experiments; case studies, simulations, and others); and (5) Open research challenges.

3.1.4 Quality Appraisal

After selecting sources and extracting information, we evaluate the quality of sources using the criteria presented in Table 3.2. Thus, we discuss quality of sources in terms of Rigor (R) of the research and Practical Relevance (P) of the proposed solution.

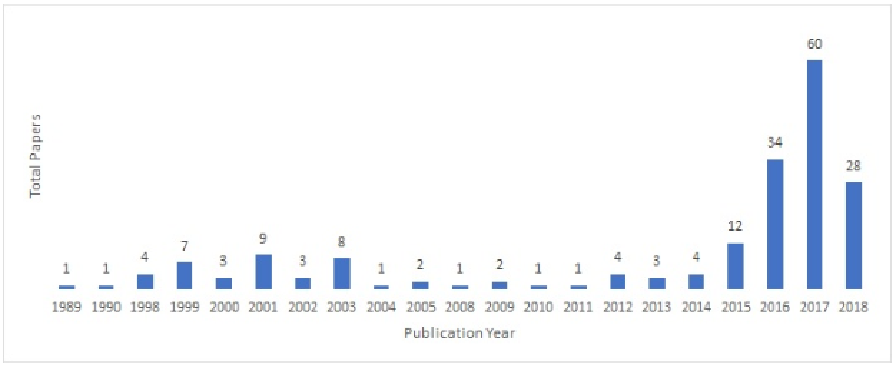
Table 3.2: Quality Assessment Criteria

ID	**Criteria**	**Score**
R1	Mathematical foundation: use of analytic modeling and/or mathematical proofs	1pt
R2	Simulation targeting performance evaluation and scale	1pt
P3	Prototype or product provided	1pt
P4	Analysis in real-life environment	1pt
P5	Workload of real-life users	1pt

3.2 Results

We executed the searches in April 2018. From the 2.297 retrieved sources (Scopus, Google Scolar and *ad-hoc* review), we excluded 144 duplicated ones. In the selection stage, we considered 189 papers as relevant after a careful review applying the selection criteria. Figure 3.1 shows the distribution of studies per year. So, it is possible to notice that selected publications start in 1989. Until 2014, papers concerning programmable data plane address technologies like Active Networks, ForCES, and FPGA implementations. From 2015, the number of studies gradually increases, probably induced by the seminal paper of the P4 language (BOSSHART et al., 2014).

Figure 3.1: Distribution per year



In order to answer the research question regarding V&V approaches for data plane programs, we identified only seven (out of 189) papers containing verification approaches as presented in Table 3.3.

Table 3.3: Verification Approaches

Verification Approach	**#Sources**	**References**
Model Checking	1	(LOPES et al., 2015)
Symbolic Execution	6	(DOBRESCU; ARGYRAKI, 2014) (FREIRE et al., 2017) (FREIRE et al., 2018) (STOENESCU et al., 2018) (LIU et al., 2018) (NöTZLI et al., 2018)

Symbolic execution and model checking are static analysis techniques. Static analysis techniques are system verification techniques that do not involve executing the program. Rather, they work on a source representation of the software either a model of the specification or design, or the source code of the program. Static analysis techniques can be used to check the specification and design models of a system to pick up defects before an executable version of the system is available. (SOMMERVILLE et al., 2007).

The model checking process involves building a formal model of a system. A set of desirable system properties are identified and written in a formal notation. The model checker then explores all paths through the model (*i.e.*, all possible state transitions), checking if the property holds for each path. If it does, then the model checker confirms that the model is correct with respect to that property. If it does not hold for a particular path, the model checker outputs a counter-example illustrating where the property is not satisfied.

Network Optimized Datalog (NoD) (LOPES et al., 2015) is a work in the category of static analysis that uses model check. 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.

Model checking is computationally very expensive because it uses an exhaustive approach to check all paths through the system model. As the size of the system increases, the number of states also increases, and a consequent increase in the number of paths to check is expected. This means that, for large systems, model checking may be impractical, due to the computer time required to run the checks. One could provide more abstract models, however they would not allow in-depth analysis.

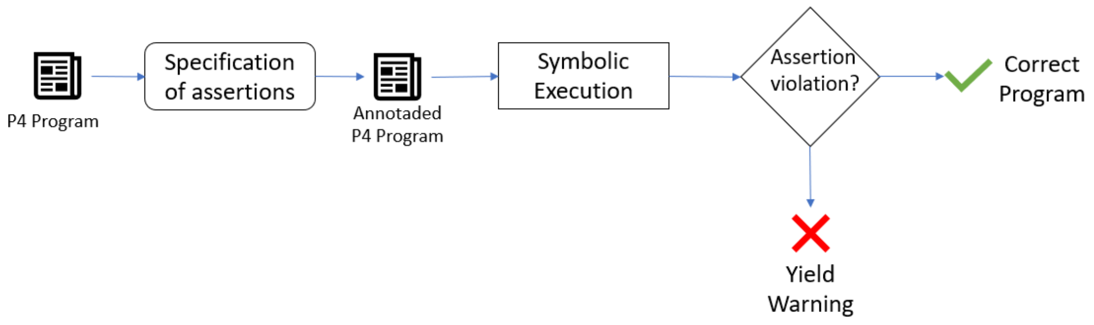
Another test strategy is symbolic execution. The key idea behind symbolic execution (KING, 1976; CLARKE, 1976) is to use symbolic values instead of concrete data as input and to represent the values of program variables as symbolic expressions over the symbolic input values. As a result, the output values computed by a program are expressed as functions of the symbolic input values. In software testing, symbolic execution is used to generate a test input for each execution path of a program. An execution path is a sequence of true and false, where a value of true (respectively false) at the i^{th} position in the sequence denotes that the i^{th} conditional statement encountered along the execution path took the “then” (respectively the “else”) branch. All the execution paths of a program can be represented using a tree, called the execution tree.

Dobrescu and Argyraki (DOBRESCU; ARGYRAKI, 2014) have used symbolic execution with S2e (CHIPOUNOV; KUZNETSOV; CANDEA, 2011) to analyze implementations of Click modular router elements (KOHLER et al., 2000). One of the key challenges of symbolic execution is the huge number of programs paths in all but the smallest programs, which is usually exponential in the number of static branches in the code. As a result, given a fixed time budget, it is critical to explore the most relevant paths first (CADAR; SEN, 2013). First of all, symbolic execution implicitly filters out all paths which (1) do not depend on the symbolic input, and (2) are unreachable given the current path constraints. Despite this filtering, path explosion represents one of the biggest challenges when adopting symbolic execution for general-purpose languages.

There are also solutions that rely on code annotations for static verification. Assertion testing is the insertion of formal statements into the program to be validated, as if they were code comments (SOMMERVILLE et al., 2007). The expression is formally presented as an assertion, along with some form of identifier, to help testers and engineers ensure that tests of the target relate properly and clearly to the corresponding specified statements about the target. For example, an assertion might be included stating that the value of some variable must lie in the range x..y. The analyzer symbolically executes the code and highlights statements where the assertion may not hold.

ASSERT-P4 (NEVES et al., 2018) 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 (LIU et al., 2018) also proposes that developers annotate programs with Hoare logic clause (pre and post conditions) to enable static verification. In Fig. 3.2 we can see an overview of how this technique works.

Figure 3.2: Assertion overview (Adapted from (NEVES et al., 2018))



Assertion test technique can be tailored to check for well-known problems, however it is not efficient to identify defects that we have no prior knowledge. In order to be able to perform assertion testing, it is necessary for the programmer to know in advance the defect he or she seeks to identify, so that he or she can annotate the code with the assertion and thus possibly identify the defect.

Both ASSERT-P4 and p4v demand additional developer effort to express the aspects that must be verified (assertions), an error-prone approach. More importantly, if some critical network property is not specified (omitted) through an assertion in the code, it will be left unverified.

On the other hand, Vera (STOENESCU et al., 2018) is another static verification tool that uses symbolic execution to test the behavior of P4 programs. For this, it checks a snapshot of a running P4 program and, unlike p4v and ASSERT-P4, it does not require annotations. A snapshot represents the full state of all the match and action tables manipulated by a P4 program in a given moment in time. This tool uses the parser of the P4 program and a snapshot of all its table rules to generate all parsable packet layouts (e.g. header combinations), and makes all header fields symbolic (i.e. they can take any value). It then tracks the way these packets are processed by the program, following all branches to completion.

Vera can do a comprehensive set of tests for any rule snapshot in the table, but a single snapshot does not cover all the possible rules that can be inserted. Even if a snapshot of a P4 program is bug free, there is no guarantee that it will be true for other snapshots. It is impractical to perform sample-space testing of all possible rule table snapshots. In addition, the match-action table changes all the time, making snapshots that have been tested no longer valid. In order to be able to identify difficult defects with Vera, it is necessary for the programmer to be very experienced and to choose carefully the rules that will compose a snapshot, otherwise the defect path will not be executed making this approach not scalable.

In the realm of dynamic analysis, p4pktgen (NöTZLI et al., 2018) is a tool for generate test packets. It also uses symbolic execution to generate test packets and predict the expected output using *bmv2* for assessing the behavior of a P4 program. This approach may even uncover bugs in the compiler and the software switch. In addition, on the downside, it requires switch deployment for testing and may not scale well for large-scale and complex programs.

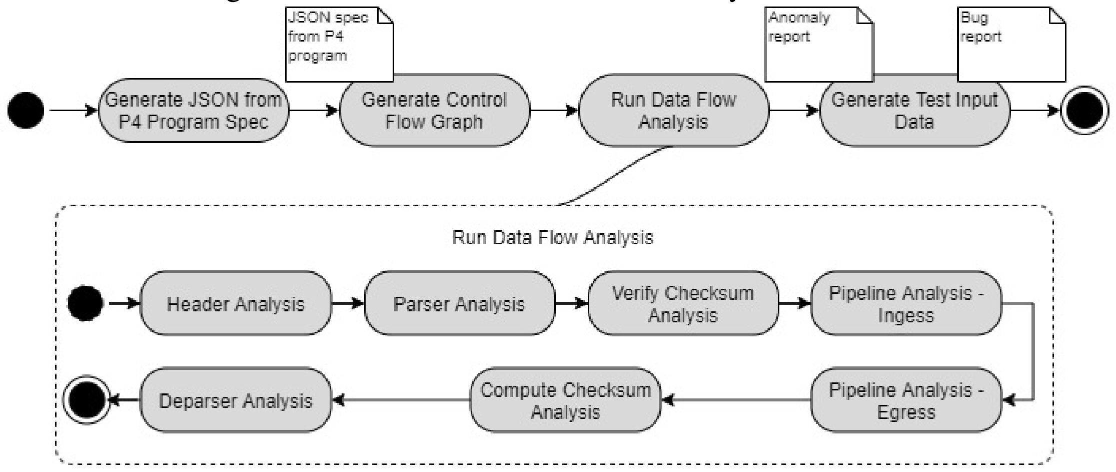
The different software verification and validation techniques are complementary. Under no circumstances they should be regarded as redundant activities. Both have different natures and objectives, capturing different classes of defects, strengthening the failure detection process and increasing the resulting software quality.

From this systematic review, it was possible to identify that the existing works of verification and validation of P4 programs explore in different ways the techniques of model checking and symbolic execution. An important disadvantage common to these verification techniques is that they cannot cover the omission class of defects unless additional code is written by the developer for assertion-based solutions. One testing technique not yet explored for P4 program verification is Data Flow Analysis, which covers the identification of some defects classified as omission and incorrect fact. Next chapter introduces a solution that uses the Data Flow Analysis technique to verify P4 programs.

4 P4 DATA FLOW ANALYSIS

This chapter explains the P4-DATA-FLOW verification process. We present our approach for using data flow analysis to uncover bugs in P4 switch programs, using Fig. 4.1 as basis. We describe in Section 4.1 how JSON is generated based on a P4 program and how the control flow graph is generated. In Section 4.2 is described how data flow analysis is performed and finally, the generation of data to perform the test to confirm the defect is described in Section 4.3.

Figure 4.1: P4 switch code data flow analysis overview.


JSON spec from P4 program
Anomaly report
Bug report
Generate JSON from P4 Program Spec
Generate Control Flow Graph
Run Data Flow Analysis
Generate Test Input Data
Run Data Flow Analysis
Header Analysis
Parser Analysis
Verify Checksum Analysis
Pipeline Analysis - Ingess
Pipeline Analysis - Egress
Compute Checksum Analysis
Deparser Analysis

4.1 Solution Overview

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, a software switch model popularly used to evaluate the functionalities of a P4 program specification. To generate the JSON file the following line of code is used: "`p4c-bm2-ss -p4v 16 -p4runtime-files build/SOURCE.p4.p4info.txt -o build/OUTPUT.json SOURCE.p4`", where SOURCE refers to the file name P4 and OUTPUT the name of the output JSON file.

Implementations of P4 programs that are in version P4_14 need to be converted to version P4_16. For that we use the P4 p4test tool to convert P4_14 codes to P4_16.

On the left side of the Fig. 4.2 it is possible to see a code snippet basic.p4 and on the right side of the Fig. 4.2 it is possible to check the same code snippet generated in JSON.

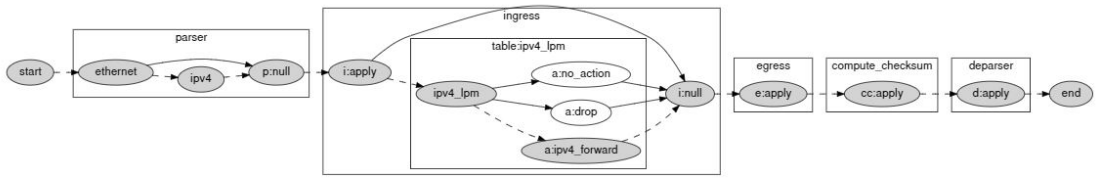
Figure 4.2: basic.p4 switch code snippet

```
/* -- P4_16 -- */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;


/************************
***** H E A D E R S *****
************************/


typedef bit<9> egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
   macAddr_t dstAddr;
   macAddr_t srcAddr;
   bit<16> etherType;
}
```

```
{
  "program" : "basic.p4",
  "_meta_" : {
    "version" : [2, 7],
    "compiler" : "https://github.com/
        ↪ p4lang/p4c"
  },
  "header_types" : [
    {
      "name" : "scalars_0",
      "id" : 0,
      "fields" : []
    },
    {
      "name" : "ethernet_t",
      "id" : 1,
      "fields" : [
        ["dstAddr", 48, false],
        ["srcAddr", 48, false],
        ["etherType", 16, false]
      ]
    },
```

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. 4.3, from the basic.p4 switch code[1], shows eight possible execution paths.

Figure 4.3: Control flow graph for the basic.p4 switch program.



For each possible path, we run data flow analysis. This process (detailed in the lower part of Fig. 4.1) 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. As a report with data flow analysis of all variables in all paths can be impractical to analyze manually, the argument `-s` can be used in the line of execution and thus, the issued report returns only the paths where they were cases of possible bugs were identified, that is, variables that were used without first being written or variables that had two consecutive writes.

The Fig. 4.4 shows the path expressions obtained for the execution path highlighted in dotted lines and gray ellipses in Fig. 4.3. As an example, take the path ex-

[1]The 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>

Figure 4.4: An example of path expressions obtained for the basic.p4 switch code, for the path highlighted in Fig. 4.3. 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.

```
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
```

pression of field `ipv4.ttl:  DUDUU` highlighted in red in Fig. 4.4. 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 designed 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. However, using the same methodology that is being proposed it is possible to implement a functional solution for any of the existing architectures.

4.2 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. 4.1. The process carried out in each task is described in the algorithms discussed next.

All algorithms were developed based on the behavioral model of the JSON format[2] that shows the specification of the JSON fields. 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.

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
```

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 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:

- **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,

[2]The JSON behavioral model can be found at <https://github.com/p4lang/behavioral-model/blob/master/docs/JSON_format.md>

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
```

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.

After processing the parser, we proceed to the analysis of the verify checksum control. 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.

Following the pipeline depicted in the lower part of Fig. 4.1, we proceed to the egress analysis, whose routine is depicted in Algorithm 4. The algorithm is very similar to the ingress algorithm, where the only difference is that the routine starts by fetching the egress in the JSON pipeline declaration (line 1).

After the egress analysis occurs the compute checksum control, whose 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 5. In summary, for each deparser declaration in the JSON file, we process it by updating the `DF_Table` with each usage definition found.

4.3 Test Packet Generation

The output of the data flow analysis step is a list of potential bugs. Potential bug means that it has the capacity to become a bug in the future, after confirmation. Thus, the last step of the proposal is to check for these potential bugs in order to confirm if they are really bugs in the program.

For exercising such cases, we use the P4 packet test framework (PTF)[3]. We use

[3]PTF GitHub repo: <https://github.com/p4lang/ptf/>

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
```

Algorithm 4 Egress Analysis

```
 1: Fetch egress 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
```

Algorithm 5 Deparser Analysis

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

this framework to confirm the possible bugs we have identified, but this framework is not part of our proposal. 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 structural 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.

Among all stages of the P4 switch code data flow analysis, illustrated in the Fig. 4.1, this generate test input data step is the only step that has manual effort to generate the package using PTF and after validating if the possible bug is really confirmed. All previous steps of the proposal are carried out automatically by our tool.

About test packet generation, our current focus is exploring data flow analysis for P4 program verification. We are investigating how to determine heuristics to automate test case generation based on the outcome of potential bugs, possibly using Header Space Analysis (HSA) or symbolic execution.

5 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 GitHub[1]. In our evaluation, we considered many switch implementations publicly available, mainly from the official P4 repo[2]. In Table 5.1, 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.

Table 5.1: 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

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. 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.

5.1 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 implementation. Fig. 5.1 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

[1]P4-DATA-FLOW GitHub repo: <https://github.com/diogocampos/p4-data-flow/>

[2]Official P4 GitHub repo: <https://github.com/p4lang/>

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 specification[3], 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. 5.1 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 apply in the `ipv4_lpm` table.

5.2 Load Balance

Our second experiment is the load_balance switch from the official P4 tutorial. In Fig. 5.2 we present one execution path with a faulty code. Before applying the table `ecmp_group`, 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 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. 5.2, 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.

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

Figure 5.1: Execution paths with faulty behavior in the simple_nat switch implementation.

```
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_nhop -> 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: UUDUU
      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_nhop -> 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
```

Figure 5.2: Faulty behavior in the load_balance switch code.

```
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
```

Figure 5.3: Faulty behavior in the flowlet switching code.

```
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: UDUUU
      protocol: UUU
      hdrChecksum: DDU
      srcAddr: UUU
      dstAddr: UUUU
```

5.3 Flowlet Switching and Checksum P4

We also evaluated the flowlet switching (SINHA; KANDULA; KATABI, 2004) and checksum implementations available in the official P4 GitHub repo. In the flowlet switching execution path shown in Fig. 5.3, there are path expressions in which ipv4 and tcp header fields are used but never defined.

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.

5.4 Comparative Analysis

In this section we describe the results of the comparison carried out on the types of defects identified by the existing P4 code verification tools and P4-DATA-FLOW.

Assertion-based verification requires previous knowledge on the types of defect. By understanding how each defect type can manifest in a P4 code excerpt, P4 developers can correctly implement the assertion to identify the presence of the defect instance in the verified code. The lack of such knowledge implies on the lack of such assertion. Thus, this verification approach cannot be fairly compared with automated verification techniques such as data flow analysis. In other words, assertions need human intervention, with developers deliberately inserting additional code (the assertion itself). On the other hand, automated verification based on static analysis does not. Finally, assertions also need to be verified, as developers could insert defects, by mistake, in the assertion code. In this respect P4-DATA-FLOW has an advantage as it identifies defects in P4 codes without having any prior knowledge of existing defects.

Vera (STOENESCU et al., 2018) uses snapshot verification without the need to insert assertions to identify defects. As it is not open for community use, we contacted the authors and asked them to make the Vera tool source code available so that we could run it on the same P4 programs verified with P4-DATA-FLOW so that we could compare the captured defects. In addition, we requested the P4 programs they use to evaluate their tool, as the currently published codes are no longer in the same version that was used in their validation. After several attempts, we received no feedback. So, the quantitative comparison of the Vera tool with P4-DATA-FLOW could not be performed.

For the reasons above, we perform a qualitative analysis to carry out this comparison between the defects identified by each work, as can be seen in the Table 5.2. It analyzes whether the defects identified by the related works are likely to be identified by our tool and vice-versa.

5.4.1 Comparison with ASSERT-P4

First, we performed a qualitative analysis of the defects identified by ASSERT-P4 in order to assess whether our tool, P4-DATA-FLOW, would also be able to identify the same defects. ASSERT-P4 (NEVES et al., 2018) performed tests on four P4 program codes. The first code used for testing is the Dapper implementation (GHASEMI; BENSON; REXFORD, 2017). The defect identified in the Dapper implementation regards the verification if the TTL field is greater than zero for the packet to be forwarded. P4-DATA-FLOW may not identify it because its strategy concerns only defects in the data flow, not taking actual values of variables into account. The second test was performed on the NetPaxos (DANG et al., 2016) code that is a network-based implementation of the Paxos consensus protocol. Regarding the defect identified in the implementation of NetPaxos, valid packets being dropped because the packets are first marked to be dropped by another action, and not unmarked by the voting actions. Our solution also does not identify this defect because there is no violation of the data flow in this situation, such

Table 5.2: Qualitative Analysis

Tool	**Verification Mechanism**	**Program**	**Defect**	**P4-DATA-FLOW catches?**
Assert-P4	Assertion	Dapper	TTL field is greater than zero	No
		Netpaxos	Valid packets being dropped	No
		Switch	Modification of a field of an invalid header	Yes
			Tunnel encapsulation	Yes
p4v	Assertion	Switch	Parsed packets not supported by the rest of the pipeline	Yes
			Order-of-operations error	Yes
			Tables that incorrectly allowed the nop action	No
			Erroneously read the inner_ethernet header	No
			Multi-table constraints	No
		NetCache	Implementation of the put operation	No
		Netpaxos	Valid packets being dropped	No
Vera	Snapshots	Switch	Implicit drops	No
			Table rules that match dropped packets	No
			Invalid memory accesses	Yes
			Header errors	Partially
			Scoping and unallowed writes	No
			Out-of-bounds array accesses	Yes
			Field overflows/underflows	No

as reading a variable without ever reading it or two consecutive writes of a variable. A third test was performed with ASSERT-P4 in the DC.p4 (SIVARAMAN et al., 2015) code that implements the behavior of a data center switch. In this test, the authors did not identify any defects, they only checked that the L3 ACL only flags packets to be filtered by another module in the system, which must also be appropriately configured. The fourth and last test performed was in the code of the Switch (P4..., 2018) program. In this code, the authors sought to identify two known defects by inserting assertions. The first defect founded regards the modification of a field of an invalid header. The second defect regards tunnel encapsulation, where encapsulated headers are overwritten whenever multiple nested levels are present. In our qualitative analysis, we understand that P4-DATA-FLOW can capture the same two defects. As for both defects there are two consecutive *write* operations in a same variable without a *read* operation before the second *write*.

After this stage of the qualitative analysis, where we compared the defects identified by ASSERT-P4, we performed the analysis of the defects identified by our tool, P4-DATA-FLOW, in order to verify whether ASSERT-P4 would also be able to identify the same defects. As ASSERT-P4 is a tool that uses assertions to identify defects, it is able to identify all defects identified in our work, as long as we have prior knowledge of the defect and that the developer correctly includes the assertion of the P4 code to identify the same defect.

5.4.2 Comparison with p4v

Qualitatively comparing P4-DATA-FLOW against p4v (LIU et al., 2018), the first case study reported refers to general safety property in switch.p4, which should never access a field of an invalid header. To validate this property, assertions were inserted before each writing and reading of a header field that checks whether the corresponding header instance is valid at that program point. They identified ten defects in switch.p4. Two were parser defects, which parsed packets not supported by the rest of the pipeline. Our solution is able to identify these defects as it also reads the variable when the packet is parsed without having previously written through the packet extraction, violating the data flow rule where every variable must first be written before being read. The other defects identified by p4v in this case study related to order-of-operations error, in which fields were modified in a header before the header was added. P4-DATA-FLOW can also identify this defect by the same criterion of reading without writing before. Still in the same case study, two defects were identified in tables that incorrectly allowed the *nop* action to be taken, one defect was in the actions for terminating L3 MPLS tunnels, which erroneously read the inner_ethernet header. Three defects correspond to multi-table constraints that the designers of switch.p4 believe hold, but do not see how to enforce using the control plane. We were unable to identify these defects because there is no violation of the data flow rules and, therefore, it is not within the scope of our solution.

In the second case study carried out by p4v (LIU et al., 2018), an architectural property for NetCache (JIN et al., 2017) was verified, a program that implements an in-network key value store on a P4-programmable target. They used p4v to automatically insert assertions into NetCache to check that the information in each header is correctly preserved when processed using the deparser and the parser. The defect identified in this program is related to the P4 implementation of the put operation, where the code correctly writes the value in the state registers and invalidates the optional header, but fails to update the operational code. With this, the output analyzer tries to analyze the optional value header again and the transition to an error state can occur and the packet can be discarded. Our tool cannot identify this defect because the variables are being written and read in the correct sequence according to the data flow, but the value being stored is incorrect. This validation is not within the scope of our solution.

P4v (LIU et al., 2018) also carried out a third case study to reproduce the same defect discovered by the developers of P4-ASSERT (NEVES et al., 2018), in NetPaxoss (DANG et al., 2016) program, in which the action that compares the round number from the arriving packet with the round number stored at the switch sets the drop flag of the arriving packet by default, under the assumption that the packet should be dropped. In this case, an incorrect packet drop definition occurs, that is, a variable is written with the wrong information. Our solution P4-DATA-FLOW would not be able to identify this defect as there is no violation of data flow rules, such as reading a variable without it having ever been read or two consecutive writes of a variable.

The second step in the qualitative analysis of the defects identified by p4v (LIU et al., 2018) was to verify whether p4v would also be able to identify the same defects identified by our P4-DATA-FLOW tool. In this sense, p4v would be able to identify the same defects identified in our work as long as the necessary assertions were inserted to exercise these defects. She would not be able to identify the defects without prior knowledge of their existence.

5.4.3 Comparison with Vera

We also performed a qualitative comparison if P4-DATA-FLOW would be able to identify the same defects identified by Vera (STOENESCU et al., 2018). Seven defects identified by the Vera tool have been reported. The first one is related to implicit drops, which happens when a packet reaches the buffer mechanism without having a defined egress_spec. They are able to identify this defect because they have included a validation if egress_spec is different from zero when it arrives at the buffer.in port. The second defect identified by Vera refers to the rules of the table that match dropped packets. They are able to identify this defect as they have entered an assertion to mark as error 511 when this situation occurs. P4-DATA-FLOW cannot identify either of these two defects because we do not validate the value that is stored in the variables, but if the flow of writing and reading of the variables is correct.

The third defect that Vera identified deals with invalid memory accesses. This defect occurs when a header field that has not been declared is accessed. Vera is able to identify this defect in Symnet's (STOENESCU et al., 2016) memory safe mode, where when accessing an unallocated field Symnet fails in the current path. P4-DATA-FLOW is also able to identify this same defect, as there is a violation of the data flow in this header field, since there is a reading without writing before.

The fourth defect reported by Vera identifies header errors, that is, malformed headers. They are able to identify these defects during the analysis using the existing SEFL (STOENESCU et al., 2016) instruction. The fifth defect refers to scoping and unallowed writes, where some metadata values are read-only in P4, but the P4 compiler allows the program to write them or values that can only be read from a table according to specifications, but the compiler allows for these readings. Vera identifies this type of defect during the SEFL translation. It is not in the scope of our P4-DATA-FLOW tool to identify these two types of defects, as we do not validate the values stored in the variables to identify whether a header is well formed or to validate if any values are parameterized to be read-only.

The sixth type of defect identified by Vera is related to out-of-bounds array accesses. They are able to identify these defects by adding, before each access to the matrix, an out-of-bounds check for the index. We were able to identify these defects also using the methodology we are proposing in this work, since in these situations there is a read in fields that have not been written before, thus violating a data flow rule. The seventh and final defect reported by Vera deals with field overflows and underflows which are the only possible arithmetic exceptions in P4 (because the division is not supported) and Vera captures them by adding a check before each addition / subtraction operation. Our tool is unable to identify this type of defect because this is another case where the value stored in the variable is being verified and this is not in our scope of defects.

The other way of qualitative comparison of defects was also carried out, where the Vera tool's ability to identify the same defects identified by our P4-DATA-FLOW solution was verified. As previously mentioned, the Vera tool uses the snapshot mechanism to perform the verification of P4 programs, that is, this tool uses the parser of the P4 program and a snapshot of all its table rules to generate all parsable packet layouts (e.g. header combinations), and makes all header fields symbolic (i.e. they can take any value). Vera can do exhaustive tests for any rule snapshot in the table, but one snapshot does not cover all the possible rules that can be inserted. Vera is able to identify the same defects identified by P4-DATA-FLOW, as long as the rules that make up the snapshot exercise the path of the defect to be identified. If the selected snapshot does not have the correct rules, Vera is unable to identify the defects identified by our tool.

5.5 Discussion on Limitations and Applicability

Our solution has shown to scale well with the number of execution paths within the switch program, as evidenced in the experiments considered. However, and similarly to Vera (STOENESCU et al., 2018), higher verification times could be observed for "branchier" codes, and therefore optimization strategies are required for such cases. The validation of the switch.p4 and DC.p4 programs with our tool did not run until the end because they are code with many branches. As far as we were able to explore the paths, no bugs were identified. Vera (STOENESCU et al., 2018) takes between 5s-15s to track the execution of a purely symbolic packet while our solution took 1s to verify an entire P4 program. 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 (STOENESCU et al., 2018) and p4v (LIU et al., 2018).

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 scalability.

Another limitation of our prototype is that the impact of some P4 primitives like resubmit and recirculate are not yet analyzed, and further analysis of the codes evaluated is still required.

6 CONCLUSION

The possibility of defining switch behavior brought by programmable forwarding planes demands novel approaches to switch development, verification and validation. In this dissertation, 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 generalized to address switch programs written in other domain specific languages, like POF (SONG, 2013).

It is important to emphasize that the testing techniques should be seen as complementary and the question that arises is how to use them so that the advantages of each are better explored in a testing strategy that leads to a effective testing activity. Thus, our approach does not replace, but complements existing work in the field, like Vera (STOENESCU et al., 2018), p4v (LIU et al., 2018), and ASSERT-P4 (NEVES et al., 2018). The approaches are complementary so that together they are likely to discover a higher proportion of defects than would be found using one technique on its own. 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 direction 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 structural testing for potential but unconfirmed bugs after analysis.

REFERENCES

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. [S.l.]: Cambridge University Press, 2016.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM Computer Communication Review**, ACM, v. 44, n. 3, p. 87–95, 2014.

BOSSHART, P. et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In: **Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM**. New York, NY, USA: Association for Computing Machinery, 2013. (SIGCOMM '13), p. 99–110. ISBN 9781450320566. Available from Internet: <https://doi.org/10.1145/2486001.2486011>. Accessed in: 30 Jan. 2020.

CADAR, C.; SEN, K. Symbolic execution for software testing: three decades later. **Communications of the ACM**, ACM, v. 56, n. 2, p. 82–90, 2013.

CAREY, S. Why a single failed router can ground a thousand flights. **The Wall Street Journal**, 2017. Available from Internet: <https://www.wsj.com/articles/why-a-single-failed-router-can-ground-a-thousand-flights-1489743001>. Accessed in: 30 Jan. 2020.

CHIPOUNOV, V.; KUZNETSOV, V.; CANDEA, G. S2e: A platform for in-vivo multi-path analysis of software systems. **Acm Sigplan Notices**, ACM, v. 46, n. 3, p. 265–278, 2011.

CLARKE, L. A. A program testing system. In: ACM. **Proceedings of the 1976 annual conference**. [S.l.], 1976. p. 488–491.

CORDEIRO, W.; MARQUES, J.; GASPARY, L. Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management. **J. Netw. Syst. Manage.**, Plenum Press, New York, NY, USA, v. 25, n. 4, p. 784–818, oct. 2017. ISSN 1064-7570.

DANG, H. T. et al. Paxos made switch-y. **Proceedings of the Symposium on SDN Research**, v. 46, n. 2, p. 18–24, 2016.

DELAMARO M. E., J. C. M.; JINO, M. **Introdução ao Teste de Software**. [S.l.]: Elsevier Editora Ltda, 2007.

DOBRESCU, M.; ARGYRAKI, K. Software dataplane verification. In: **NSDI 14**. [S.l.: s.n.], 2014. p. 101–114.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The road to sdn: An intellectual history of programmable networks. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 44, n. 2, p. 87–98, abr. 2014. ISSN 0146-4833.

FOSDICK, L. D.; OSTERWEIL, L. J. Data flow analysis in software reliability. In: **Engineering of Software**. [S.l.]: Springer, 2011. p. 49–85.

FREIRE, L. et al. Uncovering bugs in p4 programs with assertion-based verification. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: ACM, 2018. (SOSR '18), p. 4:1–4:7. ISBN 978-1-4503-5664-0. Available from Internet: <http://doi.acm.org/10.1145/3185467.3185499>. Accessed in: 30 Jan. 2020.

FREIRE, L. et al. Poster: Finding vulnerabilities in p4 programs with assertion-based verification. In: ACM. **Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security**. [S.l.], 2017. p. 2495–2497.

GHASEMI, M.; BENSON, T.; REXFORD, J. Dapper: Data plane performance diagnosis of tcp. **Proceedings of the Symposium on SDN Research**, p. 61–74, 2017.

HALEPLIDIS, E. et al. **Software-Defined Networking (SDN): Layers and Architecture Terminology**. [S.l.]: RFC Editor, 2015. RFC 7426. (Request for Comments, 7426). Available from Internet: <https://rfc-editor.org/rfc/rfc7426.txt>. Accessed in: 30 Jan. 2020.

HECHT, M. S. **Flow Analysis of Computer Programs**. [S.l.]: Elsevier Science Inc., 1977.

HORGAN, J. R.; LONDON, S. Data flow coverage and the c language. In: **Proceedings of the symposium on Testing, analysis, and verification**. [S.l.: s.n.], 1991. p. 87–97.

HOWDEN, W. E. An evaluation of the effectiveness of symbolic testing. In: **Software: Practice and Experience 8.4**. [S.l.: s.n.], 1978. p. 381–397.

HOWDEN, W. E. **Functional Program Testing and Analysis**. [S.l.]: McGraw-Hill Inc., 1986.

IEEE. IEEE standard glossary of software engineering terminology. **IEEE Std 610.12-1990**, dec. 1990.

JIN, X. et al. Netcache: Balancing key-value stores with fast in-network caching. **Proceedings of the 26th Symposium on Operating Systems Principles**, p. 121–136, 2017.

KAZEMIAN, P. Network path not found? **Forward Networks Blog**, February 2017. Available from Internet: <https://www.forwardnetworks.com/network-path-not-found-how-to-use-an-army-of-tests-to-understand-and-diagnose-your-network/>. Accessed in: 30 Jan. 2020.

KING, J. C. Symbolic execution and program testing. **Communications of the ACM**, ACM, v. 19, n. 7, p. 385–394, 1976.

KITCHENHAM, B.; BUDGEN, D.; BRERETON, P. **Evidence-based software engineering and systematic reviews**. 1st. ed. [S.l.]: CRC Press, 2015.

KOHLER, E. et al. The click modular router. **ACM Transactions on Computer Systems (TOCS)**, ACM, v. 18, n. 3, p. 263–297, 2000.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, v. 103, n. 1, p. 14–76, Jan 2015. ISSN 0018-9219.

LIU, J. et al. P4v: Practical verification for programmable data planes. In: **ACM SIGCOMM 2018**. New York, NY, USA: ACM, 2018. p. 490–503. ISBN 978-1-4503-5567-4.

LOPES, N. et al. Automatically verifying reachability and well-formedness in p4 networks. **Technical Report, Tech. Rep**, 2016.

LOPES, N. P. et al. Checking beliefs in dynamic networks. In: **NSDI**. [S.l.: s.n.], 2015. p. 499–512.

MALDONADO, J. C. et al. Critérios potenciais usos: Uma contribuição ao teste estrutural de software. **(Publicação FEE)**, [sn], 1991.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/1355734.1355746>. Accessed in: 30 Jan. 2020.

NEVES, M. et al. Verification of p4 programs in feasible time using assertions. In: **CoNEXT '18**. New York, NY, USA: ACM, 2018. p. 73–85. ISBN 978-1-4503-6080-7.

NöTZLI, A. et al. P4pktgen: Automated test case generation for p4 programs. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: ACM, 2018. (SOSR '18), p. 5:1–5:7. ISBN 978-1-4503-5664-0. Available from Internet: <http://doi.acm.org/10.1145/3185467.3185497>. Accessed in: 30 Jan. 2020.

P4.ORG language consortium. Switch. 2018. Available from Internet: <https://github.com/p4lang/switch>. Accessed in: 30 Jan. 2020.

PATEL, A. et al. An intrusion detection and prevention system in cloud computing: A systematic review. **Journal of Network and Computer Applications**, v. 36, n. 1, p. 25 – 41, 2013. ISSN 1084-8045.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 11, n. 4, p. 367–375, abr. 1985. ISSN 0098-5589. Available from Internet: <http://dx.doi.org/10.1109/TSE.1985.232226>. Accessed in: 30 Jan. 2020.

SHUKLA, A. et al. Runtime verification of p4 switches with reinforcement learning. In: **Proceedings of the 2019 Workshop on Network Meets AI & ML**. [S.l.: s.n.], 2019. p. 1–7.

SINHA, S.; KANDULA, S.; KATABI, D. Harnessing tcp's burstiness with flowlet switching. In: ACM. **Hot Topics in Networks (HotNets)**. [S.l.], 2004.

SIVARAMAN, A. et al. Dc. p4: Programming the forwarding plane of a data-center switch. **Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research**, p. 1–8, 2015.

SOMMERVILLE, I. et al. **Software engineering**. [S.l.]: Addison-wesley, 2007.

SONG, H. 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**. New York, NY, USA: ACM, 2013. (HotSDN '13), p. 127–132. ISBN 978-1-4503-2178-5.

STOENESCU, R. et al. Debugging p4 programs with vera. In: **ACM SIGCOMM 2018**. New York, NY, USA: ACM, 2018. p. 518–532. ISBN 978-1-4503-5567-4.

STOENESCU, R. et al. Symnet: Scalable symbolic execution for modern networks. In: ACM. **Proceedings of the 2016 ACM SIGCOMM Conference**. [S.l.], 2016. p. 314–327.

TRAVASSOS, G. et al. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 34, n. 10, p. 47–56, oct. 1999. ISSN 0362-1340.

URAL, H.; YANG, B. A structural test selection criterion. **Information processing letters**, v. 28, n. 3, p. 157–163, 1988.

WOODWARD M. R., H. D.; HENNEL, M. A. Experience with path analysis and testing of programs. **IEEE Trans. on Software Eng.**, SE, n. 3, p. 278–286, 1980.

ZHOU, Y. et al. P4tester: Efficient runtime rule fault detection for programmable data planes. In: **Proceedings of the International Symposium on Quality of Service**. New York, NY, USA: Association for Computing Machinery, 2019. (IWQoS '19). ISBN 9781450367783. Available from Internet: <https://doi.org/10.1145/3326285.3329040>. Accessed in: 30 Jan. 2020.

APPENDIX A — RESUMO EXPANDIDO

Com o avanço das pesquisas em redes definidas por software, o conceito de plano de dados programáveis teve um interesse aprofundado pelos pesquisadores (FEAMSTER; REXFORD; ZEGURA, 2014; CORDEIRO; MARQUES; GASPARY, 2017). Linguagens específicas de domínio como POF e P4 agora permitem que operadores de rede possam redefinir como os dispositivos de encaminhamento de dados analisam e processam pacotes.

Com isso, os operadores de rede agora podem definir seus próprios códigos para implementar alguma especificação de protocolo fazendo com que seja necessária a realização de etapas de verificação e validação (V&V) para cada código diferente implementado pelos operadores. Existe um empenho da comunidade de pesquisadores de rede em investigar soluções para solucionar defeitos de programas para o plano de dados antes que causem algum dano. Existem trabalhos que já exploraram diferentes técnicas de teste como verificação estática (LOPES et al., 2016), asserções (FREIRE et al., 2018; NEVES et al., 2018; LIU et al., 2018), execução simbólica (STOENESCU et al., 2018; FREIRE et al., 2018; NEVES et al., 2018; LIU et al., 2018), testes funcionais (NöTZLI et al., 2018; ZHOU et al., 2019) e aprendizagem de máquina (SHUKLA et al., 2019). Estas técnicas de teste conseguem identificar algumas classes de defeitos com a intervenção adicional do programador (que também pode inserir novos defeitos) para verificação, mas deixam descobertas outras classes de defeitos que precisam ser explorados utilizando outras técnicas de teste complementares para consolidar uma estratégia V&V mais ampla para programas de plano de dados.

A.1 Contribuições da Dissertação

Nesta dissertação revisamos o estado da arte em verificação e validação de programas para o plano de dados e confirmamos ainda mais o argumento de que as técnicas exploradas pelos trabalhos existentes não cobrem todas as classes de defeitos de software. A partir da revisão da literatura, também identificamos oportunidades no espaço da solução que não foram exploradas por investigações anteriores. Em particular, argumentamos que a análise do fluxo de dados é um elemento promissor para o design de ferramentas V&V que não dependem de um esforço extra de programação para encontrar defeitos de switch.

Desta forma, neste trabalho exploramos a técnica de teste de fluxo de dados com o objetivo de identificar diferentes classes de defeitos sem a necessidade de intervenção humana. Na nossa proposta mapeamose os possíveis caminhos de execução de um programa de switch P4 e analisa a ordem das operações de leitura / escrita executadas nos campos de cabeçalho e variáveis locais.

A.2 Principais Resultados Alcançados

Para avaliar a viabilidade técnica da nossa abordagem, desenvolvemos a ferramenta P4-Data-Flow e a avaliamos usando os populares códigos de switch P4 disponíveis publicamente. A partir dos nossos experimentos, confirmamos as potencialidades e limitações do uso da técnica de teste de fluxo de dados como um recurso para detectar defeitos nas implementações de switches sem exigir nenhuma entrada / esforço / conhecimento do desenvolvedor do switch, quando comparado às soluções existentes. Identificamos possíveis situações de erro, por exemplo, campos de cabeçalho lidos sem uma operação de gravação anterior.

Por fim, realizamos uma comparação qualitativa com as abordagens existentes, visando demonstrar as classes de defeitos identificadas por cada trabalho, as limitações de cada técnica e alguns direcionamentos para trabalhos futuros.