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

LUCAS MENEZES FREIRE

Uncovering Bugs in P4 Programs with Assertion-based Verification

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

Advisor: Prof. Dr. Marinho Pilla Barcellos
Coadvisor: Prof. Dr. Alberto Egon
Schaeffer-Filho

Porto Alegre
July 2018

CIP — CATALOGING-IN-PUBLICATION

Freire, Lucas Menezes

Uncovering Bugs in P4 Programs with Assertion-based Verification / Lucas Menezes Freire. – Porto Alegre: PPGC da UFRGS, 2018.

48 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor: Marinho Pilla Barcellos; Coadvisor: Alberto Egon Schaeffer-Filho.

1. P4. 2. Verification. 3. Programmable Data Planes. I. Barcellos, Marinho Pilla. II. Schaeffer-Filho, Alberto Egon. III. Título.

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

“Essentially, all models are wrong, but some are useful.”

— George E. P. Box

ACKNOWLEDGMENTS

I thank everyone that spent direct effort and time in the development of this work. Specially Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho and Marinho Barcellos. This work would not be the same without their contributions. I shall also thank all the reviewers of the papers related to ASSERT-P4 for their valuable feedback.

Finally, I thank my colleagues at UFRGS for the knowledge, insights, and encouragement originated from our conversations and time spent together.

ABSTRACT

Recent trends in software-defined networking have extended network programmability to the data plane through programming languages such as P4. Unfortunately, the chance of introducing bugs in the network also increases significantly in this new context. To prevent bugs from violating network properties, the techniques of enforcement or verification can be applied. While enforcement seeks to actively monitor the data plane to block property violations, verification aims to find bugs by assuring that the program meets its requirements. Existing data plane verification approaches that are able to model P4 programs present severe restrictions in the set of properties that can be verified.

In this work, we propose ASSERT-P4, a data plane program verification approach based on assertions and symbolic execution. Network programmers annotate P4 programs with assertions expressing general correctness properties. The annotated programs are transformed into C models and all their possible paths are symbolically executed.

Since symbolic execution is known to have scalability challenges, we also propose a set of techniques that can be applied in this domain to make verification feasible. Namely, we investigate the effect of the following techniques on verification performance: parallelization, compiler optimizations, packet and control flow constraints, bug reporting strategy, and program slicing. We implemented a prototype to study the efficacy and efficiency of the proposed approach. We show it can uncover a broad range of bugs and software flaws, and can do it in less than a minute considering various P4 applications proposed in the literature. We show how a selection of the optimization techniques on more complex programs can reduce the verification time in approximately 85 percent.

Keywords: P4. Verification. Programmable Data Planes.

Revelando Bugs em Programas P4 com Verificação Baseada em Asserções

RESUMO

Tendências recentes em redes definidas por software têm estendido a programabilidade de rede para o plano de dados através de linguagens de programação como P4. Infelizmente, a chance de introduzir bugs na rede também aumenta significativamente nesse novo contexto. Para prevenir bugs de violarem propriedades de rede, as técnicas de imposição e verificação podem ser aplicadas. Enquanto imposição procura monitorar ativamente o plano de dados para bloquear violações de propriedades, verificação visa encontrar bugs assegurando que o programa satisfaz seus requisitos. Abordagens de verificação de plano de dados existentes que são capazes de modelar programas P4 apresentam restrições severas no conjunto de propriedades que podem ser verificadas. Neste trabalho, nós propomos ASSERT-P4, uma abordagem de verificação de programas de plano de dados baseada em asserções e execução simbólica. Programadores de rede anotam programas P4 com asserções expressando propriedades gerais de corretude. Os programas anotados são transformados em modelos C e todos os seus caminhos possíveis são executados simbolicamente. Como execução simbólica é conhecida por possuir desafios de escalabilidade, nós também propomos um conjunto de técnicas que podem ser aplicadas neste domínio para tornar a verificação factível. Nomeadamente, nós investigamos o efeito das seguintes técnicas sobre o desempenho da verificação: paralelização, otimizações de compilador, limitações de pacotes e fluxo de controle, estratégia de reporte de bugs, e fatiamento de programas. Nós implementamos um protótipo para estudar a eficácia e eficiência da abordagem proposta. Nós mostramos que ela pode revelar uma ampla gama de bugs e defeitos de software, e é capaz de fazer isso em menos de um minuto considerando diversas aplicações P4 encontradas na literatura. Nós mostramos como uma seleção de técnicas de otimização em programas mais complexos pode reduzir o tempo de verificação em aproximadamente 85 por cento.

Palavras-chave: P4, Verificação, Plano de Dados Programáveis.

LIST OF ABBREVIATIONS AND ACRONYMS

DAG	Directed Acyclic Graph
HSA	Header Space Analysis
IR	Intermediate Representation
NOD	Network Optimized Datalog
SAT	Boolean Satisfiability Problem
SDN	Software-defined Networking
SEFL	Symbolic Execution Friendly Language
SMT	Satisfiability Modulo Theories
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VMN	Verification for Middlebox Networks

LIST OF FIGURES

Figure 2.1 Code circumvention example. 14
Figure 2.2 Control misconfiguration example. 15

Figure 4.1 Overview of ASSERT-P4. 19
Figure 4.2 Assertion language grammar. 21
Figure 4.3 Example of an annotated P4 program.. 22
Figure 4.4 Header translation example. 23
Figure 4.5 Table translation example. 24
Figure 4.6 Action translation example. 24
Figure 4.7 Control block translation example. 25
Figure 4.8 Parser translation example. 25

Figure 5.1 Example of submodel generation. 29
Figure 5.2 Constraint example. 31

Figure 6.1 ASSERT-P4 performance analysis. 37
Figure 6.2 Optimization techniques applied to the synthetic benchmarks. 39

LIST OF TABLES

Table 6.1 Expressiveness of the proposed assertion language 36
Table 6.2 Reduction in the verification time of each technique. 40
Table 6.3 Reduction in the number of instructions of each technique. 40
Table 6.4 Verification time and number of instructions reductions obtained from combining the techniques. 42

CONTENTS

1 INTRODUCTION..11
2 BACKGROUND AND MOTIVATION...13
2.1 P4 language...13
2.2 P4 program-specific bugs 14
2.3 Data plane verification..15
2.4 Symbolic execution..16
3 RELATED WORK 17
4 ASSERT-P4..19
4.1 Overview 19
4.2 Specifying assertions...20
4.3 Constructing C models 23
4.4 Symbolically executing program models 26
5 VERIFYING P4 PROGRAMS IN FEASIBLE TIME 28
5.1 Parallelization..28
5.2 Compiler optimizations 30
5.3 Packet and control flow constraints 30
5.4 Bug reporting strategy...31
5.5 Program slicing 32
6 EVALUATION...33
6.1 Prototype implementation..33
6.2 Bug finding..33
6.3 Language expressiveness 35
6.4 Performance analysis...37
6.5 Benchmarking optimization strategies 38
6.6 Analysis of optimization strategies on existing P4 programs...................................39
7 CONCLUSION 44
REFERENCES..45

1 INTRODUCTION

Data plane programmability allows operators to quickly deploy new protocols and develop network services. Through programming languages such as P4 (BOSSHART et al., 2014), it is possible to specify in a few instructions which and how packet headers should be manipulated by different forwarding devices in the infrastructure. Despite the flexibility, this paradigm also increases the chance of introducing bugs into the network due to incorrect protocol implementations.

Enforcement and verification are complementary approaches that can be applied as a solution to this problem. Using enforcement, the data plane can be monitored during execution to seek and block actions that result in property violations. Verification focuses on finding the bugs before the programs are deployed. It acts by assuring that the program meets the properties stated by its requirements. Several approaches have been developed in order to check if a given data plane satisfies a set of intended properties (SON et al., 2013; DOBRESCU; ARGYRAKI, 2014; LOPES et al., 2015; PANDA et al., 2017). However, the ones that are able to model P4 programs cannot reason about program-specific properties. In this dissertation, we propose ASSERT-P4, a network verification approach capable of modeling and checking (at compile time) general security and correctness properties of P4 programs. It provides an expressive assertion language that enables programmers to specify their intended properties by simply annotating their P4 programs. Once annotated, a program is symbolically executed, with assertions being checked while all its paths are traversed. Given that the time taken to perform symbolic execution grows exponentially with the program complexity, we also present a variety of optimization techniques that can be employed to reduce the verification time and the number of executed instructions. These techniques consist of using parallelization of symbolic execution, compiler optimization flags, code annotations to constrain packets and the control flow, a bug reporting strategy to optimize I/O operations and assertion violation discovery, and program slicing to reduce the complexity of the model under verification.

We built a prototype of ASSERT-P4 using KLEE (CADAR; DUNBAR; ENGLER, 2008) and the P4 Reference Compiler (CONSORTIUM, 2017b). To evaluate our approach, we tested it on four real P4 applications collected from the literature: Switch (CONSORTIUM, 2018), NetPaxos (DANG et al., 2016), Dapper (GHASEMI; BENSON; REXFORD, 2017), and DC.p4 (SIVARAMAN et al., 2015). We found bugs in the first three.

Our results show that ASSERT-P4 can uncover a broad range of bugs and software flaws, either in a data plane program itself or in its control plane configuration. A detailed performance analysis also shows that, although the verification time grows exponentially with the number of tables and assertions, pragmatically ASSERT-P4 needs less than a minute to verify various P4 applications, such as NetPaxos (DANG et al., 2015), LossRadar (LI et al., 2016), NDN.p4 (SIGNORELLO et al., 2016), Gotthard (JEPSEN et al., 2017), Dapper (GHASEMI; BENSON; REXFORD, 2017), and Timestamp Switching (EDWARDS; CIARLEGLIO, 2017). Furthermore, the proposed optimization techniques were used to reduce the verification time in up to 85% of the original, unoptimized executions.

In summary, this dissertation presents the following contributions:

1. a language for specifying general correctness and security properties of P4 programs;
2. an assertion checking and symbolic execution approach for verifying properties of P4 programs;
3. a set of techniques to enable feasible verification of P4 programs;
4. the usage of ASSERT-P4 for uncovering software flaws in P4 applications proposed in the literature;
5. a detailed performance evaluation of ASSERT-P4.

The remainder of this work is organized as follows. We first provide (in Chapter 2) a background of P4, data plane verification, and symbolic execution, as well as explain the motivation for verification of program-specific properties of P4 programs. Next, we present an overview of the state of the art (Chapter 3). Thereafter, the proposed verification solution is described in detail (Chapter 4), followed by an explanation of the optimization techniques that can be applied with ASSERT-P4 (Chapter 5). We then describe the experimental evaluation (Chapter 6) and outline the main conclusions and future work (Chapter 7).

2 BACKGROUND AND MOTIVATION

We start with a description of the main aspects of the P4 language and its compiler (Section 2.1). Next, we motivate our work with examples of bugs ASSERT-P4 aims to identify (Section 2.2), followed by the concepts and goals of data plane verification (Section 2.3), and an explanation of symbolic execution (Section 2.4).

2.1 P4 language

The P4 programming language, proposed by (BOSSHART et al., 2014), allows the programming of the data plane of network devices (that is, switches and routers) in a simple and architecture-independent manner. This allows new communication protocols to be quickly deployed in the network.

A P4 program basically includes the definitions of headers, metadata, parser, actions, tables, control blocks and external objects. *Parser* describes the mapping of input packet bits to their corresponding *headers* declared in the program. Once mapped, such headers can be manipulated by *tables* and *actions*. The exact sequence of tables and actions applied during the packet processing is defined in an imperative manner by *control blocks*. *Metadata* allows devices to temporarily store packet state information. Finally, *external objects* act as interfaces to device-specific data structures and functions. For instance, counters made available by programmable switches are manipulated by P4 programs through external objects. It is the device manufacturers responsibility to implement these interfaces.

The code of a P4 program is usually organized in libraries. A core library containing basic data types is provided along with its programming framework. Two examples of basic data types are *packet_in* and *packet_out*, used to represent respectively the incoming and outgoing packet (that is, the stream of bits). Other libraries can be freely defined by device manufacturers and network programmers.

A P4 program to be run is first translated by a compiler to instructions specific to the target device, such as CPU, GPU, FPGA, network processors, or programmable ASICs. In this process, the P4 compiler converts the program source code into an intermediate representation (IR), modeled by a directed acyclic graph (DAG). Each node of the graph, represented by an element of the P4 program, is then transformed in their corresponding low-level instructions according with the target device. Many code opti-

mizations are usually applied during this process, allowing network devices to process packets at higher rates.

2.2 P4 program-specific bugs

Bugs in P4 programs can originate from a myriad of sources (e.g. erroneous assignments, poor logic or control misconfiguration) and have different consequences depending on the program. Some bugs, for example, can be transformed into vulnerabilities if exploited towards the violation of network security policies. Next, we present two motivating examples to illustrate how bugs and their effects can be specific to each P4 implementation.

Code circumvention. Figure 2.1 shows an example of a vulnerability stemming from a logic error in a P4 program. This code snippet specifies a packet processing pipeline containing three match-action tables (*udp_table*, *tcp_table* and *tcp_acl_table*), invoked inside an *L4* control block. While it is expected that *udp_acl_table* should be applied to UDP traffic, the *tcp_acl_table* was used in its place, resulting in UDP packets that can bypass its filtering mechanism. As a consequence, the program in question could be used as a starting point for many attacks (e.g., UDP flooding). Even though correcting this problem is simple (applying the proper table that implements the UDP access control list is enough), finding it may not be trivial in large and complex programs.

Figure 2.1: Code circumvention example.

```
 1 control L4() {
 2   apply {
 3     if (headers.ip.nextHeader == TCP) {
 4       tcp_table.apply();
 5       tcp_acl_table.apply();
 6     } else if (headers.ip.nextHeader == UDP) {
 7       udp_table.apply();
 8       tcp_acl_table.apply();
 9     }
10   }
11 }
```

Source: The Authors

Control misconfiguration. Many faults in networks arise from bugs in forwarding rules (i.e., control plane configurations). In this sense, Figure 2.2 shows an example of a data plane program whose tables are erroneously configured at compile-time. The *mirror* table clones packets based on their output port (line 2), setting a new port for cloned

packets based on its action parameters. In this example, one of the forwarding rules is assigning the output port of the cloned packet to the same value as the original packet (line 8). As a consequence, both packets will be sent to the receiver.

Figure 2.2: Control misconfiguration example.

```
 1 table mirror {
 2   key = { metadata.egress_port : exact; }
 3   actions = { NoAction; clone_packet; }
 4   default_action = NoAction;
 5
 6   const entries = {
 7     0x00000001 : clone_packet(0x00000002);
 8     0x00000002 : clone_packet(0x00000002);
 9   }
10 }
```

Source: The Authors

To prevent these cases from causing negative consequences to the network, we can use an active approach of enforcing correctness properties of interest, and impeding the behavior of P4 programs that violate these properties. Alternatively, we can verify if the P4 program does not violate the properties before their deployment, giving the opportunity to find the bugs in an earlier stage of development.

2.3 Data plane verification

The investigation of solutions for data plane verification was motivated by the complexity of network data planes and the difficulty in diagnosing problems (MAI et al., 2011). These solutions provide tools that generally translate a data plane configuration to a model, upon which different techniques are used to prove that it satisfies a set of properties related to a given network policy. Examples of this approach include (MAI et al., 2011), (SON et al., 2013), (LOPES et al., 2015), and (LOPES et al., 2016).

Different types of properties can be proved depending on the verification tool. However, the majority of these properties are related to reachability, that is, checking if hosts are able to communicate. Other common properties include proving isolation (hosts should *not* be able to communicate), and ensuring that the data plane has no routing loops and black holes.

The control plane centralization provided by software-defined networking (SDN) allows the automatic verification of forwarding behavior (KAZEMIAN; VARGHESE; MCKEOWN, 2012). However, extending the programmability of SDN to the data plane

brings new challenges to the verification of networks. The most important ones are (i) providing an automatic approach to avoid manually generating models for each new data plane program source code (LOPES et al., 2016), and (ii) defining the set of verifiable properties. The first challenge is necessary to be addressed because the malleability of P4 makes manual modeling impractical. The second challenge arises from the possibility of implementing arbitrary protocols in P4, which may need to be verified with program-specific properties. In this dissertation, we address these points with the symbolic execution of automatically generated models of P4 programs, which are annotated with properties written in an expressive language.

2.4 Symbolic execution

Symbolic execution and model checking are two main techniques that can be applied to network verification. Symbolic execution approaches execute a program with symbolic (instead of concrete) inputs to traverse all its feasible paths (DOBRESCU; ARGYRAKI, 2014; STOENESCU et al., 2016; FAYAZ et al., 2016; CANINI et al., 2012a). Upon encountering a branch decision dependent on symbolic values, the symbolic execution engine invokes a satisfiability modulo theories (SMT) solver to determine which paths are feasible. The execution then traverses each possible path, including the assumption on the symbolic values necessary to choose their corresponding branch. Note that using a solver is a costly operation. Model checking is the process of modeling the system and its properties, and checking if the properties are valid in all states of the model.

Thus, the advantage of symbolic execution over model-checking is performance: symbolic engines use SMT solvers to make path reachability decisions only, while model-checking based approaches encode both network models and their properties into the solvers (e.g., as in (SON et al., 2013; LOPES et al., 2016)). However, symbolic execution faces the path explosion problem, arising from the exponential nature of traversing all paths of an execution tree. This may result in prohibitive verification times on complex programs. Despite this challenge, model-checking has no performance advantage over symbolic execution as it has to deal with the similar state explosion problem, which consists of an exponential growth of the number of states as the system grows in complexity.

3 RELATED WORK

Network verification. Many tools were proposed for verifying correctness and security properties in computer networks over the last few years. They are based on a myriad of techniques and address different properties and/or network architectures. While the types of properties vary across the literature, they are mainly related to host reachability, including isolation, absence of black holes, and loop-freedom. Some focus on the control plane, while others, in the data plane. Approaches that target the control plane proactively analyze their network operations, being useful to verify that the control plane does not yield configurations that violate the desired properties (LIU et al., 2017; BECKETT et al., 2017; GEMBER-JACOBSON et al., 2016; FOGEL et al., 2015; CANINI et al., 2012b).

Efforts that focus on the data plane are more similar to our approach. They operate by verifying if a particular snapshot of the data plane satisfies the network-wide properties. This strategy can be traced back to Anteater (MAI et al., 2011), which models the data plane as boolean functions that are analyzed with a SAT solver to check for reachability, network loops, black holes, and consistency. Similarly, Header Space Analysis (HSA) (KAZEMIAN; VARGHESE; MCKEOWN, 2012) proposes header space algebra as a technique for checking reachability, isolation of network slices and packet leakage. Based on HSA, NetPlumber (KAZEMIAN et al., 2013) incrementally updates the network model as changes occur in the data plane. This allows efficient verification in real time. Other tools that perform real time verification of the data plane are VeriFlow (KHURSHID et al., 2012), DeltaNet (HORN; KHERADMAND; PRASAD, 2017), and Flover (SON et al., 2013). VMN (PANDA et al., 2017) focuses on verifying reachability and isolation in networks containing stateful middleboxes. NOD (LOPES et al., 2015) uses Datalog to model both the network and its reachability properties. Recently, a solution that translates P4 programs to Datalog was proposed in the literature (LOPES et al., 2015), but unlike ASSERT-P4 it cannot reason about program-specific properties. In (CASCAVAL et al., 2018), the authors are in the process of publishing the use of Hoare logic to prove general and program-specific properties of P4 programs.

The symbolic execution technique has been previously used to verify data planes. (DOBRESCU; ARGYRAKI, 2014) proves that pipelines composed of Click elements satisfy *crash-freedom*, *bounded execution*, and packet filtering properties. The authors try to handle the path explosion problem by symbolically executing the Click elements

separately. Symnet (STOENESCU et al., 2016), in turn, is a verifier of data plane models built using the SEFL language, also proposed by the authors. This language contains instructions that simplify its symbolic execution, allowing the efficient verification of complex programs. Despite its scalability, Symnet cannot verify P4 programs in its current state, since the SEFL language is not able to model all the required structures and data manipulations (e.g., bitwise operations). However, the authors are in the process of publishing Vera (STOENESCU et al., 2018), a P4 verification approach capable of using SEFL and Symnet to prove properties using symbolic execution. Compared to the published network verification literature, ASSERT-P4 is the first work to allow expressing and checking program-specific properties of P4 programs.

Assertion language. Beckett et al. (BECKETT et al., 2014) present an assertion language to verify SDN applications. It enables expressing properties that the data plane should satisfy at different points of a control program. The assertions are verified using the VeriFlow (KHURSHID et al., 2012) tool, which, like Flover, acts over forwarding rules instantiated in OpenFlow devices. While the language Beckett et al. propose is used in SDN applications, our approach is to directly annotate a data plane program to prove properties of interest.

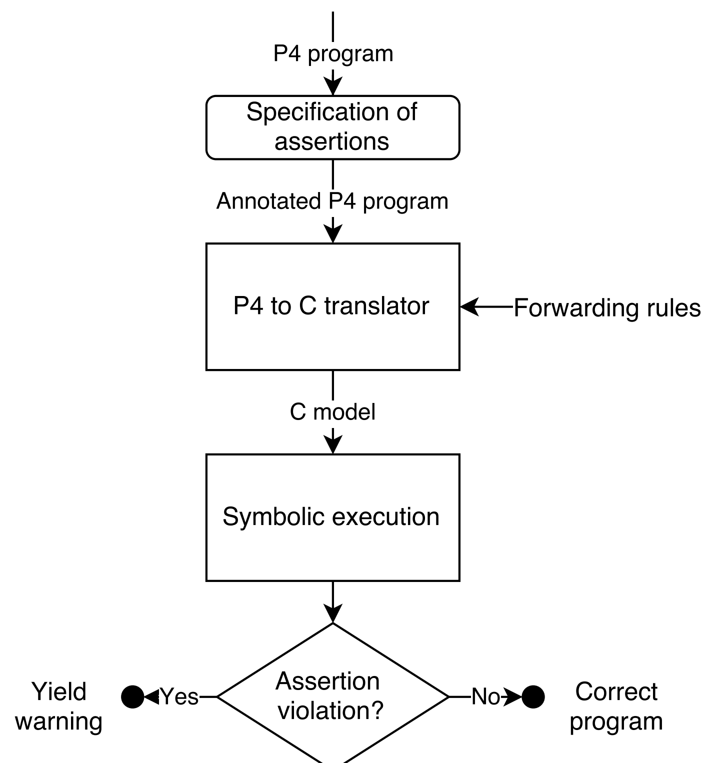
4 ASSERT-P4

This chapter explains the ASSERT-P4 verification process. We describe the control flow of the verification mechanism (Section 4.1), the design of our assertion language (Section 4.2), our model construction process (Section 4.3), and the symbolic execution of the generated models (Section 4.4).

4.1 Overview

Figure 4.1 shows the overview of ASSERT-P4. There are two key ideas behind it: i) using assertions for specifying properties about P4 programs; and ii) verifying models derived from annotated programs. The former allows programmers to express their intended properties, while the latter enables programs to be automatically verified. Using models to represent real programs is a common practice in the verification literature (STOENESCU et al., 2016; PANDA et al., 2017; FAYAZ et al., 2016; LOPES et al., 2015), and although we recognize the importance of proving that our models are equivalent to their original programs, such proofs are beyond the scope of this dissertation and left for future work.

Figure 4.1: Overview of ASSERT-P4.



Source: The Authors

The P4 developer first annotates his code with assertions expressing general properties of interest. These properties can reflect a network security policy or simply represent the expected program behavior. Once annotated, the P4 program is translated into a C-based model. During this process, forwarding rules can be optionally added as input to the translator for restricting the verification to a given network configuration. The generated model is finally checked by a symbolic execution engine, which tests all execution paths looking for assertion failures. If no assertion is violated during this process, the P4 program is considered correct with respect to the analyzed properties. Otherwise, any violations are reported, allowing the developer to correct the program. In the following sections, we describe in detail each step of the ASSERT-P4 verification process.

4.2 Specifying assertions

Programmers use assertions to express properties of P4 programs. An assertion language is needed to capture packet processing behaviors and facilitate the task of specifying complex networking properties. This includes reasoning about packet formation, forwarding, and control flow properties, whose behavior may depend not only on the state of the program variables at a specific location, but also on how the program manipulates the packets at other points of the code. To accomplish this goal, we present a novel assertion language using the code annotation mechanism available in P4. We define an *assert* annotation, enabling the developer and/or a third party to express/interpret properties in an intuitive manner.

Figure 4.2 summarizes the grammar of our assertion language. Although it resembles C-style assertions found in traditional programming languages, our concept of assertion is a bit more general in the sense it can involve both location-restricted and location-unrestricted elements. A location-restricted element is one that tests the value of a program variable at a specific location (i.e., where the assertion is specified), as in traditional programming languages like C or Java. Drawing inspiration from Beckett et al. (BECKETT et al., 2014), whose assertion language allows expressing the behavior of the network evolution, we include location-unrestricted methods in our assertion language, which are designed to facilitate the specification of P4 related properties. A location-unrestricted element tries to capture the evolution of the program and how it manipulates its variables (i.e., packet headers in this case) along its execution as a whole.

Syntactically, each assertion is composed of a boolean expression b, which may

Figure 4.2: Assertion language grammar.

```
b ::= v                  m ::= forward()
    | f                      | traverse_path()
    | m                      | constant(f)
    | !b                     | if(b, b, [b])
    | b || b                 | extract_header(h)
    | b && b                 | emit_header(h)
    | b == b             i ::= v
    | b != b                 | f
    | i >= i                 | i * i
    | i <= i                 | i / i
    | i < i                  | i % i
    | i > i                  | i + i
    | i == i                 | i - i
    | i != i
```

Source: The Authors

include primitive methods m. The set of allowed methods is {*forward, traverse_path, constant, if, extract_header, emit_header*}. Both expressions and methods can operate over one or more values v, header fields f or headers h. There is no syntax difference between location-restricted and location-unrestricted elements. Semantically, each assertion represents a boolean that should evaluate to true or false, where values and header fields evaluate to true if they are non-zero and false otherwise. Integer expressions i have the same semantics as their equivalents in the P4 language, as well as boolean expressions, which include the equality and inequality relational operators to compare logical values.

Each method m has its own meaning. Specifically, *forward()* returns true when the packet being processed will not be dropped at the end of the program. *traverse_path()* indicates if a given structure in the program (e.g., an action) was eventually traversed before the end of the program execution. *constant(f)* is true if the field f is not changed from the assertion location to the end of the program execution. *if*(b_1, b_2, [b_3]) is similar to traditional conditional statements (i.e., if the condition represented by expression b_1 is true, then the expression b_2 will be evaluated, otherwise the optional expression b_3 will be verified). *extract_header(h)* is true if the header *h* is extracted from the packet during the *parsing* process. Finally, *emit_header(h)* returns true if the outgoing packet will contain the header *h* at the end of the program execution.

The methods presented in the language enable the specification of types of properties that would be either difficult or impossible to express using only traditional assertions. The addition of the *forward* method enables the expression of forwarding properties, which are essential to data plane programs. The *traverse_path* method allows reason-

ing about the control flow of the source code. *constant* facilitates checking the integrity of variables across the program. Both *extract_header* and *emit_header* allows the expression of packet formation properties at the parser and deparser level, respectively. Finally, the *if* method assists the process of combining methods and expressions in a conditional expression.

Figure 4.3 shows an example containing an annotated P4 program where assertions are in bold, with only the most relevant parts of the program being displayed. This program describes a packet processing pipeline with a single table (*dmac*), which is instantiated inside the *TopPipe* control block. Each entry of this table can invoke one of two actions (*Drop* or *Set_dmac*). The annotated assertions aim to verify that: (i) packets marked to drop are never forwarded (line 7), and (ii) only packets with TTL greater than zero are forwarded (line 21). The two assertions contain both location-unrestricted elements (e.g., the method $forward$ captures the state of the program at the end of its execution) and also location-restricted ones (e.g., the expression "$headers.ip.ttl > 0$" tests the value of $headers.ip.ttl$ at the point in which the assertion is located).

Figure 4.3: Example of an annotated P4 program.

```
1  ...
2 control TopPipe(inout Parsed_packet headers,
3                       out OutControl outCtrl) {
4  ...
5  action Drop() {
6    outCtrl.outputPort = DROP_PORT;
7    @assert("if(traverse_path(), !forward())");
8 }
9  action Set_dmac(EthernetAddress dmac) {
10    headers.ethernet.dstAddr = dmac;
11  }
12  table dmac {
13    key = { nextHop : exact; }
14    actions = { Drop; Set_dmac; }
15    default_action = Drop;
16  }
17  apply {
18    ...
19    dmac.apply();
20    ...
21   @assert("if(forward(), headers.ip.ttl > 0)");
22  }
23}
```

Source: The Authors

4.3 Constructing C models

Once a P4 program is annotated, the tool part of the ASSERT-P4 implementation generates an equivalent program in the C language through a translation process. This section describes how we designed this process, discussing the main P4 structures (i.e., headers, tables, actions, parsers, control blocks, and external objects).

Headers. Given their similar representations, P4 headers are properly modeled by *structs* in C. Each header field is mapped to a *struct* member, and *bit fields* in C are used to keep the matching between the size of the header field and the size of its corresponding member in the generated *struct*. Each basic type in P4 is mapped to a corresponding type in C, considering its declared size. Fields with more than 64 bits can be modeled using bit arrays. Figure 4.4 exemplifies the header translation process. Note how the C struct contains a header validity field, which is implicit in Figure 4.4(a).

Figure 4.4: Header translation example.

```
1 header ethernet_t {
2    bit<48> dstAddr;
3    bit<48> srcAddr;
4    bit<16> etherType;
5 }
```

(a) P4 header.

```
1 typedef struct {
2    uint8_t isValid : 1;
3    uint64_t dstAddr : 48;
4    uint64_t srcAddr : 48;
5    uint32_t etherType : 16;
6 } ethernet_t;
```

(b) C model header

Source: The Authors

Tables. Each table in a P4 program is modeled as a function in C. Functions created from tables are constructed in different ways depending on whether the forwarding rules are supplied to the translator or not. If the rules are provided, the *match* fields in the P4 table are tested against their corresponding rule values using the specified matching approach (e.g., exact, ternary or longest-prefix match). Otherwise, the decision of which action to execute is made based on a symbolic value specially declared to force the creation of multiple execution paths by the symbolic engine (one for each action listed in the table). To avoid conflicts caused by tables from different scopes having the same name, we append an ID to their names. This solution is also applied in any situation where name conflicts may be an issue (e.g. action names). Figure 4.5 shows an example of a P4 table translated to C with no forwarding rules provided. In this case, a symbolic variable is used to make the symbolic execution traverse both actions.

Actions. Like tables, actions are also modeled as C functions. The action parameters should be translated taking into account the table modeling strategy. When the

Figure 4.5: Table translation example.

```
1  table forward_table() {
2      actions = {
3          forward;
4          NoAction;
5      }
6      key = {
7          hdr.ethernet.dstAddr: exact;
8      }
9      size = 32;
10     default_action = NoAction();
11 }
```

(a) P4 table.

```
1 void forward_table() {
2    int symbol;
3    make_symbolic(symbol);
4    switch(symbol) {
5        case 0: forward(); break;
6        default: NoAction(); break;
7    }
8 }
```

(b) C model table

Source: The Authors

forwarding rules are unknown, the action parameter values are also unknown. In this case, the actions parameters are treated as symbolic variables. If the forwarding rules are supplied, then the values specified by the rules are assigned to the corresponding parameters. An example of an action translated to C is shown in Figure 4.6. Since no forwarding rules were provided in this example, the action parameters are modeled with symbolic values.

Figure 4.6: Action translation example.

```
1 action forward(bit<9> port) {
2     standard_metadata.egress_spec = port;
3 }
```

(a) P4 action.

```
1 void forward() {
2    uint32_t port;
3    make_symbolic(port);
4    standard_metadata.egress_spec = port;
5 }
```

(b) C model action

Source: The Authors

Control Blocks. Since a control block in P4 also includes its action and table declarations, each block is translated to multiple C functions. Local scope variables in control blocks are declared as global variables in the model to allow them to be referenced by any table and action in the block. Given that the global variables are uniquely named in the model, and that they are not reused across different packets, this modeling approach does not cause side effects on the verification result. Finally, the block body usually contains invocations to tables and actions, which are modeled as their corresponding C function invocations. This process is exemplified in Figure 4.7.

Parser. Parsers are translated to multiple C functions: one for the parser declaration itself and another for each of its states. Since local parser parameters and variables

Figure 4.7: Control block translation example.

```
1 control ingress(inout headers hdr,
2                    inout metadata meta) {
3    apply {
4       forward_table.apply();
5    }
6 }
```

(a) P4 control block.

```
1 // global variables
2 headers hdr;
3 metadata meta;
4
5 void ingress() {
6     forward_table();
7 }
```

(b) C model control block

Source: The Authors

can be accessed by any state in its scope, both structures are modeled as global variables in C. Parser output parameters, which represent the packet headers, are modeled as symbolic variables, as they correspond to inputs in the model. Figure 4.8 contains an example of a P4 parser and the C model generated by this process.

Figure 4.8: Parser translation example.

```
 1 parser TopParser(packet_in b,
 2                      out Parsed_packet hdr) {
 3
 4    state start {
 5       transition parse_ethernet;
 6    }
 7
 8    state parse_ethernet {
 9      b.extract(hdr.ethernet);
10      transition select(hdr.ethernet.etherType) {
11          0x0800: parse_ipv4;
12          default: accept;
13      }
14    }
15 }
```

(a) P4 parser.

```
 1 Parsed_packet hdr;
 2
 3 void TopParser() {
 4   make_symbolic(hdr);
 5   start();
 6 }
 7
 8 void start(){
 9    parse_ethernet();
10 }
11
12 void parse_ethernet() {
13    hdr.ethernet.isValid = 1;
14    switch(hdr.ethernet.etherType){
15        case 0x0800: parse_ipv4(); break;
16        default: accept(); break;
17    }
18 }
```

(b) C model parser

Source: The Authors

Assertions. Each assertion element is modeled in C using a particular approach. Numeric and boolean expressions, as well as the *if* method, are directly translated to their equivalent statements in C. To model Location-unrestricted methods, we use boolean values that are set at different locations depending on the method, and tested at the end of the model, after it gets its final state when executed.

To model the *extract_header*, *emit_header*, and *traverse_path* methods, a global boolean value is created for each one of their occurrences in the P4 program. Such variables assume an initial false value, and are assigned to true at different model locations depending on its corresponding method. In occurrences of an *extract_header(x)* method, the assignment is made just after an *extract* method invocation, which receives the *x* header as a parameter in the P4 program. Similarly, the assignment corresponding to the *emit_header(x)* method is made immediately after an *emit* invocation (associated to the *packet_out* basic type) containing the *x* header as a parameter. For *traverse_path*, the assignment occurs just before the assertion that declares it. *forward* methods are modeled with a single boolean value initially set to true. Its value is assigned to false inside the drop action and reject parse state. *constant(f)* is translated by storing the field *f* in a C variable right after (or before) an assertion, and testing if the variable value is the same at the end of the program.

External objects. This type of structure is specific to each forwarding device, and P4 programs only interact with their interfaces. For this reason, the behavior of each external object should be previously known. In practice, this means integrating its corresponding model into the translator by using libraries, for example. This limitation is inherent to the design of P4, which consists of both architecture-dependent and architecture-independent code. In this work, we support the external objects necessary to translate the examples presented in Chapter 6 (e.g. counters and meters of the standard architecture).

4.4 Symbolically executing program models

After being generated by the process described in the previous section, the C model of a P4 program is verified by a symbolic engine. The symbolic execution of a program requires that all its possible control flows (i.e., its execution paths) are evaluated through symbolic input variables. To this end, the ASSERT-P4 implementation described in this work uses the KLEE symbolic engine (CADAR; DUNBAR; ENGLER, 2008).

Essentially, P4 programs describe how a data packet should be processed when entering a forwarding device, generating an output packet at the end or simply dropping the original packet. In this scenario, the incoming packet headers entering the device are treated as inputs to the model and thus are always assigned to symbolic values. The number of execution paths of a P4 program, in turn, is essentially given by its packet processing pipeline structure. Whenever a table can only be accessed under some condition (e.g., depending on the used protocol), a new execution path is created. The same happens whenever multiple actions can be invoked by the same table, generating a new branch for each possibility. This behavior gives rise to the path explosion problem as the number of paths increases exponentially with the program complexity. In the next chapter, we present different approaches to address this performance challenge of our proposal.

5 VERIFYING P4 PROGRAMS IN FEASIBLE TIME

By using the symbolic execution technique, ASSERT-P4 has to face the scalability problem originated from path explosion. While many P4 programs currently found in the literature are fairly small (DANG et al., 2015; LI et al., 2016; SIGNORELLO et al., 2016; JEPSEN et al., 2017; GHASEMI; BENSON; REXFORD, 2017; EDWARDS; CIARLEGLIO, 2017), the complexity of the algorithms implemented by P4 source code is expected to grow assuming that real world P4 programs will implement multiple protocols. This effect is already possible to observe on the DC.p4 and Switch programs, which aggregate a multitude of protocols a typical data center may need.

To address this problem, we investigate five optimization techniques with the goal of decreasing the verification time of the proposed mechanism: parallelization, compiler optimizations, packet and control flow constraints, bug reporting strategy, and program slicing. Since path explosion is the performance bottleneck of ASSERT-P4, the techniques described in this chapter are mainly aimed at reducing the amount of paths the verifier must traverse in a single execution.

5.1 Parallelization

During the symbolic execution of a program, decision points are reached and execution unfolds according to symbolic values. A decision point may have multiple possible paths to continue execution (as in *if* and *switch* statements). If so, the symbolic execution forks, creating a new branch for each feasible path. Even though these branches are concurrent, KLEE follows the paths sequentially, not taking advantage of potential gains enabled by parallelization. Cloud9 (BUCUR et al., 2011) allows KLEE to use multiple processing elements while symbolically executing any C code[1]. We follow a different approach, and propose a simple parallelization strategy that is specific to P4 programs.

The strategy consists of dividing the model into *submodels*, which are statically generated from decision points (e.g. `if` and `switch`). One submodel is created for each branch, and run via a concurrent KLEE process. Figure 5.1 exemplifies this process with the corresponding code fragments, with the original model and its two submodels shown respectively in Figures 5.1(a), 5.1(b) and 5.1(c). In each submodel a value is assumed for the condition and only its corresponding instructions are executed. The submodels gen-

[1] the tool was discontinued in 2013.

Figure 5.1: Example of submodel generation.

```
1 if(hdr.ethernet.etherType == IPV4){
2    parse_ipv4();
3 } else {
4    accept();
5 }
```

(a) Original model.

```
1 klee_assume(hdr.ethernet.etherType == IPV4);
2 parse_ipv4();
```

(b) Submodel with valid condition.

```
1 klee_assume(hdr.ethernet.etherType != IPV4);
2 accept();
```

(c) Submodel with invalid condition.

Source: The Authors

erated are completely independent and thus can be executed concurrently, in any order. Notice that, even though we use various global variables when creating the C models (see Section 4.3), their access will not cause race conditions since each submodel is executed in a different KLEE process. If multiple processing elements are available in the underlying hardware, the procedure can be repeated on submodels to increase the degree of concurrency.

To maximize the performance gains of parallelization, we should minimize the maximum height of the execution trees of all submodels. This is so because the longest path is the one that will take the most time to traverse. Since the symbolic execution tree is unknown beforehand, it cannot be used to find the optimal solution. Therefore, we propose a heuristic based on the anatomy of P4 programs to partition the model.

Decision points that occur earlier in the program have more chances of being part of feasible paths, as they require less conditions to be traversed. In the case of a P4 program, initial decision points are typically found at the parser. Our heuristic then starts by creating the submodels from these first conditions seen by the parser. Once the first set of submodels is generated by this strategy, further divisions may not be as efficient because they have increasing chances of generating submodels on branches of unreachable paths. Alternatively, decision points associated with tables are appropriate candidates to submodel creation, since packets that traverse the longest paths usually pass through P4 tables after being accepted by the parser. In this case, each action in a table is traversed using a different submodel. The heuristic creates submodels, before any processing starts,

by applying both approaches (parser and table branches). Once all the submodels have been generated, they are dynamically assigned (in arbitrary order) to any idle processing elements.

5.2 Compiler optimizations

The way ASSERT-P4 performs symbolic execution of P4 programs is through the generation of a C model, upon which KLEE is applied. KLEE, furthermore, works on a LLVM assembly language representation of the C code, which is generated using the LLVM compiler (LATTNER; ADVE, 2004). One of the compiler features is support for optimization passes, such as control flow graph simplification (which can remove dead code and merge blocks of instructions) and global variable optimization (which marks unchanged variables as constant and removes unused variables). These passes can alter the source code in order to make it more efficient, potentially reducing the symbolic execution time. Given the theoretical advantages of LLVM compiler optimizations, we explore in this work how these optimizations affect the verification time of the models generated by the ASSERT-P4 translation process.

5.3 Packet and control flow constraints

During the verification process, P4 programmers may be interested in the verification of properties of only certain types of packets or control flows. For instance, a P4 program can implement both the TCP and UDP protocols, but the properties to be verified may be related to TCP only. To avoid wasting time symbolically executing the UDP paths of the code, we propose to include packet and control flow constraints to the verification process that direct the symbolic execution to the paths of interest and ignore the others.

On the one hand, the use of constraints can potentially reduce the verification time more than the other techniques presented in Chapter 5. On the other hand, it is a manual procedure: the P4 programmer has to specify which packet and control flow constraints are applicable. It is so because constraints depend on the P4 program and the properties to be verified, which cannot be obtained from the P4 source code only.

Constraints are implemented in ASSERT-P4 by a method through which P4 programs are annotated with assumptions on the code. We define the *assume* annotation,

which receives as argument a boolean value taken to be true. This annotation can be directly translated to a *klee_assume* method in the C model. The method belongs to the KLEE API and is responsible for implementing the assumption within the KLEE symbolic engine.

To illustrate this approach, consider a P4 program that implements multiple L3 protocols, but the properties of interest concern only the IPv4 protocol. This example is shown in Figure 5.2. The annotated P4 code (Figure 5.2(a)) consists of the parser state responsible for reading the Ethernet bits of the packet, which indicate which L3 protocol the packet contains. By assuming that the type of the next protocol is IPv4, the symbolic execution will ignore the other paths, and either traverse the parse_ipv4 parser state or halt if the assumption is impossible to hold. Figure 5.2(b) shows how the assumption and transition statements are translated to the C model.

Figure 5.2: Constraint example.

```
1 state parse_ethernet {
2     b.extract(hdr.ethernet);
3     @assume(hdr.ethernet.etherType == IPV4);
4     transition select(hdr.ethernet.etherType) {
5         IPV4 : parse_ipv4;
6         IPV6 : parse_ipv6;
7         ICMP : parse_icmp;
8         ...
9     }
10 }
```

(a) P4 code annotated with a constraint.

```
1 klee_assume(hdr.ethernet.etherType == IPV4);
2 parse_ipv4();
```

(b) Constraint translated to the C model.

Source: The Authors

5.4 Bug reporting strategy

Whenever ASSERT-P4 encounters a path with an assertion violation, it immediately prints the error to the user and continues traversing the path. This way, multiple violations can be revealed in the same run. Ideally, each should be reported just once, despite the fact that the same violation may appear in multiple paths. Besides being inefficient, a high amount of redundant reports may lead to I/O contention. Alternatively, the same approach taken by KLEE can be used, that is, aborting the current path immediately.

However, this is inefficient when multiple assertions are being verified, as it would require new runs to discover additional violations.

To investigate the impact of the bug reporting strategy on the verification time, we extend KLEE with a *print_once* method. It prints the assertion error message only if the message was not previously printed during the symbolic execution. The implementation ensures the error is reported as soon as the assertion violation is encountered, flushing the output.

5.5 Program slicing

To verify an assertion, ASSERT-P4 symbolically executes the whole P4 program. However, the assertion result may depend only on a subset of the program instructions. Thus, the verification procedure spends unnecessary time processing statements that do not affect the outcome. This creates the opportunity to use a program slicing technique (WEISER, 1981), which is used to remove the subset of a program that does not affect a selected criteria.

For example, consider a P4 program that process packets which may contain TCP headers, and assume that we annotate this program with an assertion that depends only on the TCP destination port. The slicing algorithm in this case can automatically generate a program slice that contains only parsers, actions, tables, and control flow instructions that can directly or indirectly modify the value of the TCP destination port. This would simplify the program, removing the parts of the code related to other protocols, and even the sections associated with TCP that do not modify the destination port.

We take advantage of the model based approach adopted by ASSERT-P4 to slice the C model instead of the P4 program. This allows the usage of existing program slicing tools to implement the optimization, and makes the development of a slicing tool for P4 programs unnecessary. To this end, we include the Frama-C (KIRCHNER et al., 2015) slicing plug-in inside the verification workflow, applying the tool before symbolically executing the C model.

6 EVALUATION

Fist, we describe the implementation of the prototype and experimental environment employed in the evaluation (Section 6.1). We use them to provide evidence that ASSERT-P4:

1. can detect a broad spectrum of bugs and policy violations in programmable data planes (Section 6.2).
2. allows the specification (and proof) of general correctness and security properties of P4 programs (Section 6.3).
3. is efficient even for relatively complex P4 programs and control configurations (Section 6.4).
4. can be optimized with the techniques presented in Chapter 5 (Sections 6.5 and 6.6).

6.1 Prototype implementation.

We have prototyped ASSERT-P4 on top of the KLEE symbolic execution engine (version 1.3.0) with LLVM version 2.9. To build C models, we first convert a P4 program to its JSON representation using the reference compiler provided by the *P4 Language Consortium*, and then translate the JSON representation (a DAG) to C code using a translator we developed specifically for this purpose. The translator implements the process described in Section 4.3 straightforwardly, containing approximately 950 lines of Python code. Shell scripts are used to automatically coordinate the invocation of each tool in the verification process. We make all the source code as well as the workloads employed in this evaluation publicly available.[1] The tool may be used by other researchers, who may want to reproduce our results. All experiments have been performed using a Linux virtual machine (kernel version 4.8.0) with four 3 GHz cores and 16 GB of RAM.

6.2 Bug finding

First, we demonstrate the effectiveness of ASSERT-P4 in finding bugs and policy violations in programmable data planes. To this end, we studied the specifications and source code of different P4 programs, which were annotated with up to 18 assertions per

[1] <https://github.com/ufrgs-networks-group/assert-p4>

program. We uncovered some of them in four recent P4 applications, which we present here. All identified bugs were manually confirmed in their respective source code. With the exception of the DC.p4 example, it was not necessary to provide forwarding rules to expose these issues.

Dapper (GHASEMI; BENSON; REXFORD, 2017): Dapper is a data plane performance diagnosis tool that infers TCP bottlenecks by analyzing packets in real time. It is divided into two phases: i) identification of new connections based on SYN flags; and ii) separation of incoming and outgoing traffic (from a server point of view) based on sequence and ack numbers. For each case, the application updates state variables used for performance diagnosis, then it forwards the packets based on the IPv4 destination address. Using our verifier, we found that Dapper forwards IPv4 packets with the time to live field equal to zero. We placed the assertion *if(ipv4.ttl == 0, !forward())* at the beginning of the ingress control block. We noted that even though the TTL field is correctly decremented, its value is never checked before forwarding. The verifier encountered violations of the property in less than a second.

NetPaxos (DANG et al., 2016): NetPaxos is a network-based implementation of the Paxos consensus protocol. There are two different types of P4 programs in this application, one for Leaders/Coordinators and another for Acceptors. All the other actors (i.e., proposers and learners) are assumed to be entirely implemented in end hosts. According to the protocol, Leaders determine a round number and ask acceptors for acknowledging it. Acceptors, in turn, decide whether they acknowledge or not a given request from a Leader. This process is repeated until a quorum of acceptors acknowledges the same round number, allowing the leader to establish a value for a given variable and consensus is achieved. By verifying the current version of the P4 code and forwarding rules made available by the authors, we found a bug on the acceptor implementation. The acceptor is supposed to vote by adding the vote information on the incoming packets and forwarding them to the learners. We verified if the packets are being forwarded after their modification using assertions of the format *if(traverse_path(), forward())* inside the actions that perform the vote. The verifier found assertion violations in less than a second, indicating that there are valid packets with voting information being dropped. The problem occurs because the packets are first marked to be dropped by another action, and not unmarked by the voting actions. This bug can be corrected by marking the packets to be forwarded inside the actions which perform the vote. According to the authors feedback on this bug, the code was ported to P4_16, leaving the old code base unmaintained and exposed to

bugs.

DC.p4 (SIVARAMAN et al., 2015): DC.p4 implements the behavior of a data center switch. It contains multiple functionalities such as L2/L3 forwarding, ECMP, VLAN, packet mirroring, tunneling and multiple ACLs (i.e., L2, L3 or based on more specific headers). This program contains more than 2500 lines of code distributed among 37 tables. These tables, in turn, are organized assuming an architecture with two sequential packet processing pipelines, one for incoming/ingress packets and another for outgoing/egress packets, interleaved by a queue system. We verify if configuring the L3 ACL table to drop traffic with a specific destination IP address properly filters this type of packet. We used the assertion *if(ipv4.dstAddr == FILTER_ADDR, !forward())* to express that packets with IPv4 destination addresses equal to *FILTER_ADDR* should be dropped. We found that just configuring the L3 ACL is not enough for dropping IPv4 packets regardless of the policy being enforced. In fact, we checked that the L3 ACL only flags packets to be filtered by another module in the system, which must also be appropriately configured. Although this is not an actual bug, it can help effectively identify misconfigurations since there is no documentation explaining how to properly configure the program.

Switch (CONSORTIUM, 2018): Since the introduction of the DC.p4 paper, its code base has evolved to the Switch.p4 program, which is actively maintained. We have used ASSERT-P4 to reproduce known, reported bugs on its repository. The first one is the modification of a field of an invalid header.[2] This bug is replicated by testing with an assertion if the header is valid before setting its fields. The second bug is related to tunnel encapsulation,[3] where encapsulated headers are overwritten whenever multiple nested levels are present. We included an assertion to test if the inner headers are not valid before performing the encapsulation. The assertion failed, confirming that encapsulated headers can be overwritten and their original contents discarded.

6.3 Language expressiveness

To evaluate our assertion language, we assessed its expressiveness in terms of the properties we can specify for different P4 programs. Table 6.1 shows a subset of the properties we tested for each P4 application. The associated assertions are italicized. It

[2]https://github.com/p4lang/switch/pull/102
[3]https://github.com/p4lang/switch/issues/97

is possible to see that we can specify a large set of properties, both program-dependent (e.g., the ones testing if registers are correctly manipulated in Dapper) and generic ones (e.g., testing whether headers have been removed from packets or not). Furthermore, both security and correctness properties can be specified, such as header integrity and well-formedness, respectively.

Table 6.1: Expressiveness of the proposed assertion language

Program	**Properties / Assertions**
VSS (CONSORTIUM, 2016)	Packets with zero TTL values are dropped *if(ipv4.ttl == 0, !forward)* Marked to drop packets are not forwarded *if(traverse_path, !forward)*
MRI (CONSORTIUM, 2017a)	Switch IDs added to packets are authentic *constant(swid)* Added IDs are not removed *if(extract_header(swid), emit_header(swid))*
Timestamp switching (EDWARDS; CIARLEGLIO, 2017)	Out of range timestamps are not forwarded to receivers *if(forward, rtp.ts < max_timestamp)*
sTag (LOPES et al., 2016)	Hosts connected to ports of different colors cannot communicate *if(ingress_port == color_a && ipv4.dstAddr == color_b_host, !forward)*
Dapper (GHASEMI; BENSON; REXFORD, 2017)	Only SYN packets register new flows *If(traverse_path*, tcp.ack == false)* *path that register new flows Load flow registers when is Ack packet *if(tcp.ack == 1, traverse_path*)* *path that load registers
NetPaxos (DANG et al., 2015)	Acceptor correctly votes according to paxos phase *if(traverse_path*, paxos.msgtype == 1A)* *at the handle_1a action Leader increases round number at each instance *if(traverse_path*, paxos.msgtype == 2A)* *at the increase_instance action
DC.p4 (SIVARAMAN et al., 2015)	L3 ACL is effective *if(ipv4.dstAddr == blocked_addr, !forward)* Cloned and original packet have different output ports ! *(cloned_outport == original_port && constant(cloned_outport))*

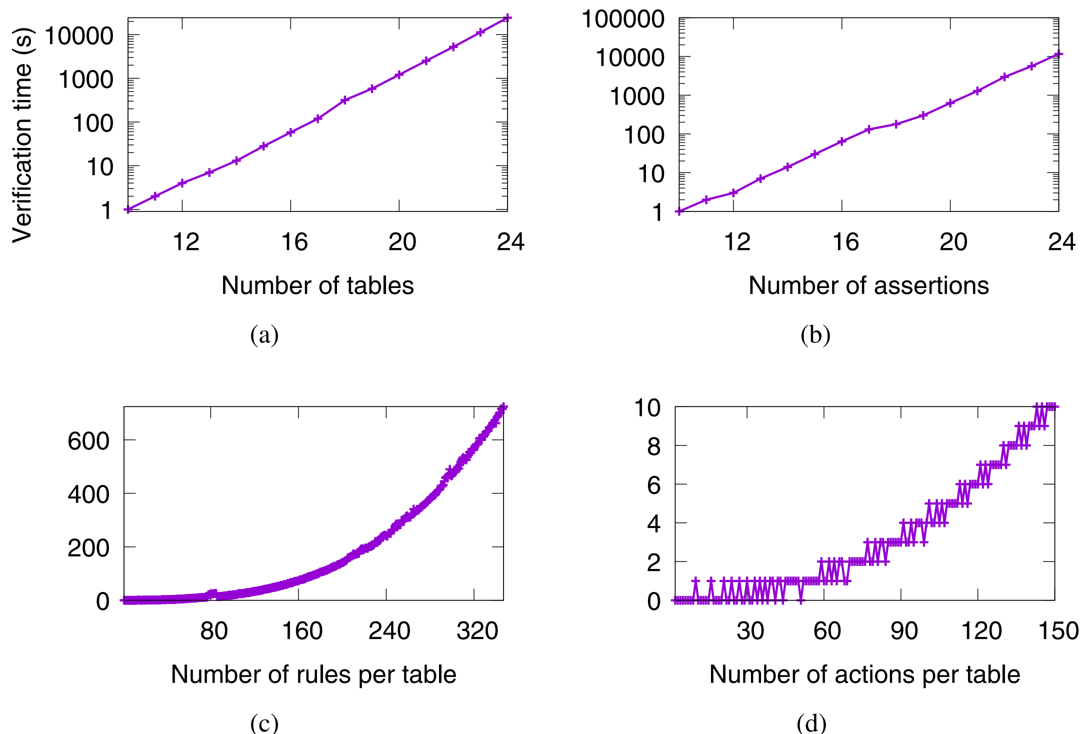
Source: The Authors

As a research direction, we envision the automatic translation of high-level networking and security policies to our assertion language, similarly to the way (BECKETT et al., 2014) translates high-level control flow properties to VeriFlow invariants. This will allow network operators to easily verify complex network topologies.

6.4 Performance analysis

We assessed how our verification approach scales according to different characteristics of P4 programs. This section shows the performance values obtained originally, and the next section presents the performance obtained with the use of the optimization techniques proposed in Chapter 5. We used the Whippersnapper (DANG et al., 2017) benchmark to generate data plane programs, and measured the impact of multiple parameters in verification times: (i) tables in the packet processing pipeline; (ii) actions associated with each table; (iii) forwarding rules used to configure a program; and (iv) assertions used to express properties. Figure 6.1 shows the results. We adopted the following default values for parameters: no forwarding rules and assertions, 1 table in Fig. 6.1(b), 2 tables in Figs. 6.1(c) and 6.1(d), and 3 actions in the first table and 2 actions in every subsequent table.

Figure 6.1: ASSERT-P4 performance analysis.



Source: The Authors

Note that instead of performing multiple executions of a small set of points of the benchmark domain, we chose to execute the experiments with more points, with a fine granularity in the domain, resulting in a natural redundancy to the presented results. By examining the graphs, we can observe the presence of trends without significant oscillations.

The results show that verification time grows exponentially with all the factors. However, it is less susceptible to the increase in the number of rules (Fig. 6.1(c)) and actions per table (Fig. 6.1(d)) compared to the number of tables (Fig. 6.1(a)) and assertions (Fig. 6.1(b)). ASSERT-P4 was able to verify within a few seconds most of the programs in Sections 6.2 and 6.3. However, the plots show clearly that our non-optimized version does not scale well, and that the verification of larger programs, with more tables and assertions, is likely unaffordable. This prompted us to investigate the adoption of optimization strategies to the context of P4 program verification.

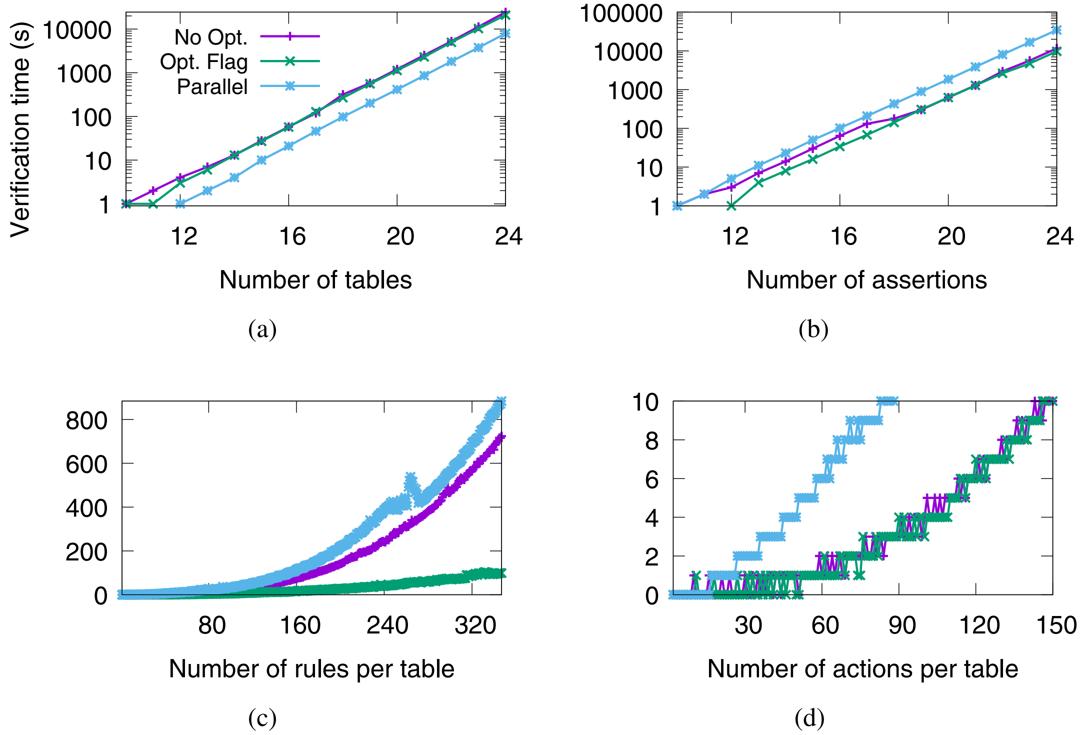
6.5 Benchmarking optimization strategies

The benchmarks shown in Section 6.4 can be re-executed in combination with some of the optimization techniques proposed in Chapter 5. The parallelization and compiler optimizations are general techniques that are suited to be analyzed with synthetic programs generated by the Whippersnapper benchmark, whereas the results of applying the other optimization techniques are tightly coupled to a particular program and the properties of interest. Therefore, we compare the original benchmark results with their execution alongside the parallelization and compiler optimization techniques in Figure 6.2.

We can observe that, using the four available cores of the machine, the parallelization technique was effective in reducing the verification time when the number of tables of the program varied (Figure 6.2(a)), while the parallelization overhead increased the verification time on the other cases. Since the number of submodels created by the parallelization strategy grew in proportion to the number of rules per table (Figure 6.2(c)) and actions per table (Figure 6.2(d)), the number of concurrent executions quickly exceeded the number of processing elements of the machine, generating a proportionally larger overhead as the value of the x-axis increases.

Parallelization was also ineffective when we varied the number of assertions (Figure 6.2(b)). This can be explained by comparing the submodel generation strategy (focused on parser and table branches) with the location of the assertions, which where all

Figure 6.2: Optimization techniques applied to the synthetic benchmarks.



Source: The Authors

inserted inside a single action. Thus, the submodels generated in this case were unbalanced, taking them a long time to be verified if they include the action containing the assertions and requiring a negligible verification time otherwise.

The compiler optimization flags resulted in moderate performance gains in all cases, except when the number of rules per table varied (Figure 6.2(c)), which presented a significant reduction in verification time from an exponential to a linear growth. This can be explained by compiler optimizations applied to the cascading if-else statements used to decide which action should be executed based on the matched values of the forwarding rules.

We can conclude that the efficiency of these optimization techniques depends on the characteristics of the program under verification. Hence, in the next section we analyze the impacts of using all the proposed techniques with various P4 programs found in the literature with the goal of obtaining insights on their application on existing programs.

6.6 Analysis of optimization strategies on existing P4 programs

We now present the results obtained from measuring the impact of each optimization technique applied to different existing P4 programs. We study their behavior by using

the techniques in isolation, as well as by combining them in a single execution. To this end, we employ two metrics: (i) the verification time it takes to explore all the paths of the model, and (ii) the total number of instructions executed by the symbolic engine. Tables 6.2 and 6.3 present the performance gains of each technique in comparison to using no optimizations, where the *Opt* and *Print* columns respectively represent the compiler optimization flags and the bug reporting strategy.

Table 6.2: Reduction in the verification time of each technique.

Program	**Opt**	**Print**	**Constraints**	**Parallel**	**Slice**
Dapper	45.87%	18.26%	49.71%	63.77%	54.60%
sTag	-	2.47%	6.17%	-39.51%	-554.43%
NetPaxos	-	5.47%	32.03%	-26.56%	-348.43%
Timestamp Switching	5.75%	0.00%	3.45%	-34.48%	-514.94%
VSS	9.41%	7.06%	2.35%	-30.59%	-420%
MRI	6.18%	3.09%	6.18%	-13.51%	-

Source: The Authors

Table 6.3: Reduction in the number of instructions of each technique.

Program	**Opt**	**Constraints**	**Parallel**	**Slice**
Dapper	58.04%	50.05%	87.79%	39.35%
sTag	-	38.89%	49.30%	22.14%
NetPaxos	-	85.33%	18.96%	74.58%
Timestamp Switching	56.46%	61.25%	40.96%	89.48%
VSS	39.94%	0.28%	10.82%	99.52%
MRI	49.75%	30.68%	20.60%	-

Source: The Authors

Parallelization. The results show that the proposed parallelization approach can greatly reduce verification time of a moderate sized program. In the Dapper example, the total verification time was reduced in approximately 64% using only 4 cores. For very small programs that can be verified in milliseconds (sTag, NetPaxos, Timestamp Switching, VSS, and MRI), the added overhead due to parallelization does not justify the use of this technique, increasing the total verification time.

When generating submodels using the approach described in Section 5.1, each submodel ends with a fraction of the total number of instructions of the original model. To achieve satisfying performance gains, the parallelization technique should try to minimize the difference between the number of instructions of the submodels. The fourth column of Table 6.3 presents the reduction on the number of instructions achieved by the submodel with the greatest number of instructions (i.e., worst case) when compared to the the original model. In this case, we can observe that this approach can reduce the number of instructions regardless of program size. However, by comparing Tables 6.2 and 6.3,

we can conclude that the cost of creating a new KLEE process for each submodel outweighs the reduction in symbolic execution time obtained from decreasing the number of instructions on very small programs.

Compiler optimizations. Unlike (DONG et al., 2015), which demonstrates that applying compiler optimizations on KLEE executions can decrease performance on some applications, our results indicate that these optimizations can have a positive effect on the time taken to symbolically execute P4 program models. This effect can be specially observed on more complex programs, which spend proportionally more time performing symbolic execution during verification. The performance gains can be explained by the reduction in the total number of instructions executed by the symbolic execution engine, which ranges from approximately 40 percent to up to almost 60 percent on the successfully tested programs.

The caveat of this technique, however, is that during the sTag and NetPaxos experiments the optimization passes of the compiler made KLEE 1.3.0 with LLVM 2.9 exit with an error during the assignment of symbolic values to the metadata structure. In the cases where no error occurred, the outcome of the tested assertions maintained the same original values as the unoptimized executions. Further investigation revealed that while these errors do not appear in different versions of KLEE and LLVM, their results cannot be presented alongside the versions currently used because they have different performance characteristics. These negative interactions with the symbolic execution engine coupled with the potential to reduce the total number of instructions even further in this domain can motivate the research of optimization passes specific for the symbolic execution of P4 programs.

Packet and control flow constraints. Although this technique has the additional cost of requiring the programmer to annotate the code with assumptions, the results presented in Table 6.3 reveal that it can greatly reduce the number of instructions symbolically executed. This reduction of instructions also leads to an equivalent reduction of the verification time of complex programs in which the symbolic execution is the verification bottleneck, as can be observed in the Dapper example. Furthermore, by using packet constraints in the bug finding examples of the Switch.p4 program (see Section 6.2), we were able to reduce the time taken to reveal the bugs from the order of days to the order of seconds. This was achieved by annotating the code with assumptions that led to the tested assertions. Note that Switch.p4 was not included in Tables 6.2 and 6.3 because its complexity makes the time necessary to exhaust all its paths unpractical for this analysis.

Bug reporting strategy. Since changing the bug reporting strategy has no effect on the number of instructions executed during symbolic execution, this technique was measured only using the verification time. The advantages of printing the failed assertions only once per execution are more noticeable when many different paths cause an assertion to fail. This can be observed on the Dapper example, where the verification time dropped by 18 percent. Conversely, this approach results in no performance gains when no assertion failed was found, as in the Timestamp Switching program. Albeit yielding smaller performance gains when compared to the other techniques, this approach can be justified given that once this technique is implemented, it can be used with no additional costs to the programmer.

Program slicing. Program slicing is capable of reducing the number of instructions considerably, from about 22 percent to more than 99 percent. However, the cost of performing the slicing with the Frama-C framework far outweighs the reduction of symbolic execution time obtained from trimming the program size on smaller programs, which can be verified without the additional overhead in milliseconds. Furthermore, the Frama-C approach to slicing has no support for programs with recursion. This resulted in a failed attempt to slice the MRI program, which contains a recursion in its parser section of the code. Therefore, we conclude that while program slicing can be an effective technique in some cases, the development of efficient slicing approaches capable of dealing with parser recursions is necessary to enable the full adoption of this technique by ASSERT-P4.

Combining the techniques. Since the use of the optimization techniques are not mutually exclusive, we analyze the potential of combining them to achieve optimal verification time in each one of the examples. For this purpose, we executed each program with all the techniques that reduced the total verification time in Table 6.2. The results are presented in Table 6.4.

Table 6.4: Verification time and number of instructions reductions obtained from combining the techniques.

Program	Used techniques	Verification Time	No. of Instructions
Dapper	opt, print, constraints, parallel	85.72%	90.89%
sTag	print, constraints	6.17%	50.70%
NetPaxos	print, constraints	35.93%	12.51%
TS Switching	opt, constraints	10.34%	14.76%
VSS	opt, print, constraints	11.76%	43.92%
MRI	opt, print, constraints	13.89%	32.54%

Source: The Authors

The only technique used in all six programs was the packet and control flow con-

straints. The compiler optimizations had to be omitted from the programs that caused errors during KLEE execution, and the print once strategy of bug reporting was not used when no assertion was violated on the Timestamp Switching case. Parallelization was only added on the Dapper example where its overhead is justified. Finally, the program slicing technique was not used in any of these cases. While slicing can be effective in isolation, the cost of executing Frama-C takes an increasingly larger part of the total verification time as the other techniques are combined, making its overhead exceed the gains obtained from a reduction in program complexity. Overall, by analyzing the results presented in Table 6.4, we conclude that the techniques we explored can decrease the verification time of P4 programs, with their effects being more noticeable on more complex programs, which spend more time performing symbolic execution.

7 CONCLUSION

We presented in this work an assertion language that can be used by P4 programmers to express correctness and security properties of a specific implementation. Our solution is more expressive than other data plane verification approaches, being the first work to allow proving properties specific to P4 source code and optionally the forwarding rules used by its tables. Our mechanism verifies the assertions using symbolic execution over C models automatically generated from the program and assertions. Alongside our tool, we presented five optimization techniques that can be used to reduce the verification time of complex programs.

We evaluated our approach by finding a broad range of bugs in real P4 programs found in the literature. The performance analysis of the proposed mechanism revealed that despite its efficiency in verifying small programs, the execution time grows exponentially with relation to the number of tables, actions, forwarding rules, and assertions. However, we also demonstrated in our experiments that combining the proposed optimization techniques we can reduce the verification time of non-trivial P4 programs in 85 percent.

As future work, we intend to explore the application of ASSERT-P4 in verifying network-wide properties of networks composed of P4 programs. The assertion language can also be investigated with the goal of providing the automatic insertion of assertions. These assertions could be used to verify general properties such as reading fields of invalid headers or checking the bounds of arrays. The P4 to C translation can be improved by proving the correctness of the process, as well as increasing the number of external objects modeled. Finally, the compiler flags and program slicing optimization techniques can be fine-tuned to ASSERT-P4 by investigating optimization passes and slicing approaches optimal to our use cases.

REFERENCES

BECKETT, R. et al. A general approach to network configuration verification. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 155–168. ISBN 978-1-4503-4653-5. Available from Internet: <http://doi.acm.org/10.1145/3098822.3098834>.

BECKETT, R. et al. An assertion language for debugging sdn applications. In: **Proceedings of the Third Workshop on Hot Topics in Software Defined Networking**. New York, NY, USA: ACM, 2014. (HotSDN '14), p. 91–96. ISBN 978-1-4503-2989-7. Available from Internet: <http://doi.acm.org/10.1145/2620728.2620743>.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2656877.2656890>.

BUCUR, S. et al. Parallel symbolic execution for automated real-world software testing. In: **Proceedings of the Sixth Conference on Computer Systems**. New York, NY, USA: ACM, 2011. (EuroSys '11), p. 183–198. ISBN 978-1-4503-0634-8. Available from Internet: <http://doi.acm.org/10.1145/1966445.1966463>.

CADAR, C.; DUNBAR, D.; ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: **Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2008. (OSDI'08), p. 209–224. Available from Internet: <http://dl.acm.org/citation.cfm?id=1855741.1855756>.

CANINI, M. et al. A NICE way to test openflow applications. In: **Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)**. San Jose, CA: USENIX, 2012. p. 127–140. ISBN 978-931971-92-8. Available from Internet: <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini>.

CANINI, M. et al. A nice way to test openflow applications. In: **Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2012. (NSDI'12), p. 10–10. Available from Internet: <http://dl.acm.org/citation.cfm?id=2228298.2228312>.

CASCAVAL, C. et al. p4v: Practical verification for programmable data planes. In: **To appear in Proceedings of the 2018 ACM SIGCOMM Conference**. New York, NY, USA: ACM, 2018. (SIGCOMM '18).

CONSORTIUM, T. P. language. **VSS Example**. [S.l.]: GitHub, 2016. <https://github.com/p4lang/p4c/blob/master/testdata/p4_16_samples/vss-example.p4>.

CONSORTIUM, T. P. language. **MRI Exercise**. [S.l.]: GitHub, 2017. <https://github.com/p4lang/tutorials/blob/master/SIGCOMM_2017/exercises/mri/solution/mri.p4>.

CONSORTIUM, T. P. language. **P4 reference compiler**. [S.l.]: GitHub, 2017. <https://github.com/p4lang/p4c>.

CONSORTIUM, T. P. language. **Switch**. [S.l.]: GitHub, 2018. <https://github.com/p4lang/switch>.

DANG, H. T. et al. Paxos made switch-y. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 46, n. 2, p. 18–24, may 2016. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2935634.2935638>.

DANG, H. T. et al. Netpaxos: Consensus at network speed. In: **Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research**. New York, NY, USA: ACM, 2015. (SOSR '15), p. 5:1–5:7. ISBN 978-1-4503-3451-8. Available from Internet: <http://doi.acm.org/10.1145/2774993.2774999>.

DANG, H. T. et al. Whippersnapper: A p4 language benchmark suite. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: ACM, 2017. (SOSR '17), p. 95–101. ISBN 978-1-4503-4947-5. Available from Internet: <http://doi.acm.org/10.1145/3050220.3050231>.

DOBRESCU, M.; ARGYRAKI, K. Software dataplane verification. In: **11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)**. Seattle, WA: USENIX Association, 2014. p. 101–114. ISBN 978-1-931971-09-6. Available from Internet: <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/dobrescu>.

DONG, S. et al. Studying the influence of standard compiler optimizations on symbolic execution. In: **2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.: s.n.], 2015. p. 205–215.

EDWARDS, T. G.; CIARLEGLIO, N. Timestamp-aware rtp video switching using programmable data plan. In: **ACM SIGCOMM**. [S.l.: s.n.], 2017.

FAYAZ, S. K. et al. BUZZ: Testing context-dependent policies in stateful networks. In: **13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)**. Santa Clara, CA: USENIX Association, 2016. p. 275–289. ISBN 978-1-931971-29-4. Available from Internet: <https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/fayaz>.

FOGEL, A. et al. A general approach to network configuration analysis. In: **12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)**. Oakland, CA: USENIX Association, 2015. p. 469–483. ISBN 978-1-931971-218. Available from Internet: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel>.

GEMBER-JACOBSON, A. et al. Fast control plane analysis using an abstract representation. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: ACM, 2016. (SIGCOMM '16), p. 300–313. ISBN 978-1-4503-4193-6. Available from Internet: <http://doi.acm.org/10.1145/2934872.2934876>.

GHASEMI, M.; BENSON, T.; REXFORD, J. Dapper: Data plane performance diagnosis of tcp. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: ACM, 2017. (SOSR '17), p. 61–74. ISBN 978-1-4503-4947-5. Available from Internet: <http://doi.acm.org/10.1145/3050220.3050228>.

HORN, A.; KHERADMAND, A.; PRASAD, M. Delta-net: Real-time network verification using atoms. In: **14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)**. Boston, MA: USENIX Association, 2017. p. 735–749. ISBN 978-1-931971-37-9. Available from Internet: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex>.

JEPSEN, T. et al. Gotthard: Network support for transaction processing. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: ACM, 2017. (SOSR '17), p. 185–186. ISBN 978-1-4503-4947-5. Available from Internet: <http://doi.acm.org/10.1145/3050220.3060603>.

KAZEMIAN, P. et al. Real time network policy checking using header space analysis. In: **Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2013. (nsdi'13), p. 99–112. Available from Internet: <http://dl.acm.org/citation.cfm?id=2482626.2482638>.

KAZEMIAN, P.; VARGHESE, G.; MCKEOWN, N. Header space analysis: Static checking for networks. In: **Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2012. (NSDI'12), p. 9–9. Available from Internet: <http://dl.acm.org/citation.cfm?id=2228298.2228311>.

KHURSHID, A. et al. Veriflow: Verifying network-wide invariants in real time. In: **Proceedings of the First Workshop on Hot Topics in Software Defined Networks**. New York, NY, USA: ACM, 2012. (HotSDN '12), p. 49–54. ISBN 978-1-4503-1477-0. Available from Internet: <http://doi.acm.org/10.1145/2342441.2342452>.

KIRCHNER, F. et al. Frama-c: A software analysis perspective. **Form. Asp. Comput.**, Springer-Verlag, London, UK, UK, v. 27, n. 3, p. 573–609, may 2015. ISSN 0934-5043. Available from Internet: <http://dx.doi.org/10.1007/s00165-014-0326-7>.

LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In: **Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization**. Washington, DC, USA: IEEE Computer Society, 2004. (CGO '04), p. 75–. ISBN 0-7695-2102-9. Available from Internet: <http://dl.acm.org/citation.cfm?id=977395.977673>.

LI, Y. et al. Lossradar: Fast detection of lost packets in data center networks. In: **Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: ACM, 2016. (CoNEXT '16), p. 481–495. ISBN 978-1-4503-4292-6. Available from Internet: <http://doi.acm.org/10.1145/2999572.2999609>.

LIU, H. H. et al. Crystalnet: Faithfully emulating large production networks. In: **Proceedings of the 26th Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2017. (SOSP '17), p. 599–613. ISBN 978-1-4503-5085-3. Available from Internet: <http://doi.acm.org/10.1145/3132747.3132759>.

LOPES, N. et al. **Automatically verifying reachability and well-formedness in P4 Networks**. [S.l.], 2016. Available from Internet: <https://www.microsoft.com/en-us/research/publication/automatically-verifying-reachability-well-formedness-p4-networks/>.

LOPES, N. P. et al. Checking beliefs in dynamic networks. In: **12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)**. Oakland, CA: USENIX Association, 2015. p. 499–512. ISBN 978-1-931971-218. Available from Internet: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes>.

MAI, H. et al. Debugging the data plane with anteater. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 41, n. 4, p. 290–301, aug. 2011. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2043164.2018470>.

PANDA, A. et al. Verifying reachability in networks with mutable datapaths. In: **14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)**. Boston, MA: USENIX Association, 2017. p. 699–718. ISBN 978-1-931971-37-9. Available from Internet: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths>.

SIGNORELLO, S. et al. Ndn.p4: Programming information-centric data-planes. In: **NetSoft**. [S.l.: s.n.], 2016.

SIVARAMAN, A. et al. Dc.p4: Programming the forwarding plane of a data-center switch. In: **Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research**. New York, NY, USA: ACM, 2015. (SOSR ’15), p. 2:1–2:8. ISBN 978-1-4503-3451-8. Available from Internet: <http://doi.acm.org/10.1145/2774993.2775007>.

SON, S. et al. Model checking invariant security properties in openflow. In: IEEE. **2013 IEEE International Conference on Communications (ICC)**. [S.l.], 2013. p. 1974–1979.

STOENESCU, R. et al. Debugging p4 programs with vera. In: **To appear in Proceedings of the 2018 ACM SIGCOMM Conference**. New York, NY, USA: ACM, 2018. (SIGCOMM ’18).

STOENESCU, R. et al. Symnet: Scalable symbolic execution for modern networks. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: ACM, 2016. (SIGCOMM ’16), p. 314–327. ISBN 978-1-4503-4193-6. Available from Internet: <http://doi.acm.org/10.1145/2934872.2934881>.

WEISER, M. Program slicing. In: **Proceedings of the 5th International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 1981. (ICSE ’81), p. 439–449. ISBN 0-89791-146-6. Available from Internet: <http://dl.acm.org/citation.cfm?id=800078.802557>.