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

MIGUEL CARDOSO NEVES

Enforcing properties in programmable networks

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

Advisor: Prof. Dr. Marinho Pilla Barcellos

Porto Alegre
May 2020

CIP — CATALOGING-IN-PUBLICATION

Neves, Miguel Cardoso

Enforcing properties in
programmable networks / Miguel Cardoso Neves. – Porto Alegre: PPGC da UFRGS, 2020.

96 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Marinho Pilla Barcellos.

1. Programmable networks. 2. Network verification. 3. Network debugging. 4. SDN. 5. P4. 6. Monitoring. I. Barcellos, Marinho Pilla. II. Título.

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

“If you always do what you always did,
you will always get what you always got.”
— Albert Einstein

ACKNOWLEDGEMENTS

This thesis would not be possible without the support of many extraordinary people. First of all, I am very thankful to Marinho Barcellos for his patience and commitment to this work. More than an advisor, Marinho became a good friend that has taught me important lessons. I am also very grateful to Kirill Levchenko for co-advising this work. His creativity and pragmatism certainly made this research more interesting and useful. Also, visiting Kirill's research group at UC San Diego was one of the most enriching experiences in my life.

My family and friends also played a central role in this journey. In particular, I am hugely grateful to my parents, Paulo Neves and Rosane Cardoso, to give up from so much in their lives to make my dreams possible, and to my fiancée, Elisandra Pradella, for being by my side at all hours. I was also very fortunate to have good friends sharing the same workspace with me. In special, thanks to Lucas Muller, Fabrício Mazzola, Pedro Marcos, Rodrigo Oliveira and Sérgio Gutiérrez for transforming my work hours (and after hours) into something pleasant and comfortable.

I am thankful to my professors at UFRGS, who actively contributed to my development as a professional and human being. In particular, thanks to all the members of the Computer Networks Research Group, with whom I was fortunate to collaborate and engage in exciting discussions. Finally, thanks to Theophilus Benson, Christian Rothenberg, Luciano Gaspary and Richard Nelson, the committee of my thesis proposal and thesis defenses, for their constructive feedback that helped advance our research and improve this manuscript.

ABSTRACT

Avoiding software bugs and misconfigurations in programmable networks is challenging. Recent studies show that they are among the biggest causes of failures in network infrastructures. Moreover, their consequences can be disastrous. Previous efforts have proposed network debugging and verification techniques as a means to check that the network behaves as expected, but these techniques usually lead to incomplete solutions that can not catch all bugs or face severe scalability issues. In this thesis, we introduce the abstraction of data plane monitors, special modules that allow network programmers to enforce desired properties in a scalable and expressive way. Together with P4box, a system we propose for instrumenting data plane programs with monitors, our abstraction creates an enforcement kernel that cannot be hindered, tampered or circumvented by faulty code. To assess the benefits of our mechanism, we are exploring two use cases: dynamic and static property enforcement. The former is useful when verification does not meet time constraints while the latter enables the verification of previously unfeasible properties. Our experiments using P4box in programmable network hardware show that monitors represent a small overhead in terms of latency and resource consumption when dynamically enforcing a broad range of properties. Moreover, they enable P4box to verify (or statically enforce) reachability properties for large networks ($>$ 190 routers) within a few minutes using off-the-shelf equipment.

Keywords: Programmable networks. network verification. network debugging. SDN. P4. monitoring.

Assegurando propriedades em redes programáveis

RESUMO

Evitar *bugs* e erros de configuração em redes programáveis é um desafio. Estudos recentes mostram que essas estão entre as maiores causas de falhas em infraestruturas de rede. Além disso, as consequências dessas falhas podem ser catastróficas. Trabalhos na literatura propõem técnicas de depuração e verificação de redes como forma de checar se a infraestrutura está funcionando da maneira esperada, mas tais técnicas comumente levam a soluções incapazes de identificar todos os *bugs* e enfrentam sérios problemas de escalabilidade. Nesta tese nós introduzimos o conceito de monitores de planos de dados, módulos especiais que permitem a programadores de rede assegurar propriedades de interesse de forma expressiva e escalável. Juntamente com o sistema que estamos propondo para instrumentar programas de rede com monitores de planos de dados, chamado P4box, nosso mecanismo cria um núcleo de proteção que não pode ser impedido, violado ou evitado por programas sujeitos a falhas. A fim de mostrar os benefícios do nosso mecanismo, exploramos dois casos de uso: assegurar propriedades dinamica e estaticamente. Enquanto o primeiro é útil em cenários onde verificação não consegue atingir restrições de tempo, o segundo permite a verificação de propriedades que não eram possíveis até então. Resultados mostram que monitores de planos de dados implicam numa baixa sobrecarga aos dispositivos de rede em termos de latência e consumo de recursos ao assegurar propriedades dinamicamente. Além disso, eles permitem que o sistema proposto (i.e., P4box) verifique propriedades de rede como atingibilidade entre *hosts* em grandes topologias (com mais de 190 roteadores) em poucos minutos.

Palavras-chave: Redes programáveis, verificação de redes, depuração de redes, SDN, P4, monitoramento.

LIST OF ABBREVIATIONS AND ACRONYMS

ISP	Internet Service Provider
NAT	Network Address Translator
PISA	Protocol Independent Switch Architecture
SDN	Software Defined Network
SEFL	Symbolic Execution Friendly Language
SMT	Satisfiability Modulo Theory
SMV	Symbolic Model Verifier
TTL	Time to Live

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

1.1 Context and motivation

Programmable networks allow operators to modify the behavior of network devices (by reprogramming either the control or the data plane) to quickly deploy new protocols, customize functions or implement advanced services. This flexibility has become mandatory to deal with the increasing scale and traffic demand of applications (YAP et al., 2017). The introduction of network programming languages has greatly lowered the barriers for configuring networks, offering important abstractions and facilitating the specification of complex policies. Today it is reasonable to think that an ecosystem of networking software is emerging, where devices (whether a switch or the network controller) run code written by teams of developers across multiple organizations, assembled by a network operator from libraries and modules to implement a particular set of features.

To reap the benefits of this software ecosystem, network programmers need to ensure that the software they produce behaves correctly. Recent studies show that software bugs and mosconfigurations are the biggest causes of failures in large network infrastructures (up to 60%) (MEZA et al., 2018; GOVINDAN et al., 2016), and the introduction of greater programmability only exacerbates the problem. Fortunately, decades of progress in software engineering and verification have produced mature tools and techniques - from testing to formal methods - for ensuring that software behaves correctly. In the networking domain, these tools are of great importance to reduce the number of incidents, and many of them have been proposed over the last years to enforce the most varied properties (LI et al., 2018). Nevertheless, the problem of network software reliability is far from solved. Despite the simplicity of the network programming model compared to general-purpose software development, e.g., network programming languages usually do not support dynamic memory allocation or even loops (BOSSHART et al., 2014), current state-of-the-art network debugging and verification tools still require considerable amounts of time, skill and effort when applied in production environments.

1.2 Contributions

Most of the complexity for identifying and removing bugs and misconfigurations from network infrastructures comes from three fundamental aspects. First, there is a

complex chain of interactions between the control and the data plane and among network devices themselves. This makes difficult modeling or stressing all possible behaviors the network can present. Second, the scale of current networks result in tools that often hit the wall of theoretical limits (e.g., soundness *versus* completeness, state-space explosion). Even worse, simplifications in network models may lead to unreliable results. Finally, many tools are designed by experts in formal methods and software engineering, which causes these tools to struggle in terms of usability when manipulated by operators and networking practitioners that do not have the same background.

This thesis tackles the problem of preventing failures caused by software bugs and misconfigurations in programmable networks. To that purpose, we rely on two key insights: i) the creation of small, privileged, isolated and safe modules (called monitors) in the network data plane, which are suitable for verification and greatly reduce the scale of the problem; and ii) the usage of abstractions (rather than a new language or complex formalism) largely drawn from P4 – a widespread data plane programming language – to specify properties of interest, which minimizes the burden to network programmers and operators with the learning process. The development of **data plane monitors** is the *first contribution* of the thesis.

Based on these insights we propose P4box, a system for enforcing properties in programmable networks. Our system instruments P4 programs with data plane monitors at compile time. It does this in such a way that monitors can interpose and modify the behavior of the program without being hindered, tampered or circumvented. Moreover, P4box ensures monitors respect a set of desired properties by verifying their code (which is usually much smaller than the original program) through symbolic execution. The design and implementation of **P4box** is the *second contribution* of the thesis.

To show the value of our mechanism, we investigate two use cases. First, we study how to dynamically enforce properties in programmable networks using P4box and its monitors. **Dynamic property enforcement** is particularly useful when network programmers want to import code produced by (potentially untrusted) third parties or when verification does not meet time constraints. In this case, programmers can use monitors to specify additional program blocks devoted exclusively to enforce the desired property. This is the *third contribution* of the thesis.

Finally, we explore how to enable **static property enforcement**[1] in programmable networks using monitors and P4box. We are specially interested in properties that can

[1] We use the terms *verification* and *static enforcement* interchangeably throughout the text.

not be verified (at least in practical times) even by state-of-the-art techniques such as reachability among end hosts. Rather than verifying the whole data plane, we show that it is possible to ensure these properties by verifying only monitors. This is the *fourth contribution* of the thesis.

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 examines programmable networks (including their architecture, main programming languages and desired properties), emphasizes the necessity of property enforcement mechanisms and summarizes existing solutions. Chapter 3 introduces program monitors, describes P4box and discusses its properties and limitations. Chapter 4 explores our first case study, dynamic enforcement, showing how to enforce program and network properties at runtime using P4box and quantifying its overhead to network devices. Chapter 5 then addresses our second case study, static enforcement, describing how to use P4box to scale network verification in the context of programmable data planes. Finally, Chapter 6 presents concluding remarks about our work and outlines research directions for future investigations.

2 BACKGROUND AND RELATED WORK

2.1 Programmable networks

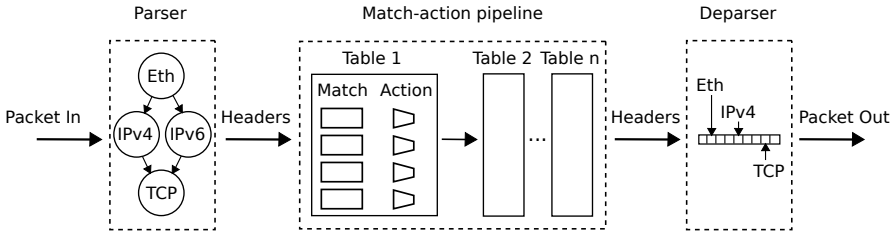
Architecture. A programmable network is one in which the behavior of network devices is handled by software independently from the network hardware (FEAMSTER; REXFORD; ZEGURA, 2014; MACEDO et al., 2015). This idea has evolved over the last thirty years and culminated in an architecture that is driven by two key principles: i) the separation between the control and the data plane; and ii) the programmability of both planes. The control plane is a logically centralized program that acts as an operating system for the network. It usually runs on a set of physically distributed commodity servers and interacts with network elements using an API, e.g., OpenFlow (MCKEOWN et al., 2008) or P4Runtime[1]. In this scenario, a single control software controls multiple data plane elements. The data plane, on the other hand, is implemented as a programmable packet processor present in each network device. This processor is silicon-independent (i.e., can run over an ASIC, FPGA, CPU or GPU) and offers a match + action abstraction that is specified using a high-level programming language, e.g., P4 (BOSSHART et al., 2014). Depending on the control-plane configuration, the data plane of an element can behave like a router, firewall, NAT, load balancer or something in between.

Programmable network devices. In programmable networks, forwarding devices (a.k.a. *targets*) implement variations of the Protocol Independent Switch Architecture - PISA[2]. In this architecture, a device contains multiple programmable blocks, which can be parsers, deparsers, match-action stages or queueing systems. Figure 2.1 presents an example of a PISA-based switch containing three programmable blocks (dashed boxes): a parser, a match-action pipeline and a deparser. Each programmable block is configured by developers using a data plane programming language, and the organization and capabilities of these blocks are abstracted to data plane programs as an interface or *architecture model*. Current examples of programmable network devices include hardware and software switches (SHARMA et al., 2017; SHAHBAZ et al., 2016), FPGA-based packet processing accelerators (WANG et al., 2017), packet filters (HØILAND-JØRGENSEN et al., 2018) and network interface cards - NICs (STEPHENS; AKELLA; SWIFT, 2018).

P4 programming language. Currently, the standard *de facto* language to describe the datapath of programmable network devices is P4. As a domain specific language, P4

[1] <https://p4.org/p4-runtime/>
[2] <https://p4.org/assets/p4-ws-2017-p4-architectures.pdf>

Figure 2.1: Example of PISA-based switch. Dashed blocks can be programmed in P4.



offers many constructs to facilitate the specification of packet processing tasks. Programmers can, for example, declare packet headers, parsers, tables, actions to modify packets, and control blocks to compose sequences of tables. These abstractions are used to configure one or more programmable blocks, and the configuration of all blocks in a device comprises a P4 program. Figure 2.2 shows an example of a program for configuring the PISA-based switch presented in Figure 2.1. We omitted some parts for the sake of simplicity. In this example, the match-action pipeline implements a single table that routes packets based on their IPv4 source and destination addresses (l.8-15). The pipeline block is then composed with the parser and deparser specifications according to the architecture model to form the datapath (l.22). It is worth mentioning that the P4 program is only part of the device configuration. It is still necessary to specify the rules (i.e., the control plane logic) that dictate its forwarding behavior.

Figure 2.2: Example P4 program.

```
1  parser ParserImpl( packet_in packet ){...}
2
3  control Pipeline( inout headers hdr ){
4    ...
5    action route( bit<9> iface ){ ... }
6
7    /* Route IPv4 packets */
8    table route_packet {
9      actions = { route; }
10     key = {
11       hdr.ipv4.srcAddr : ternary;
12       hdr.ipv4.dstAddr : ternary;
13     }
14     size = 1024;
15   }
16
17   apply{ route_packet.apply(); }
18 }
19
20 control DeparserImpl( packet_out packet ){...}
21
22 Switch(ParserImpl(), Pipeline(), DeparserImpl())
```

2.2 Desired properties

Programmable networks are subject to many types of bugs, which can ultimately compromise their security, reliability and performance. On the control plane, bugs usually arise at the controller program and its applications, and are the consequence of a large (sometimes distributed) code base that must deal with many simultaneous events (CANINI et al., 2012; EL-HASSANY et al., 2016). The data plane, on its turn, has a much simpler programming model (e.g., P4 programs have no loops or dynamic memory allocation), but still has demonstrated to be prone to software errors. These errors vary in nature, but overall they can be the consequence of both generic bugs (i.e. well-known from other software development contexts) such as information overwriting[3] and data use-before-initialization[4], and also network specific bugs such as the creation of malformed packets (LOPES et al., 2016), incorrect implementation of protocol specifications (NEVES et al., 2018) or policy violations due to bad table configurations (LOPES et al., 2015; STOENESCU et al., 2016). In this work, we focus on enforcing properties at the data plane, as it implicitly captures all control plane functionality (expressed in the form of forwarding rules) and is comparatively simpler to analyze due to its limited operations (essentially dropping, modifying or forwarding packets).

At a high-level, one can classify the desired properties of a programmable network according to three criteria: i) if it concerns one or multiple devices; ii) if it is generic or associated to an specific program or protocol; and iii) if it involves context or not (ZAOSTROVNYKH et al., 2017; FAYAZ et al., 2016). Below, we define each of these types of properties and give some examples. This is important to understand the capabilities of the state-of-the-art techniques that will be presented in the next section in terms of the classes of bugs and misconfigurations they can prevent.

- **Program *versus* network properties.** Program properties refer to the software running on individual elements in the network (e.g., a switch, router or middlebox) regardless of how they are configured or connected in a topology. Examples include absence of: buffer over/underflows, invalid pointer dereferences, out-of-bounds array indexing, variable use-after-free or use-before-initialization. Network properties, on the other hand, concern the resulting behavior of the network when its devices are combined (i.e., configured and connected) in a particular way. Prop-

[3] <https://github.com/p4lang/switch/issues/97>
[4] <https://github.com/p4lang/switch/pull/102>

erties such as reachability, waypointing and absence of forwarding loops are all in this group.

- **Semantic *versus* general safety properties.** When a property specifies the behavior of an specific program or protocol running in the network it is considered a semantic property. For example, the designer of a router may want to ensure that IPv4 packets have their TTL field decremented on every hop, while NAT programmers would like to check that their implementation conforms to the traditional NAT specification (i.e., RFC 3022). Otherwise, properties are considered general safety ones. Absence of buffer overflows and forwarding loops are also general safety properties.
- **Context-dependent *versus* context-independent properties.** Context-dependent properties consider the presence of stateful elements in a program or network. For instance, the forwarding decision in a stateful firewall typically depends on previous packets seen by the device, so it is necessary to take this state into account when verifying reachability or isolation (e.g., host A can communicate with host B only if host B has initiated a connection with host A). Properties that do not have this dependency are context-independent ones (e.g., TTL decrementation).

Note that we are focusing in this work on a restricted set of boolean related properties that does not take into account, for example, quantities and probabilities. Although being able to enforce non-boolean properties is highly desirable, as we will see in the next section state-of-the-art tools still face serious issues to enforce many boolean invariants on programmable networks, and this is the gap we are trying to fill with this thesis. We leave the investigation of techniques for enforcing quantitative and probabilistic properties in programmable networks as a future work.

2.3 Current enforcement approaches

Tremendous progress has been made towards ensuring that a programmable network does not violate its desired properties. Most of the efforts fall under the scope of two broad techniques: *network verification* and *debugging*. In this section, we present an overview of these efforts to help putting our contributions in perspective. Notice that we do not aim to be exhaustive in our review, but rather offer to the reader an intuition of the main gaps we are trying to fill. We refer to the work of Li et al. (2019) for a more detailed

study[5].

2.3.1 Network verification

Network verification uses formal analysis techniques (e.g., model checking, theorem proving, SAT/SMT solving) to prove that a property holds in the network for any possible state or configuration (i.e., sequence of packets, protocol stack, set of forwarding rules or network events). Network verification tools can be targeted to either the control or the data plane. Control plane tools usually verify the software running in the network controller (i.e., the network operating system and its applications), while data plane tools act directly over the datapath (i.e., code and forwarding rules) of network devices.

There are two main approaches for control plane verification: i) synthesizing verified controllers through programming frameworks; and ii) verifying control programs using automatic generated models. In the former, network programming languages (e.g., NetCore (MONSANTO et al., 2012) and NetKAT (ANDERSON et al., 2014)) have built-in constructs and proven derivations (i.e., theorems and axioms) that allow programmers to encode their properties using the language itself (i.e., there is no need to create a model of the system). The program is then compiled to generate a correct-by-construct network controller. The latter, on the other hand, converts control programs into equivalent models, and use these models to prove the desired properties. For example, Kinetic (KIM et al., 2015b) converts a control program into an SMV model and uses the NuSMV model checker for verifying properties. NICE (CANINI et al., 2012), on its turn, proposes a customized model-checker for OpenFlow-based controllers. Vericon (BALL et al., 2014) converts control programs into first-order logic formulas and checks them using the Z3 theorem prover. Finally, SDNRacer (EL-HASSANY et al., 2016) explores execution traces to build a happens-before model and identify sequences of events that lead to property violations. Table 2.1 summarizes these control plane verifiers.

Data plane verification encompasses checking the program running on a particular network device or in the whole set of devices forming the network topology. Intuitively, verifying a single network device should be much easier than verifying many of them, but it actually depends on the model and technique being adopted as well as the property of interest. For instance, (DOBRESCU; ARGYRAKI, 2014) uses symbolic execution to prove general safety properties (e.g., crash-freedom and bounded execution) on Click

[5] As opposed to us the authors do not cover the P4 landscape.

Table 2.1: Summary of control plane verifiers.

Verifier	Model	Approach
NetCore		Formal semantics
NetKAT		Formal semantics + Kleene algebra with tests
Kinetic	✓	Model checking
NICE	✓	Model checking + symbolic execution
Vericon	✓	SMT solving
SDNRacer	✓	Happens-before graphs

Table 2.2: Summary of data plane verifiers.

Verifier	Network properties	Stateful	Real time	Approach
DOBRESCU et al., 2014		✓		Symbolic execution
VigNAT		✓		Symbolic execution + theorem proving
Hassel	✓			Header space analysis
NoD	✓			Datalog
SymNet	✓	✓		Symbolic execution
VMN	✓	✓		SMT solving
Veriflow	✓		✓	IP-based packet equivalence classes
NetPlumber	✓		✓	Header space analysis + graph algorithms
APKeep	✓		✓	Multi-field packet equivalence classes

elements. Their tool takes around twenty minutes to check the desired properties over an IP router implementation. VigNAT (ZAOSTROVNYKH et al., 2017) takes a step further and proves semantic properties over a NAT implementation (written in C) in less than forty minutes using a hybrid strategy based on symbolic execution and formal theorem proving. In contrast, Hassel (KAZEMIAN; VARGHESE; MCKEOWN, 2012) can verify the existence of forwarding loops in a network containing more than 25 switches and routers in less than 12 minutes through a dedicated algebra computed over header spaces (i.e., the notion of viewing packet headers as points in a geometric space).

Other data plane verifiers include: NoD (LOPES et al., 2015), which allows operators to model networks and properties using Datalog; SymNet (STOENESCU et al., 2016), which proposes a symbolic execution friendly language (SEFL) for modeling networks and symbolically executing these models; and VMN (PANDA et al., 2017), which takes middleboxes into account while checking reachability properties in ISPs and data center networks. Unlike all the previous approaches, Veriflow (KHURSHID et al., 2013), NetPlumber (KAZEMIAN et al., 2013) and APKeep (ZHANG et al., 2020) use customized techniques and data structures (e.g., equivalence classes) to enable real-time network verification (i.e., in the order of seconds or miliseconds). Table 2.2 summarizes these data plane verifiers.

Table 2.3: Summary of P4 verifiers. ◐ = partial support.

Verifier	Network properties	Approach
p4v		SMT solving
assert-p4		Symbolic execution
Vera	◐	Symbolic execution
P4k	◐	K framework
P4nod	◐	Datalog

Finally, many tools try to cope with the need for manually building a new model of the network whenever its data plane changes. For example, ASSERT-P4 (NEVES et al., 2018) automatically converts a P4 program into an equivalent model in C. p4v (LIU et al., 2018), on the other hand, uses SMT constraints to represent the data plane code. Both tools are able to check only program-specific properties though. Vera (STOENESCU et al., 2018) and P4NoD (LOPES et al., 2016) create models of data plane programs that can be used as input to SymNet and NoD, respectively. Although they can quickly verify small data plane programs (i.e., in the order of seconds), the verification time grows exponentially with both the program and the network size. Moreover, they either require programmers to manually compose program models to create a network-wide one or restrict the set of properties that programmers can check (e.g., enforce only reachability and well-formedness properties). P4K (KHERADMAND; ROSU, 2018), a tool that defines an executable semantics for P4 in K, has reported promising results in terms of performance, but it requires from programmers expertise in K for specifying the desired properties, network topology, forwarding rules and even input packets. Table 2.3 provides a summary of these P4-enabled verifiers.

2.3.2 Network debugging

Network debugging involves generating test packets or network events (e.g., link failures) and monitoring the network response. Network debugging tools can perform some or all of these tasks (i.e., probing, monitoring, etc). For example, Pingmesh (GUO et al., 2015) generates probe packets among selected pairs of servers and monitors the network response to diagnose performance and connectivity problems. ATPG (ZENG et al., 2012), on the other hand, focuses on finding the minimum set of packets that exercise every link and forwarding rule in the network. Other tools that involve packet generation include: BUZZ (FAYAZ et al., 2016), which considers the presence of stateful elements

(e.g., middleboxes) in a topology and uses symbolic execution to generate sequences of packets that trigger relevant states; p4pktgen (NöTZLI et al., 2018), which takes into account the programmability of the data plane and generates test packets for P4 programs; and p4rl (SHUKLA et al., 2019), which uses reinforcement learning-guided fuzzing to augment coverage of a P4 program input space (i.e., cover more program paths with less test packets).

Some network debuggers can also systematically create relevant events (e.g., the partition of a distributed network controller, a packet loss or a switch failure) to exercise diverse operational conditions in a network. In this sense, STS (SCOTT et al., 2014) randomly generates sequences of events based on manually assigned probabilities and logs the activity of the network controller in response to those events. The log is then analyzed in order to find the minimal causal sequence that triggered a property violation. Armageddon (SHELLY et al., 2015), on its turn, tries to find the optimal sequence of link failures that enables operators to fail every link (and observe the respective behavior of the control plane) without violating any reachability property in the network. This is important when applying debuggers directly on production environments. Table 2.4 summarizes these network debugging tools.

Table 2.4: Summary of network debugging tools.

Debugger	**Packet probing**	**Failure generation**	**Monitoring**	**P4 support**
Pingmesh	✓		✓	
ATPG	✓			
BUZZ	✓			
p4pktgen	✓			✓
p4rl	✓		✓	✓
STS	✓	✓	✓	
Armageddon		✓	✓	

Finally, many tools involve exclusively monitoring the network infrastructure to catch property violations and their root cause. OpenSketch (YU; JOSE; MIAO, 2013), UnivMon (LIU et al., 2016), FlowRadar (LI et al., 2016), SketchVisor (HUANG et al., 2017) and *Flow ("Star Flow") (SONCHACK et al., 2018) propose new data structures for efficiently storing and manipulating measurement information in network devices. Trumpet (MOSHREF et al., 2016) and PathDump (TAMMANA; AGARWAL; LEE, 2016), on the other hand, make use of end hosts for storing monitoring information rather than forwarding devices. SwitchPointer (TAMMANA; AGARWAL; LEE, 2018) proposes a hybrid approach that combines the visibility of network devices with the greater flexibil-

ity and amount of resources in end hosts.

In addition to the location where monitors run, state-of-the-art tools allow operators to set flexible monitoring campaigns using query languages and programmable frameworks. For example, INT (KIM et al., 2015a) provides a low-level programmable monitoring framework that exposes queue lengths and other performance metadata from switches by piggybacking them on packets. PathQuery (NARAYANA et al., 2016), Marple (NARAYANA et al., 2017) and Sonata (GUPTA et al., 2018) propose high-level languages based on predicates and functional constructs (e.g., map, filter, groupby) that facilitate the expression of complex monitoring tasks. NetQRE (YUAN et al., 2017) and Varanus (NELSON et al., 2016) extend this idea to capture quantitative and stateful network policies, respectively. Finally, Stroboscope (TILMANS et al., 2018) adds the notion of time and schedules measurement tasks according to resource constraints on forwarding devices. Table 2.5 provides a summary of the network monitoring tools we discussed.

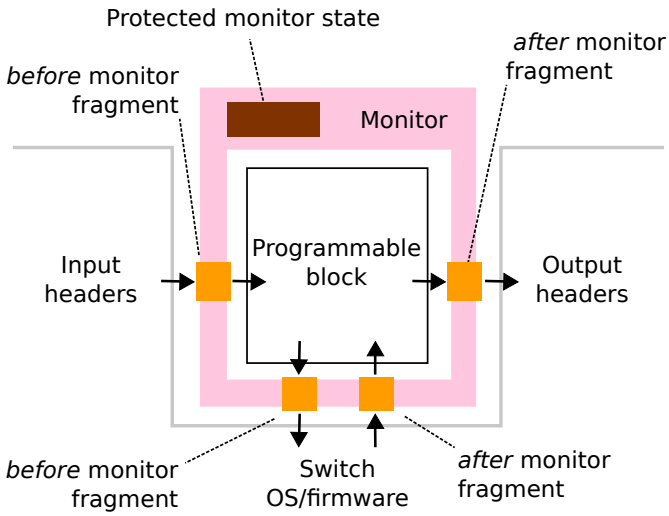
Table 2.5: Summary of network monitoring tools.

Tool	**Custom data structure**	**Custom query language**	**P4 support**
OpenSketch	✓		
UnivMon	✓		✓
FlowRadar	✓		✓
SketchVisor	✓		
*Flow	✓		✓
Trumpet	✓	✓	
PathDump		✓	
SwitchPointer	✓		✓
INT			✓
PathQuery		✓	
Marple		✓	✓
Sonata		✓	✓
NetQRE		✓	
Varanus		✓	
Stroboscope		✓	

3 P4BOX: CREATING AN ENFORCEMENT KERNEL

P4box is a system that allows network programmers to deploy data plane *monitors* in programmable networks. Monitors are privileged, isolated and safe modules that can be attached before and after control blocks, parser state transitions, and calls to external functions of a P4 program. Each monitor can modify the input and output of the code block or function it supervises. This enables the verification of pre- and post-conditions which can be used to enforce specific properties or modify the behavior of the monitored block. P4box instruments the P4 program with its monitors at the intermediate representation level (i.e., during the compilation phase). The resulting program (original code plus monitors) then continues the compilation as before, which allows P4box to be used with any backend compiler based on the $\mathrm{P4}_{16}$ reference implementation. In the rest of this section, we provide an overview of P4box and runtime monitors (Section 3.1), describe the three kinds of monitors P4box can deploy in detail (Sections 3.2.1, 3.2.2 and 3.2.3), show how we verify monitors to ensure their correctness (Section 3.3), and present our prototype implementation (Section 3.4).

Figure 3.1: P4box programming model.



3.1 Overview

A runtime monitor interposes on the interaction of a P4 control block or parser with the rest of the execution environment (Figure 3.1), allowing the monitor programmer to modify the behavior of the enclosed P4 block. A P4 programmable block (either

a control block or parser) interfaces with the rest of the P4 execution environment at entry into the block, return from the block, and at calls to architecture-supplied external functions. In the P4box programming model, when a programmable block is invoked, control first passes to a monitor, also written in P4, before passing to the intended programmable block. Similarly, when a programmable block completes processing, control first passes to the monitor before returning to the device. This allows a monitor to modify the behavior of programmable blocks in a well-defined way.

Monitors can also interpose on calls to external functions: when a programmable block invokes an external function, control first passes to the monitor, then the function, and then back to the monitor again, before returning to the programmable block. A monitor can thus modify the apparent behavior of a external function. Monitors are declared and defined at the top level of a P4 program, alongside control blocks, parser blocks, and other top-level declarations. The syntax for a monitor is:

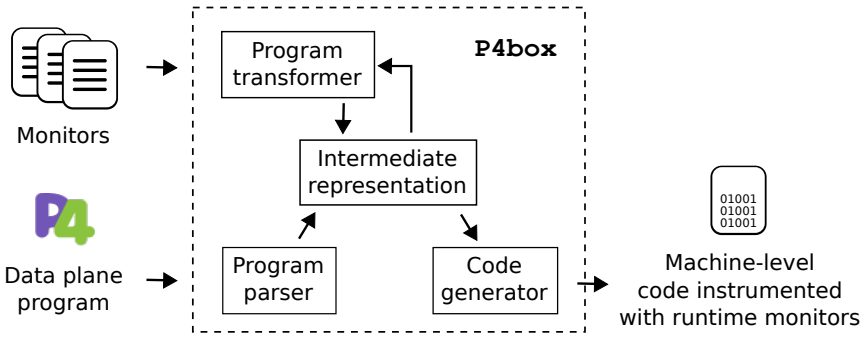
```
monitor <name> ( [param-list] ) on <object> {
    [local-declarations]
    (before | after) { <p4-statements> }
}
```

Each monitor is identified by a unique <*name*> and may receive additional parameters (<*param-list*>) containing headers and metadata in addition to the parameters of the monitored object. Every monitor must be associated with a data plane <*object*>, which can be a parser, control block or extern function. The resource type defines the set of <*p4-statements*> elements the monitor supports (e.g., match-action tables, counters, registers, parser states). Monitors can have two types of methods, namely: *before* and *after*, which specify code fragments that are executed before and after the monitored resource, respectively. Finally, they can also contain local declarations (e.g., actions, tables) visible inside the monitor but not the monitored block.

Figure 3.2 shows the P4box workflow. The original P4 program and P4 source files defining runtime monitors are provided to P4box which combines the original program with the monitors at the intermediate level to produce a new program suitable for further compilation. At the end, machine-level code containing all monitors is generated for a variety of targets. During the instrumentation process, P4box takes advantage of language features provided by P4 such as separate scopes and namespaces in addition to static analysis to provide the following guarantees for each monitor:

- **Complete mediation:** The flow of execution of the original data plane program

Figure 3.2: P4box workflow.



will always pass through a monitor (when one is defined by the programmer). This means it is not possible for the original program to circumvent a monitor.

- **Non-interference:** The original program cannot interfere in the operation of a monitor (e.g., by modifying its local variables or headers), which means monitors are completely isolated from the data plane program.

Together, the complete mediation and non-interference properties allow monitors to restrict what the original P4 program is allowed to do even when the latter is *untrusted*. Hence, monitors are also a form of software sandbox that can be used to encapsulate untrusted or buggy P4 programs. Next, we show examples and describe each of the three kinds of monitors P4box supports in more detail.

Figure 3.3: Example of control block monitor to enforce header protection.

```
1  monitor hdrInvMonitor() on Pipeline {
2    ipv4_t protec_ipv4;
3    udp_t  protec_udp;
4
5    before {
6      protec_ipv4 = hdr.inner_ipv4;
7      protec_udp = hdr.inner_udp;
8    }
9
10   after {
11     if( protec_ipv4 != hdr.inner_ipv4 ||
12         protec_udp != hdr.inner_udp ){
13       /*Run enforcement action
14         (e.g., restore original header
15         value, notify the control plane,
16         write log) */
17   }}
15 }
```

3.2 Data plane monitors

3.2.1 Control block monitors

P4box can attach monitors to top-level control blocks. In this case, *before* and *after* contain statements that will be executed at the beginning and the end of block, respectively. Figure 3.3 shows an example of a control block monitor. This monitor is responsible for ensuring that a header is not erroneously modified by the data plane program. The monitor is attached to the processing pipeline and has two elements: i) before the programmable block, it collects state from the original packet as soon as it is parsed (l.5-8); and ii) after the block, it tests whether monitored headers were modified (l.10-17). Local variables (i.e., visible only to the monitor) are used to store protected headers (l.2-3). If the monitor detects a violation, different actions can be performed to enforce the desired property (e.g., restore the original header value, notify the network controler, log an event), being up to the programmer to decide what to do.

P4box performs the instrumentation of control blocks in three steps: first, monitor parameters containing headers and metadata are merged with parameters of the monitored block (e.g., joining the fields of two structs to create a super struct). If during this process P4box identifies there is no feasible mapping (e.g., because there is no parameter in the monitored block that supports the merge operation), a message is emitted and the instrumentation process is aborted; second, *before* and *after* blocks as well as local declarations are inserted in the monitored block; finally, a name resolution pass maps monitor names to their new namespaces. The left part of Figure 3.4 illustrates this transformation, where a generic control block is instrumented with its monitoring primitives. A corresponding example is shown on the right, representing the instrumentation performed to the monitor specified in Figure 3.3. As a result of this transformation, all packets crossing the control block also pass through the monitor since P4 assumes network devices execute statements in order.

3.2.2 Parser monitors

Parser monitors, on their turn, can be attached to top-level parsers. As such, *before* and *after* can contain finite state machines and both of them must have a start and accept state. It is possible to specialize a parser monitor to an specific parser state, in which

Figure 3.4: Instrumentation of control blocks.

```
control <control_name>
    ( <combined-params> ){
  [local_elements]
  [monitor_local_elements]

  apply{
    [before_statement]
    ...
    [block_statement]
    ...
    [after_statement]
  }
}
```

```
control pipeline(inout newHeaders hdr,
                 inout metadata meta){
  ipv4_t protec_ipv4;
  ...
  apply {
    protec_ipv4 = hdr.inner_ipv4;
    ...
    if(protec_ipv4 != hdr.inner_ipv4
     || protec_udp != hdr.inner_udp){
      ...
    }
  }
}
```

case *before* and *after* are associated only to the latter. An example of a parser monitor is shown in the next chapter (Figure 4.2 – lines 6 to 17), where the monitor is attached to the parse_ethernet state and used to extract an enforcement header. Parser monitors are also particularly useful for extracting packet bits that for some reason (e.g., confidentiality) should not be visible to the data plane program.

To instrument parsers, P4box takes into account if *before* and *after* are attached to states or not. If not, P4box assumes the start and end (i.e., accept) states of the monitored parser as its hooking points. Otherwise, it applies the transformations shown in the left part of Figure 3.5 to the monitored parser. Assuming state S_k is being monitored, P4box links the finite state machine specified inside *before* (before_FSM) between states S_{k-1} and S_k by modifying state transitions. An analogous process is performed for the finite state machine specified inside *after* (after_FSM), linking it between states S_k and S_{k+1}. The right part of Figure 3.5, on its turn, shows an example of these transformations, where P4box performs the instrumentation to the parser monitor specified in Figure 4.2. Instead of transitioning directly from state parse_ethernet to parse_ipv4, the execution flow goes through states _M_START_ and parse_wp_header.

3.2.3 Extern monitors

Extern monitors are attached to extern calls. Their capabilities are restricted to what actions can do in P4 because of limitations the latter have on extern callers (e.g., it is not possible to make local declarations or invoke a table from inside an action). Similar to parser monitors, extern monitors can also be specialized to subgroups of a resource (e.g., a subset of the headers emitted to a packet). In this case, a type signature is used to apply

Figure 3.5: Instrumentation of parsers.

```
parser <parser_name>
     ( <combined-params> ){
  [local_elements]
  [monitor_local_elements]
  ...
  state <s_k-1> {
    transition [before_FSM];
  }
  [state before_FSM {
    transition <s_k> }]
  state <s_k> {
    transition [after_FSM];
  }
  [state after_FSM {
    transition <s_k+1> }]
  state <s_k+1> {
    transition <s_k+2>
  }
  ...
}
```

```
parser pipeline(packet_in packet,
             out newHeaders hdr){
  ...
  state parse_ethernet {
    transition _M_START_;
  }
  state _M_START_ {
    transition select(...){
      16w0xFFFF : parse_wp_header;
      ...
    }
  }
  state parse_wp_header {
    transition parse_ipv4;
  }
  state parse_ipv4 {
    transition parse_tcp;
  }
  ...
}
```

a monitor only to part of the extern calls. An example is presented in Figure 4.2 – lines 20 to 24, where the extern monitor is applied only to calls for emitting headers of type ethernet_t. Extern monitors are useful to mediate how the data plane program interacts with the platform underlying it.

P4box instruments extern calls by adding *before* and *after* blocks right before and after every monitored call, respectively. The left part of Figure 3.6 illustrates this transformation, where the same extern call appears twice (inside an action and directly in the control block body). For the particular case in which a monitor has a type signature, only calls with that signature are instrumented. As an example, the right part of Figure 3.6 shows the instrumentation to the extern monitor specified in Figure 4.2.

Figure 3.6: Instrumentation of extern calls.

```
control <control_name>
    ( <combined-params> ){
  action <action_name>(){
    ...
    [before_statement]
    [extern_A_call]
    [after_statement]
    ...
  }

  apply{
    ...
    [before_statement]
    [extern_A_call]
    [after_statement]
    ...
  }
}
```

```
control DeparserImpl(
          packet_out packet,
          in newHeaders hdr){
  apply{
    ...
    packet.emit(hdr.ethernet);
    packet.emit(hdr.wp_header);
    packet.emit(hdr.ipv4);
    ...
  }
}
```

Figure 3.7: Assertion language grammar.

```
b ::= v
    | f
    | m
    | !b
    | b || b
    | b && b
    | b == b
    | b != b
    | i >= i
    | i <= i
    | i < i
    | i > i
    | i == i
    | i != i

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

3.3 Monitor correctness

Monitors are less likely to contain bugs compared to P4 programs due to their smaller size and complexity. For example, a monitor to enforce header protection has no more than a dozen of lines of code while traditional P4 programs usually have hundreds to thousands of lines (two to three orders of magnitude larger) (STOENESCU et al., 2018; LIU et al., 2018). Despite their simplicity, monitors are still subject to bugs and misconfigurations. For this reason, we developed an automated framework for allowing programmers to check invariants in their specified monitors. In this section, we describe how developers can verify properties in monitors from the same P4 program. We refer to Section 5.2 for details about how we extend this idea for proving network-wide properties on sets of monitors distributed over a network topology.

Our framework is inspired in assert-p4 (NEVES et al., 2018), a state-of-the-art tool for checking invariants in P4 programs. Like assert-p4, our framework is based on assertions and symbolic execution (see Figure 3.9 for its workflow). First of all, programmers annotate monitors with assertions expressing properties of interest. We adopt the same assertion language as proposed in Neves et al. (2018), since monitors are essentially comprised of P4 constructs. Figure 3.7 shows the language grammar. As we can see, each assertion is composed of a boolean expression (b), which may include constant values (v), header fields (f), primitive methods (m) or logical, relational and arithmetic expressions involving these elements.

Note that our concept of assertion is more general than the C-style assertions found in traditional programming languages, and includes both *location-restricted* and *location-unrestricted* elements. A location-restricted element is one that tests the value of a monitor variable where the assertion is specified, as in traditional programming languages like C

or Java. The location-unrestricted ones, in contrast, apply to the entire monitor space. They can be used for example to guarantee higher level properties that the monitors are expected to satisfy, such as data invariance – asserting certain headers are never modified throughout the code.

The methods work as follows. if(b_1, b_2, [b_3]) is similar to traditional conditional statements: if expression b_1 is true, then expression b_2 will be evaluated, otherwise the alternative b_3 will be evaluated). This is the only location restricted method, with all other ones being unrestricted. traverse_path() indicates if a given construct inside a monitor (e.g., an action) will be eventually traversed before the monitor execution ends. constant(f) is true if field f will not change from the assertion location onwards, i.e., until the execution of *all* monitors terminate. forward() returns true if the packet is not dropped after the execution of all monitors. extract_header(h) is true if a header h has been, or will be, extracted from the packet. Finally, emit_header(h) returns true if packet will be transmitted with header h.

Figure 3.8 shows an example assertion (in bold purple – line 6) to the monitor described in Section 3.2.1. The assertion contains a location-unrestricted method and tests whether the *protec_ipv4* variable is not being erroneously modified by the monitor. Once annotated, monitors are assembled in a "virtual program" respecting the same order of execution as the monitored code. This means if monitors A and B are monitoring programmable blocks X and Y, respectively, and X runs before Y, then A will precede B. In addition, the assembled code also contains all header and metadata definitions from the original program, which are treated as symbolic inputs by the verification engine and enable programmers to check invariants on monitors that manipulate program state (e.g., change a header value). After the assembling phase, the new virtual program is translated into an equivalent model in C, and assertions are checked using a symbolic execution tool.

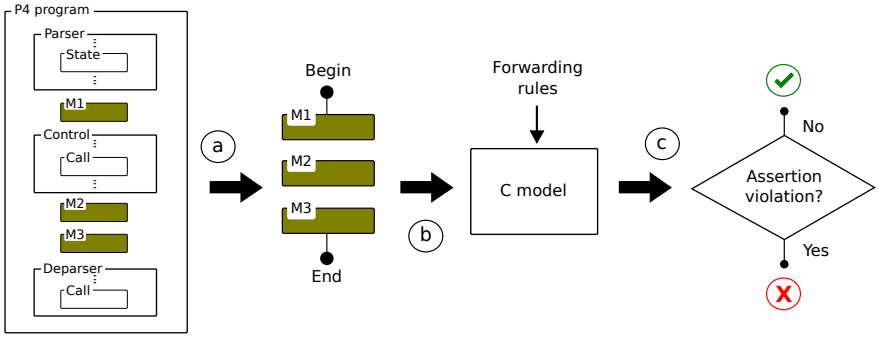
Translating monitors to C allows us to use an off-the-shelf symbolic execution engine, e.g., KLEE (CADAR; DUNBAR; ENGLER, 2008), to check the desired properties. Moreover, tools to ensure the correctness of the translation process are also available[1]. As an example, Figure 3.10 shows the resulting model for the annotated monitor presented in Figure 3.8 (we omit some parts for the sake of simplicity). The *main* code (lines 25-32) controls the call order for the monitors, which are on their turn modeled as additional functions (lines 14-23). We make all monitor inputs (i.e., packet headers, metadata and protected state) symbolic (lines 8-11), so that they can be comprehensively checked by

[1] <https://github.com/gnmartins/assert-p4>

Figure 3.8: Example of annotated monitor.

```
1  monitor hdrInvMonitor() on Pipeline {
2    ipv4_t protec_ipv4;
3    udp_t  protec_udp;
4
5    before {
6      @assert("constant(protec_ipv4)");
7      protec_ipv4 = hdr.inner_ipv4;
8      protec_udp = hdr.inner_udp;
9    }
10
11   after {
12     if( protec_ipv4 != hdr.inner_ipv4 ||
13         protec_udp != hdr.inner_udp ){
14       /*Run enforcement action
15         (e.g., restore original header
16         value, notify the control plane,
17         write log) */
18   }}
19 }
```

Figure 3.9: Workflow for checking monitor correctness. M1, M2, M3 = annotated monitors. a = monitor assembling. b = model extraction. c = symbolic execution.



the symbolic execution engine. Local monitor definitions (e.g. variables and match-action tables) are modeled as unique global constructs (lines 4-5). In particular, each table and action definition is modeled as a separate function. Finally, each assertion is modeled independently, and usually involves variables that are set and tested at relevant points in the program. For example, the assertion modeled in lines 16 and 30 checks whether the monitor, which should ensure a packet header is not modified, is not itself erroneously modifying the header. We refer to Neves et al. (2018) for more details on the translation process. A control plane configuration, expressed in the form of forwarding rules during the model extraction phase, can be considered if a monitor contains one or more match-action tables. Finally, if the verification fails, a trace is generated containing the sequence of commands executed to reach the violated assertion to help programmers correcting the error.

Figure 3.10: Equivalent model in C to the monitor described in Section 3.2.1.

```
1  #include "klee.h"
2
3  //Model monitor locals
4  ipv4_t protec_ipv4;
5  udp_t protec_udp;
6
7  //Make monitor inputs symbolic
8  void symbolizeInputs(){
9    klee_make_symbolic(&hdr, sizeof(hdr), "hdr");
10   klee_make_symbolic(&meta, sizeof(meta), "meta");
11 }
12
13 //Model monitor logic
14 void hdrInvMonitor_before(){
15   protec_ipv4 = hdr.inner_ipv4;
16   constant_protec_var = protec_ipv4;
17   protec_udp = hdr.inner_udp;
18 }
19
20 void hdrInvMonitor_after(){
21   if( protec_ipv4 != hdr.inner_ipv4 ||
22       protec_udp != hdr.inner_udp ){ ... }
23 }
24
25 int main(){
26   symbolizeInputs();
27   hdrInvMonitor_before();
28   hdrInvMonitor_after();
29   //Model assertions
30   hasChanged( constant_protec_var, protec_ipv4 );
31   return 0;
32 }
```

3.4 Implementation

We implemented a prototype of P4box by extending the $\mathrm{P4}_{16}$ reference compiler[2]. Our system has around 3K lines of C++ code and is publicly available[3]. We modified the front-end compiler to: i) instrument programs by adding additional passes over their intermediate representation; and ii) generate our C models.

[2] <https://github.com/p4lang/p4c>
[3] <https://github.com/mcnevesinf/p4box>

4 CASE STUDY: DYNAMIC ENFORCEMENT

The value of a mechanism like P4box is best seen through examples. In this section, we show how P4box can be used to dynamically enforce several kinds of properties in programmable networks.

4.1 Program Properties

As defined in Section 2.2, program properties concern the behavior of a program running on an individual device. These properties must hold regardless of how the device is configured or connected in a topology. They are also referred to as *network function properties* in the literature (ZAOSTROVNYKH et al., 2017). In this work, we consider program properties that are either semantic or general safety properties. Below we show how we enforce two program properties of interest, well-formedness and header protection.

Well-formedness. The output of a data plane program is *well-formed* if it complies with relevant protocol standards. *Well-formedness* determines the interoperability between multiple implementations of a protocol stack. In terms of programmable data planes, this means that the packets produced by one data plane program can be processed by another, and vice-versa. Enforcing well-formedness invariants is particularly useful in hybrid networks (i.e., networks containing both P4-enabled and legacy devices), where the elements may not support the same set of protocols. P4box can enforce well-formedness properties (e.g., packets do not contain both an IPv4 and IPv6 header, ICMP packets always have an IPv4 header) with simple checks of header validity at the end of the processing pipeline.

Header protection. In some cases, it may be desirable to ensure that a header is not modified by a forwarding device or programmable block. For example, in an deployment where VLANs are used to isolate potentially untrusted domains, it may be necessary to provide strong assurance that a VLAN tag is not modified by a forwarding device. P4box can be used to ensure that headers are not modified by collecting the appropriate packet state at the beginning of the processing pipeline (e.g., the value of a VLAN tag), and comparing it against the emitted headers. Such properties can be easily extended to allow only transformations to a pre-defined domain (e.g., source MAC can be modified only to a set of output interface addresses).

4.2 Network-Wide Properties

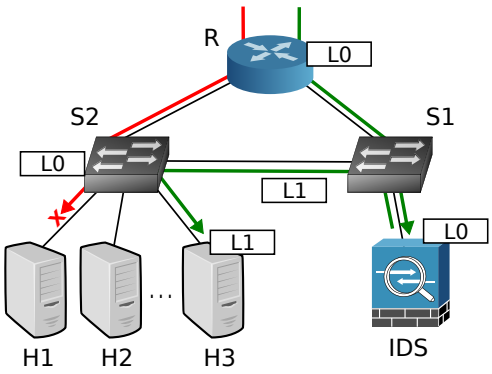
We now describe how P4box can enforce common network-wide properties. As a reminder, these properties concern forwarding devices when configured and connected in a particular topology (see Section 2.2). Although we focus on general safety and context-independent properties in this work, P4box could also be used to enforce semantic and context-dependent (or stateful) ones. We leave exploring the latter as a future work.

Waypointing Network operators may want to force packets to pass through a sequence of devices (waypoints) before the network delivers them to an end host. P4box can enforce waypoint properties by checking and updating labels whenever these packets cross a device in the chain. As an example, Figure 4.1a shows a scenario where packets coming from an external network (i.e., through router R) must first be inspected by an IDS system before arriving at a web server (hosts H1–H3). In this case, a P4box monitor in R introduces labels in each packet in order to enforce waypointing. These labels are then updated by another monitor at switch S1, and a third monitor checks them at switch S2 for dropping packets that are destined to the web servers and do not contain the updated tag (L1). Figure 4.1b shows how P4box interacts with the P4 program to enforce waypointing, where vertical arrows represent the flow of execution. Note that P4box traps the program at three points: first, between the parsing of the Ethernet and IPv4 headers, to check whether the packet contains a label and extract the latter; second, right before the beginning of the match-action pipeline, to operate on the label (e.g., check, updates or remove) depending on how the device is connected in the topology; finally, to emit the label during the deparsing phase.

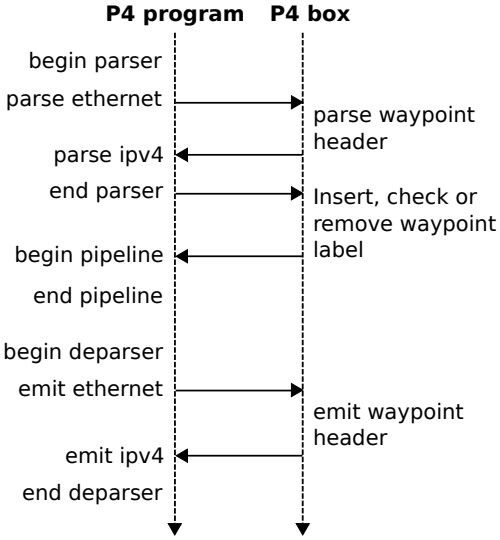
Figure 4.2 shows a summary (with some parts omitted) of the code used to enforce waypoint properties. Each trap is programmed as a separate monitor. Parser (lines 6-17) and extern (lines 20-24) monitors are employed to extract and emit labels, which are declared in the *wp_header* header (line 2). Moreover, a control block monitor uses match-action tables to insert, check/update and remove labels according to the incoming/outgoing ports of the packet. P4box monitors can be configured (proactive or reactively) to reroute packets on-the-fly and correct property violations. Moreover, we can extrapolate the labeling mechanism described above to enforce path conformance (i.e., to guarantee that the actual path taken by a packet conforms to the operator policy). In this case, P4box monitors check and update packet labels on every hop.

Traffic locality. Sometimes operators want to preserve traffic locality, e.g., pack-

Figure 4.1: Enforcing waypointing.



(a) Example topology.

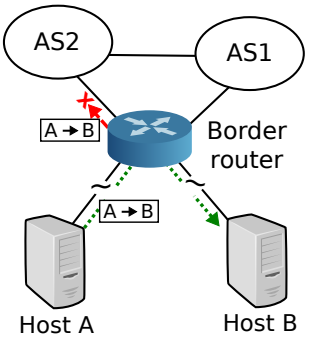


(b) Interaction between P4box and the P4 program.

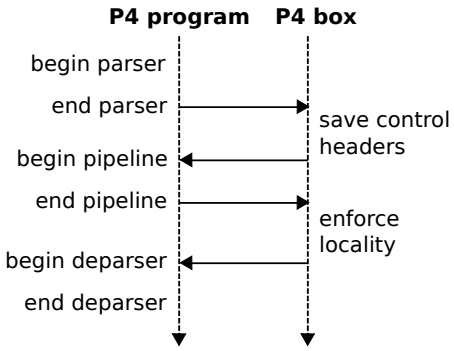
Figure 4.2: Monitors to enforce waypointing.

```
1  struct p4boxState {
2    waypoint_t wp_header;
3  }
4
5  //Parser monitor to extract enforcement header
6  monitor wpParser(inout p4boxState pstate) on ParserImpl {
7    after parse_ethernet {
8      state start {
9        transition select(packet.lookahead<bit<32>>()){
10         16w0xFFFF : parse_wp_header;
11         default : accept;
12       }
13     }
14     state parse_wp_header {
15       packet.extract(pstate.wp_header);
16       transition accept;
17 }}}
18
19 //Extern monitor to emit enforcement header
20 monitor wpExtern(inout p4boxState pstate)
21                              on emit<ethernet_t>{
22   after {
23     packet.emit(pstate.wp_header);
24 }}
25
26 monitor wpControl(inout p4boxState pstate) on Pipeline {
27   ...
28   table check_waypoint {...}
29   ...
30
31   before {
32     //Enforce waypointing property
33     insert_label.apply();
34     check_waypoint.apply();
35     remove_label.apply();
36 }}
```

Figure 4.3: Enforcing traffic locality.



(a) Example topology.



(b) Interaction between P4box and the P4 program.

ets flowing between two VMs in the same rack must not leave the top-of-rack switch in a data center, or traffic between two hosts in the same autonomous system should not leave its borders (LOPES et al., 2015). P4box can enforce traffic locality by controlling the set of output ports a packet can take. For example, packets from host A to B in Figure 4.3a are not allowed to be forwarded to upper ports. Figure 4.3b shows how P4box interacts with the P4 program to enforce traffic locality. First, it hooks the flow of execution at the beginning of the processing pipeline to save the state of required headers (e.g., MPLS or IPv4) before the program can modify them. Then, at the end of the pipeline, it uses the saved state as well as information about the outgoing port to check whether the packet can be forwarded. Figure 4.4 shows relevant parts of the monitor used to enforce traffic locality. It contains a single table that matches a set of control headers and the outgoing port (l.8-16), and runs an enforce_locality action (e.g., send the packet to a different outgoing port) when a violation is detected (l.4).

4.3 Performance

4.3.1 Evaluation Methodology

Because dynamic enforcement happens at run time, it may impose a performance penalty compared with static verification. In this section, we analyze the performance overhead of P4box and show it is small for many useful properties and applications.

Figure 4.5 shows the topology of the setup for evaluating P4box. The device under test (DuT) is equipped with a 4-core Intel Core i3 530 2.93GHz CPU and a single-port

Figure 4.4: Monitor to enforce traffic locality.

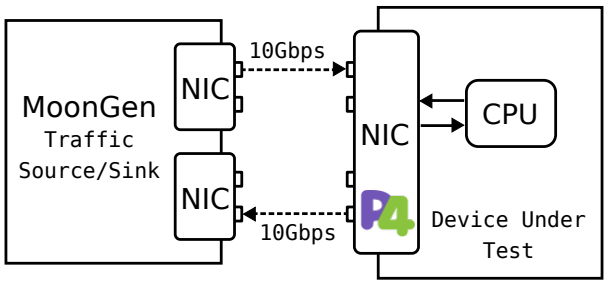
```
1   monitor tlMonitor(inout p4boxState pstate)
2                                    on Pipeline {
3     //Run enforcement action
4     action enforce_locality(){ ... }
5
6     //Check if packet violates locality
7     //(i.e., tries to leave AS)
8     table traffic_locality_table {
9       actions = { NoAction; enforce_locality; }
10      key = {
11        hdr.ipv4.srcAddr : ternary;
12        hdr.ipv4.dstAddr : ternary;
13        standard_metadata.egress_port : exact;
14      }
15      size = 512;
16    }
17
18    after { traffic_locality_table.apply(); }
19  }
```

40G Agilio CX smart NIC running in breakout mode (i.e., 4x10G virtual interfaces). The traffic generator, on its turn, contains a 4-core Intel Xeon E31220 3.1GHz CPU and two dual-port 10G Agilio CX NICs. We configure the traffic generator with MoonGen (EMMERICH et al., 2015) and use a single interface in each NIC for sending and receiving traffic respectively, leaving the other interfaces unused. Unless explicitly mentioned otherwise, our analyses consider the traffic generator creates a 10 Gbps stream of 64-byte UDP packets ($\sim$14.8 million packets per second).

All P4 programs run as embedded firmware in the DuT NIC and are isolated from other end host resources (e.g., CPU, memory and operating system). We use P4box to create instrumented P4 programs and then the Netronome P4 compiler with MAC timestamps and shared content stores enabled to convert instrumented programs into target specific code. Except for Section 4.3.2, in which we analyze the cost of enforcing each property separately, all our experiments assume P4box instruments data plane programs with the four properties described in Sections 4.1 and 4.2, so that we could measure overheads in more demanding conditions.

We measure throughput, latency and power consumption to compare the forwarding performance of the device under test with and without P4box. To measure throughput, we count the number of packets processed in the NIC each second using a P4 counter. We report the average of 10 runs where each run lasts for 30 seconds. To measure the packet processing latency, we collect NIC ingress/egress timestamps and report results over 100 packets. Finally, we use the automated script provided by Netronome (*nic-power*) to read the board power consumption every 100 milliseconds, and similarly to latency mea-

Figure 4.5: Testbed topology. Dashed arrows represent the data flow. Solid arrows indicate control traffic (e.g., for programming the NIC firmware using P4 and collecting statistics).



surements also report results over 100 reads. All measurements are performed after a 5 seconds warm-up interval.

4.3.2 Property overhead

We start looking at the overhead of each property in isolation. To evaluate this overhead, we instrument a very simple data plane program (L3 routing – see Table 4.2) with P4box configured to enforce a single property, and measure the performance drawback compared to a baseline (i.e., the same program without any instrumentation). Table 4.1 shows the latency overhead, in microseconds, for enforcing the properties described in Sections 4.1 and 4.2. As we can see, the overhead is under 5 μs even when we consider all properties together – last line in the table. This is at least one order of magnitude smaller than the latency cost for processing a packet in many data plane applications (see Section 4.3.3). Also, the overhead is clearly not additive, meaning the cost for enforcing a combination of properties is not the same as the sum of the cost for enforcing the individual ones. This is because P4box employs resource sharing among monitors in order to optimize their performance. For instance, the same protected state used to enforce header protection can also be used to enforce traffic locality (see lines 2 and 11 from Figures 3.3 and 4.4, respectively).

4.3.3 Application performance

Next, we evaluate the forwarding performance of the device under test while running real-world applications instrumented with P4box. We select instances of four popu-

Table 4.1: Average, 5th and 95th-percentile latency cost of the properties described in Sections 4.1 and 4.2.

Property	**Latency (us)**		
	Avg	**5th**	**95th**
Well formedness	1.91	1.24	3.61
Header protection	1.32	1.02	2.30
Traffic locality	1.25	1.02	1.80
Waypointing	0.97	0.87	1.40
All 4 properties	2.35	1.74	3.12

Table 4.2: Evaluated applications. LoC = Lines of Code.

Application	**#Tables**	**Stateful**	**LoC**
L3 routing (P4 Consortium, 2018)	3	N	160
Load balancing (SHI et al., 2019)	11	N	420
Surveillance protection (DATTA et al., 2019)	6	N	480
DDoS detection (LAPOLLI; MARQUES; GASPARY, 2019)	2	Y	540

lar applications across different domains:

1. L3 routing, which forwards packets based on destination IP addresses (P4 Consortium, 2018);
2. Load balancing, which uses Othello hashes for mapping virtual IPs (VIPs) to destination servers (DIPs) (SHI et al., 2019);
3. DDoS detection, which adopts counting sketches to identify malicious flows (LAPOLLI; MARQUES; GASPARY, 2019);
4. Surveillance protection, which encrypts IP addresses to obfuscate information about Internet users and devices (DATTA et al., 2019).

Table 4.2 summarizes the P4 programs implementing these applications. Each program has a distinct number of matching tables, which results in different pipeline depths. Moreover, three of the programs do not manipulate any persistent state in the device while the remaining one uses registers for storing packet counts.

Figure 4.6 compares the throughput of the device under test for the evaluated applications. P4box represents a drop of about 9% (1.4 Mpps) for load balancing, 6% (0.9 Mpps) for surveillance protection and 0.7% (0.1 Mpps) for DDoS detection. Interestingly, there was no noticeable overhead for L3 routing as this application was able to achieve line rate in both scenarios.

Figure 4.7 compares the cumulative distribution of the packet processing latency for the different applications. Observe that P4box adds a small latency overhead for pack-

Figure 4.6: Average throughput for the evaluated applications. Standard deviation is less than 0.1 Mpps.

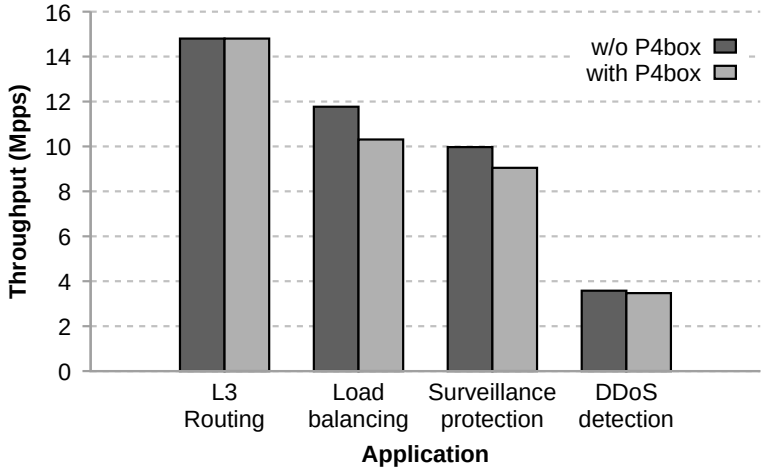
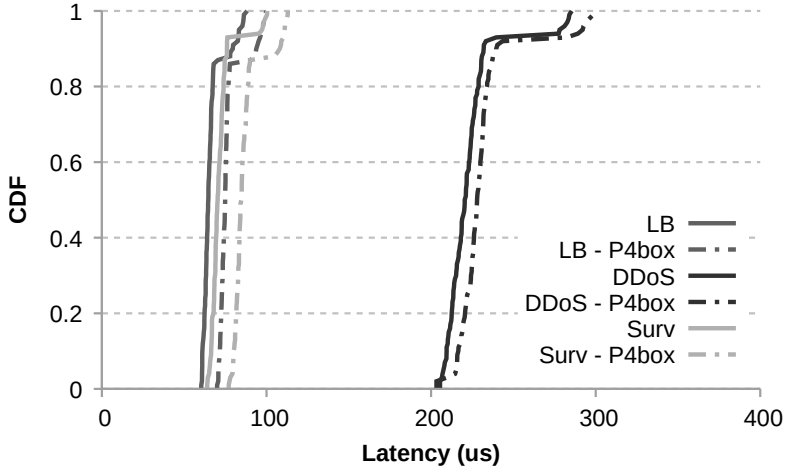


Figure 4.7: CDF of the packet latency for the evaluated applications.

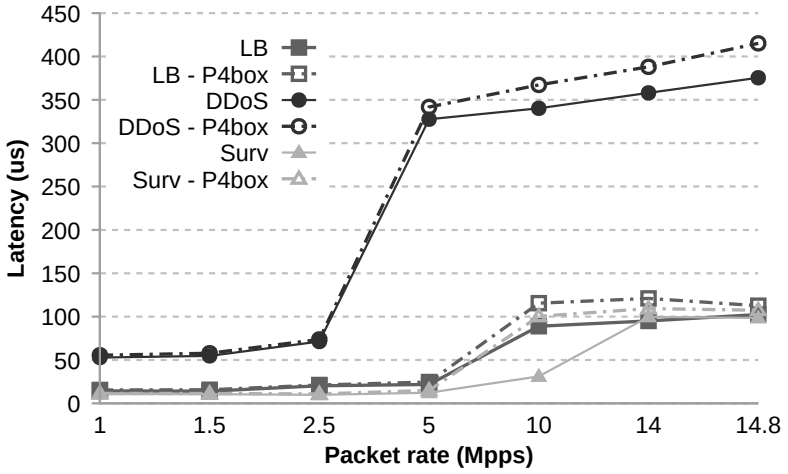


ets and that the overhead depends on the application size/complexity. For example, the increase in the median latency is around 4% for DDoS detection, 15% for load balancing and 19% for surveillance protection. Results are similar when we look at the tail latencies, with an overhead smaller than 15% at the 99th percentile in the worst case (for load balancing). Overall, the more complex the application the lower the penalty for running P4box.

4.3.4 Effect of packet rate

We now turn our attention to examining how different packet rates affect P4box. We consider a maximum load scenario in which the traffic generator sends traffic at the constant rate of 10Gbps, but varies the packet size and consequently the number of packets sent per unit of time. For example, the traffic generator can send up to 14.8 million 64-

Figure 4.8: 95-percentile tail latencies at different packet rates.



byte packets per second, but this number reduces to approximately 1 million if it instead sends packets of 1500 bytes.

Figure 4.8 compares the 95-percentile tail latency for different applications as a function of the packet rate. P4box overhead is negligible up to 5 Mpps. This is because NIC resources are not overloaded at low rates. Above 5 Mpps, P4box increases tail latencies around 20% as a result of bottleneck on NIC. This bottleneck is more prominent in computing-intensive applications such as DDoS detection, where higher processing demands per packet induce a head-of-line (HOL) blocking and consequently queueing formation at input ports (STEPHENS; AKELLA; SWIFT, 2018).

4.3.5 Power consumption

Finally, we evaluate how P4box affects the SmartNIC power consumption. First, we measure the overhead for different link utilizations. We start with an idle system, and gradually increase the input rate until it achieves full link capacity (10 Gbps). Figure 4.9 shows the results for the L3 routing application. As we can see, P4box overhead is smaller than 5% (0.4W) even in the worst case (i.e., when link utilization is maximum). Moreover, this overhead slightly decreases for lower utilizations.

We also measure the overhead for different applications and packet rates. In this case, we consider a line rate scenario where different packet sizes result in different packet rates, but do not affect the link utilization (always 100%) - similarly to the analysis performed in Section 4.3.4. Table 4.3 shows that P4box increases power consumption less than 0.5W for all applications. Interestingly, the overhead is smaller for higher packet

Figure 4.9: Average SmartNIC power consumption for different link utilizations. Standard deviation is less than 0.1W.

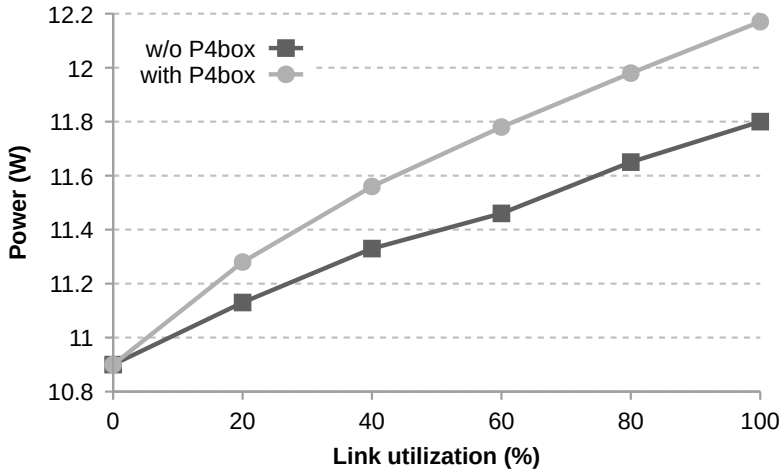


Table 4.3: Average power consumption (in Watts) at line rate for different applications. Standard deviation is less than 0.1W.

Application	**Packet size / rate**					
	64 bytes / 14 Mpps			**1500 bytes / 900 Kpps**		
	w/o P4box	**with P4box**	**%**	**w/o P4box**	**with P4box**	**%**
Load balancing	12.79	12.92	+1.01	11.25	11.25	0
Surveillance protection	12.70	12.74	+0.31	11.21	11.32	+0.98
DDoS detection	12.63	12.71	+0.63	11.21	11.77	+4.99

rates. We believe this is because of the increased packet processing demand, which keeps more processing units (called Micro Engines - MEs in Netronome ASICs (Netronome, 2014)) active/occupied along time for both approaches (i.e., with and without P4box).

5 CASE STUDY: STATIC ENFORCEMENT

Static enforcement involves checking whether a property holds entirely at compile time. Compared to dynamic enforcement, it does not result in any overhead to network devices. However, it can easily become unfeasible enforcing a property statically due to the inherent complexity of the verification problem. For example, many network verification tools are based on model-checking, which is known to suffer from state space explosion (CLARKE et al., 2012).

Unfortunately, none of the state-of-the-art tools for verifying P4 and programmable data planes (LIU et al., 2018; STOENESCU et al., 2018; KHERADMAND; ROSU, 2018; LOPES et al., 2016) can scale to check properties in networks containing more than a dozen devices. In this context, our monitor abstraction can alleviate the verification burden by reducing the problem size. Our intuition is simple: prove that a property holds in the network by checking the smaller monitor space and ensure the latter is not affected by the remaining network configuration. In other words, P4box allows network programmers to *slice their networks* and enables the verification of reduced slices.

5.1 Motivating example

Figure 5.1 shows an example of the benefits of using P4box to check policies in a programmable network. In this example, routers R1-R3 are in the same administrative domain and run a diverse set of network functions (e.g., IP routing, access control, NAT, DDoS detection) implemented in P4. Now consider a scenario where an operator wants to know whether it is possible for a host X connected to router R1 to communicate with (i.e., reach) another host Y connected to router R2 but behind a NAT, as depicted in Figure 5.1a.

Unfortunately, analyzing this scenario is not feasible using existing network verifiers. First, many tools (LOPES et al., 2015; STOENESCU et al., 2016; KHURSHID et al., 2013; PANDA et al., 2017) do not support the automatic verification of programmable data planes or only support checking invariants on a single data plane program (NEVES et al., 2018; LIU et al., 2018). Extending these tools to check a P4-based network would require significant effort and expertise from operators, which is usually not a reasonable assumption. A few recent tools (LOPES et al., 2016; KHERADMAND; ROSU, 2018) do support checking network-wide invariants in programmable networks, but they normally

require long times even for checking a single property in a small topology (only a few nodes). For instance, a recent study has reported it may take several minutes to check that two hosts have the same reachability set (i.e., they can reach the same set of end hosts) in a network programmed with a simple VLAN-like data plane program (less than 100 lines of code) (LOPES et al., 2016). Often, production data plane programs have hundreds to thousands of lines of code (HE et al., 2019).

Unlike existing verifiers, P4box allows programmers to quickly check network-wide invariants (e.g., reachability) in complex topologies containing hundreds of devices running multiple network functions. The key idea is to modularly check slices of the original topology that are flexibly expressed using our monitor abstraction (e.g., a network programmer can use a separate monitor to express each network function). Figure 5.1b illustrates this idea, where the original query was broken down into two independent (and easier to solve) queries. On the left, the programmer can check whether a packet from host X is able to reach router R2, while he can verify whether a packet from X at R2 is correctly translated and forwarded to Y on the right. Note the different *monitor spaces* for both queries: each monitor space represents a network slice and is simpler to verify than the original (complete) network snapshot.

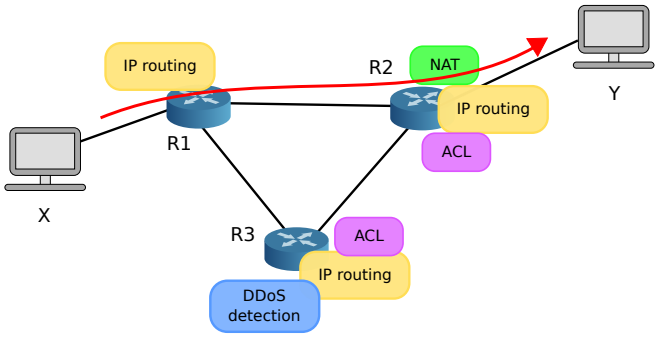
5.2 Modeling networks

We created a simplified network abstraction for extending our monitor models described in Section 3.3 and represent topologies involving more than a single device. Our abstraction is inspired in the Symbolic Execution Friendly Language (SEFL) models proposed by Stoenescu et al. (2016), and takes as input: i) the set of monitors instantiated in each network device; ii) the network topology; and iii) an ingress port of interest.

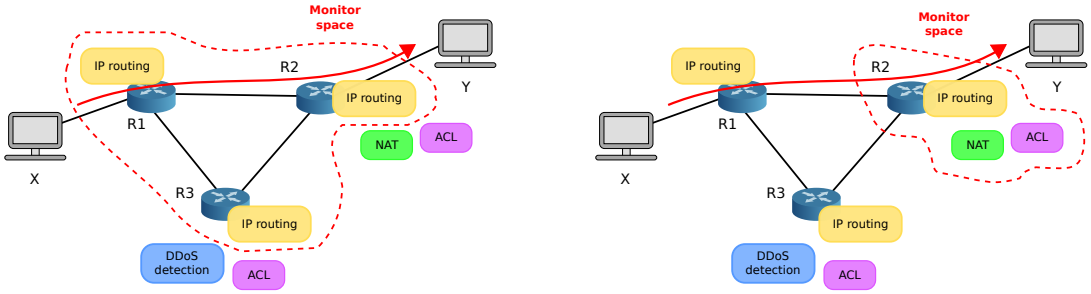
Figure 5.2 shows an example of the proposed abstraction, which represents the network depicted in Figure 5.1. Each network device (e.g., a switch or router) has separate input (shown as a triangle) and output (shown as a rectangle) ports, so two pairs of ports and two links are needed for modeling bidirectional connectivity. Solid arrows indicate network access ports (both ingress and egress) while dashed ones represent internal links (i.e., between two trunk ports). Programmers specify this information to P4box using DOT[1] (a widespread graph description language), where monitors, data plane programs, and sets of forwarding rules are modeled as node attributes.

[1] https://www.graphviz.org/pdf/dotguide.pdf

Figure 5.1: Motivating example to show the benefits of P4box monitors to static property enforcement.



(a) Checking reachability between X and Y. All devices and their configurations must be verified.



(b) Checking reachability in a sliced network using P4box monitors. *Left:* check whether X can reach R2. *Right:* check at R2 whether packets from X can be correctly translated and forwarded to Y.

Similarly to what we do for checking monitor correctness in a single network device, P4box converts the specified network into an equivalent model in C, where each device (or its monitors) is represented as a function. Figure 5.3 illustrates this idea. P4box uses a graph traversal algorithm (Breadth-first search) starting at the specified ingress port to visit every topology node while recursively modeling its connections (line 32). It represents packet headers as a single global structure (line 3) that is modified as the packet travels through the network, and models the transition of a packet from one device to another as assignments between their input and output ports (lines 18-19 and 21-22). Packet headers are made symbolic at the beginning of the model execution (line 28) so that every possible packet is automatically tested at once. We create a separate model for each network access port. Although this decision requires running multiple models for performing an all-paths analysis, it also represents a natural parallelization of the latter.

Figure 5.2: Example of network model adopted by P4box.

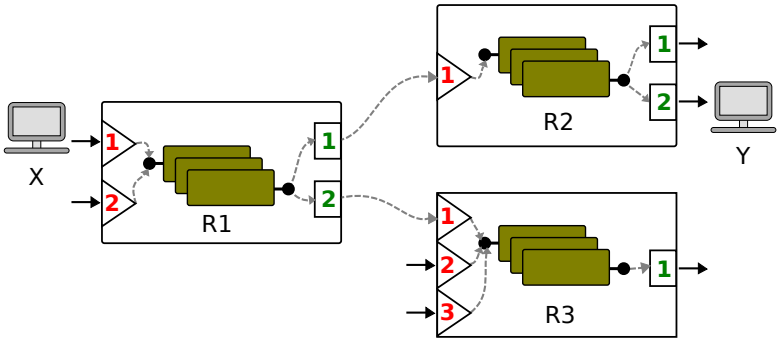


Figure 5.3: Equivalent C model to the topology shown in Figure 5.2.

```
1  std_meta_t std_meta_R1, std_meta_R2, std_meta_R3;
2  metadata meta_R1, meta_R2, meta_R3;
3  headers hdr;
4
5  bool reach_R2_2 = false;
6
7  void run_model_R2(){
8    ...
9    reach_R2_2 = (std_meta_R2.output_port == 2);
10   ...
11 }
12 ...
13 void run_model_R1(){
14   //Run device monitors
15   exec_monitors_R1();
16
17   //Check next device according to output port
18   if(std_meta_R1.output_port == 1){
19      std_meta_R2.input_port = 1;
20      run_model_R2();
21   } else if(std_meta_R1.output_port == 2){
22      std_meta_R3.input_port = 1;
23      run_model_R3();
24   }
25 }
26
27 int main(){
28   make_inputs_symbolic();
29
30   //@assume: packet arrives at ingress port R1.1
31   std_meta_R1.input_port = 1;
32   run_model_R1();
33
34   //Check whether assertion was violated
35   if( !reach_R2_2 ) {
36     assert_error();
37   }
38
39   return 0;
40 }
```

5.3 Optimizations

Although conceptually straightforward, the models described in Section 5.2 still do not scale well when symbolically executed. In particular, P4box translates each match-

Figure 5.4: Optimizing network models by grouping similar rules under the same branch.

Match	Action
192.168.56.1	Forward(1)
192.168.55.123	Forward(2)
192.168.58.45	Forward(2)
192.168.52.170	Forward(2)

(a) Match-action table with exact rules.

```
if ( hdr.ipv4.srcIP == 192.168.56.1 ) {
   Forward(1);
} else if ( hdr.ipv4.srcIP == 192.168.55.123 ) {
   Forward(2);
} else if ( hdr.ipv4.srcIP == 192.168.58.45 ) {
   Forward(2);
} else if ( hdr.ipv4.srcIP == 192.168.52.170 ) {
   Forward(2);
}
```

(b) Original model

```
if ( hdr.ipv4.srcIP == 192.168.56.1 ) {
   Forward(1);
} else if ( hdr.ipv4.srcIP == 192.168.55.123 or
            hdr.ipv4.srcIP == 192.168.58.45 or
            hdr.ipv4.srcIP == 192.168.52.170 ) {
   Forward(2);
}
```

(c) Optimized model.

action table into a series of *If/else* instructions which creates as many branches in the model as the number of forwarding rules in the program. As a consequence, the symbolic execution engine will test a different execution path for each forwarding rule in the network, which easily becomes impractical (current packet classifiers have thousands to hundreds of thousands of rules (LIANG et al., 2019)). In this section, we present a technique (adapted from Stoenescu et al. (2018)) to optimize our models and thus decrease their verification times.

Our technique is based on the fact that many forwarding rules usually trigger the same action invocation. For example, a Forwarding Information Base (FIB) either drops a packet or forwards it to a given output interface. As a result, it is often possible to generate exactly one branch for each distinct action invocation (rather than forwarding rule) by simply grouping similar rules under the same branch. Figure 5.4 illustrates this principle. According to the figure, the four match-action rules in the table (Figure 5.4a) are actually invoking only two different actions and hence can be modeled using two branches (Figure 5.4c) rather than four as in the original model (Figure 5.4b). Note that this optimization is applicable to a wide range of matching strategies including exact, longest-prefix and range matchings. Moreover, symbolic execution also benefits from smaller constraints (STOENESCU et al., 2018; WAGNER; KUZNETSOV; CANDEA, 2013) so our models could be further optimized by combining terms in conditional expressions. We leave that as a future work.

5.4 Enforcing properties

We now briefly describe how programmers can statically enforce a diverse set of *network-wide* properties using P4box.

Reachability. To answer reachability queries, programmers can instrument network egress ports with assertions of the form *"device.output_port == GIVEN_PORT"*. These assertions are then translated into boolean variables which are set and checked at appropriate locations (see Figure 5.3 - lines 5, 9 and 35-37). Isolation is checked by asserting that a given port is *not* reachable.

Waypointing. It is possible to ensure that packets traverse a given network device by checking the *traverse_path()* method at that location. This idea can be extended to check an *unordered chain* of devices by simply replicating the same assertion to other chain elements. Unfortunately, we cannot verify *ordered chains* without extending the assertion language. We leave that as future work.

Bounded path length. To guarantee that packets follow paths of a certain maximum length, programmers can assert that the TTL is not constant and always inside a certain interval at every network device. For example, the assertion "*!constant(ipv4.ttl) && (ipv4.ttl >= 60) && (ipv4.ttl <= 64)*" ensures that traffic will not traverse a path longer than four hops[2]. This assertion can also be extended to encompass different TTL default values (VANAUBEL et al., 2013) by adding more intervals to the formula.

Tunneling. To enforce tunnels are correctly deployed, programmers can assert that tunneling headers are properly inserted/removed at the tunnel endpoits using the *emit_header()* method, while checking that encapsulated packets are not modified by devices inside the tunnel through *constant()* assertions. Note that this approach is valid for different kinds of tunnels (e.g., MPLS, IP-in-IP, GRE).

5.5 Evaluation

In this section, we present a performance evaluation of the static enforcement capabilities of P4box. Our goals are twofold: first, we want to understand how effective P4box monitors are to reduce verification times by slicing data plane programs; second, we want to determine how P4box's performance scale with both network configuration

[2]This assumes the TTL varies monotonically among devices (i.e., always decrease or increase), although a TTL increment still likely indicates a bug. Other scenarios may require extending the assertion language.

and topology size.

5.5.1 Setup

All our experiments were performed on a single-core VM with 4GB of RAM. The VM was running Ubuntu 18.04 and KLEE version 2.1. We considered five topologies from Topology Zoo[3] in our experiments. These topologies range in size from only a few to hundreds of devices and are mostly from ISP networks. We programmed them using a basic L3 routing application (P4 Consortium, 2018) rewritten in P4 using our monitor abstraction, and installed forwarding rules in order to establish routes between random nodes that are uniformly distributed. The exact number of routes (which ranges from 1 to 20K) depends on each evaluation scenario. Also, all packet fields were made symbolic so we can check properties for any kind of traffic at once.

We used the Linux *time* utility to measure the modeling and verification times as well as the memory consumption in all experiments. Unless stated otherwise, each reported measurement is the average of 10 trials for checking reachability assuming the least connected node as the ingress node. Although assuming the least connected node may simplify the verification task at some point because low connected nodes tend to result in less branches in the C model compared to highly connected ones, we also believe this choice reflects better a production end-to-end reachability analysis as more connected nodes usually correspond to core devices (e.g., the ones interconnecting multiple Points-of-Presence or multiple geographically distributed regions in the context of an ISP (YOSHIDA et al., 2009; Knight et al., 2011)). We explore the effect of checking reachability from other nodes on the verification time in Section 5.5.3.

5.5.2 Effectiveness

To demonstrate the effectiveness of P4box in reducing the scale of the verification task, we measure the time taken to model and check a topology as we vary the number of network function (NF) instances. Intuitively, the smaller the network slice (i.e., the number of NF instances it contains) the lower the modeling and verification times. Hence we are interested in analyzing how big a slice can be without overtaking a prohibitive ver-

[3] http://www.topology-zoo.org/

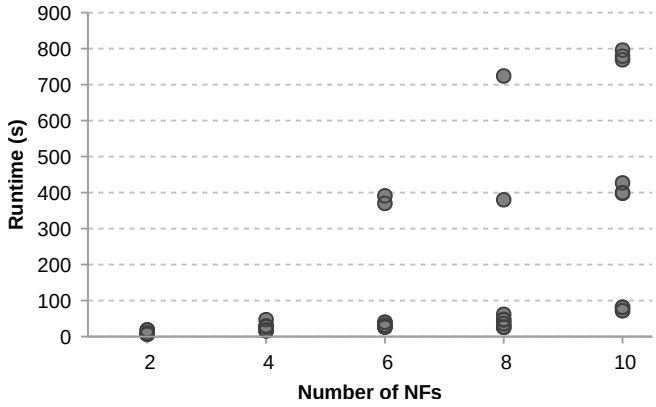
Table 5.1: Average time to generate a C model for different numbers of network function instances. Standard deviation is less than 20 ms.

Number of NFs	Model time (s)
2	0.91
4	0.93
6	0.94
8	0.96
10	0.98

ification cost. We use the ATT topology configured with 1K routes and vary the number of NAT instances from 2 to 10 in these experiments. First, we evaluate the time required to generate a C model. As we can see from Table 5.1, P4box can generate models in less than a second even for a reasonable number of instances. Moreover, the time to generate a model does not vary significantly (∼70 ms in our experiments) with respect to the number of NFs.

Next, we examine the verification time. Figure 5.5 shows it grows quickly when we increase the number of NFs. For example, it may take more than 700 seconds to check a topology with just 8 NAT instances. Interestingly, the verification time grows in steps. This is mainly beacuse the symbolic execution engine must check all possible program paths at each reachable node, meaning the number of paths (and consequently the verification time) steps up whenever the engine reaches a new node that contains a NAT. The same behavior does not apply to lower numbers of NFs (e.g., less than 4) because the probability of reaching a node containing a NAT instance is smaller.

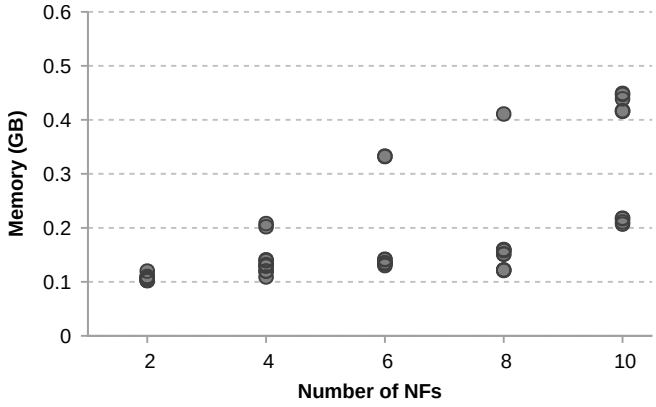
Figure 5.5: Verification time for different numbers of network function instances in the network.



Finally, Figure 5.6 shows the memory consumption for checking the network as we vary the number of NF instances. We observe that P4box requires less than 500 MB in all evaluated scenarios. Moreover, its memory consumption also grows in steps (similarly to the verification time) as each new execution path from a reached node requires storing

additional state, typically in the form of new SMT constraints.

Figure 5.6: Memory consumption for different numbers of network function instances in the network.



5.5.3 Scalability

To determine the scalability of P4box, we examine how efficiently it can model and check different network topologies and their configurations.

Topology. We first check different network topologies while keeping a constant device configuration load (100x the number of network nodes). This means that each network node is expected to be the ingress node for approximately 100 flows at the average. Figure 5.7 shows the time required to model each topology (ordered by their size). We observe that P4box takes only a few seconds (less than 15) to generate a model in the worst case (Cogentco - 197 nodes). Moreover, the modeling time substantially decreases in smaller topologies. For example, it took around 1.2 seconds to model the ATT network (25 nodes). This stems from the fact that there are both less nodes to model as well as optimizations to process.

We also examine how quickly P4box can verify the created models. As we can see from Figure 5.8, it can check reachability properties in less than two minutes for all evaluated networks. Similarly to the model time, we also see a substantial decrease in the verification time as we reduce the network size (e.g., ATT network can be verified in just a few seconds). This is mainly due to a smaller amount of network paths to check. As a future work, we plan to extend our analysis to other network topologies (in particular data center ones) where symmetry characteristics enable us to greatly reduce the verification time even for large instances containing hundreds of thousands of virtual machines (PLOTKIN et al., 2016).

Figure 5.7: Time to create a C model for different network topologies.

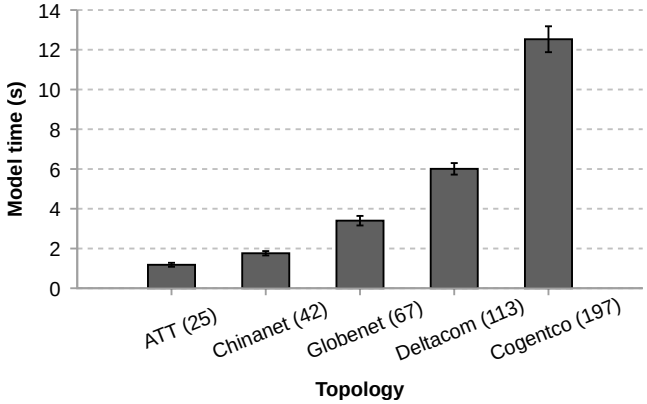
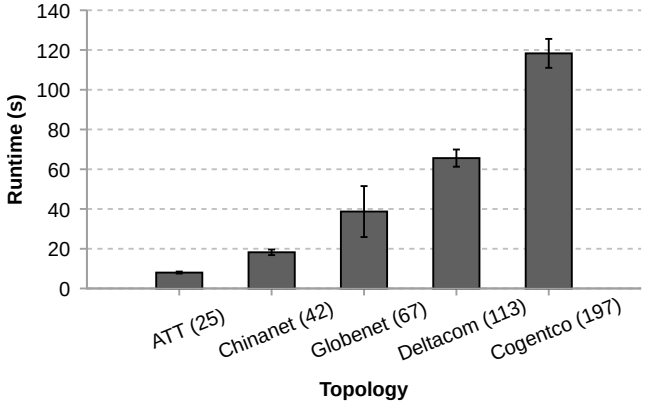
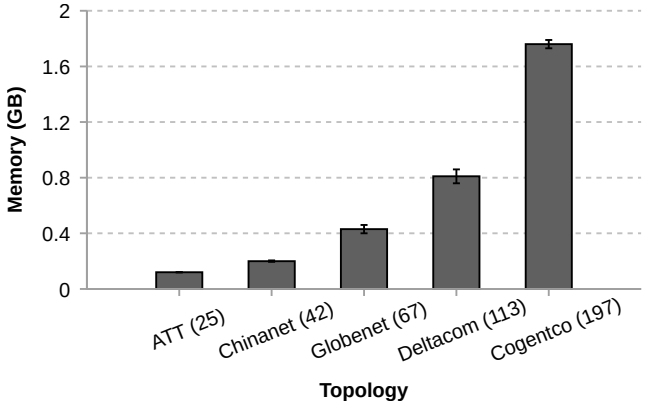


Figure 5.8: Verification time for different network topologies.



Finally, we analyze the memory consumption for checking the evaluated topologies. As Figure 5.9 shows, although it takes 13x more memory to check the largest topology (Cogentco) compared to the smallest one (ATT) in our experiments, memory consumption is still lower than 2 GB in all scenarios. This enables the verification task to be easily performed on a commodity server.

Figure 5.9: Memory consumption for checking different network topologies.



Network configuration. We now turn our attention to analyzing the effect of the network configuration load on P4box performance. To this end, we vary the number

of instantiated routes (and consequently of forwarding rules) while fixing the network topology (ATT). Figure 5.10 shows the time required to model the network as we vary the number of routes. Note that the x axis is in thousands of routes. We observe a linear increase in the model time as the number of routes increases. However, it still takes less than 3 seconds for modeling a network containing 20K routes, which represents a reasonable configuration size in practice (BECKETT et al., 2017). This is a much smaller increase compared to varying the topology size (see Figure 5.7) and stems from the fact that computing model optimizations is simpler than walking through a more complicated topology modeling new nodes.

Figure 5.10: Time to create a C model for different numbers of routes.

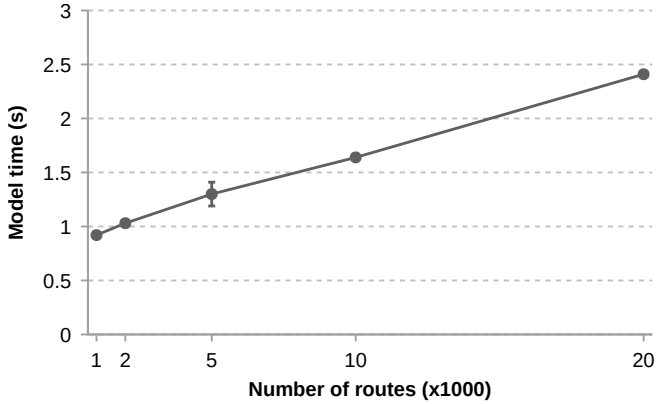


Figure 5.11 shows the time taken by P4box to verify the ATT topology as we vary the number of instantiated routes. As we can see, the verification time grows exponentially despite our optimizations. For example, it takes around 600 seconds to check a network containing 20K routes. This is because of the "branchy" nature of our models which require one conditional statement for each forwarding rule (or group of rules). Other verification tools based on symbolic execution also suffer from the same issue and finding a solution is yet an open research problem (STOENESCU et al., 2018; NöTZLI et al., 2018). A potential alternative would be checking the generated models using a different verification approach, e.g., predicate abstraction or verification conditions (Dahlweid et al., 2009), which performs better in the presence of "branchy" programs. We leave that as a future work.

Figure 5.12 shows the memory consumption of P4box for different numbers of routes. Similarly to the modelling time, we observe memory consumption also grows linearly as we increase the number of routes. Interestingly, it takes only around 900 MB to check a topology configured with 20K routes. However, P4box took 2x that capacity

Figure 5.11: Verification time for different numbers of routes.

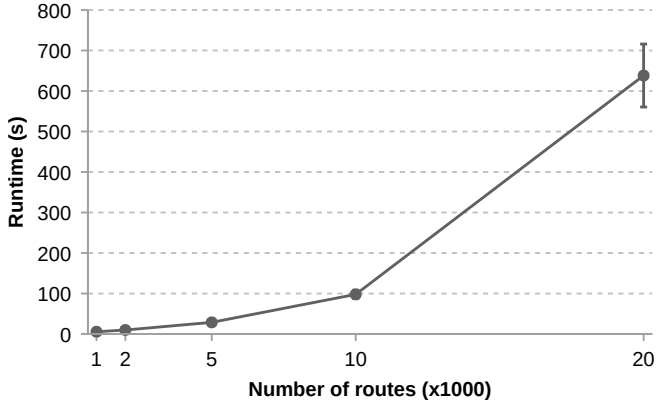
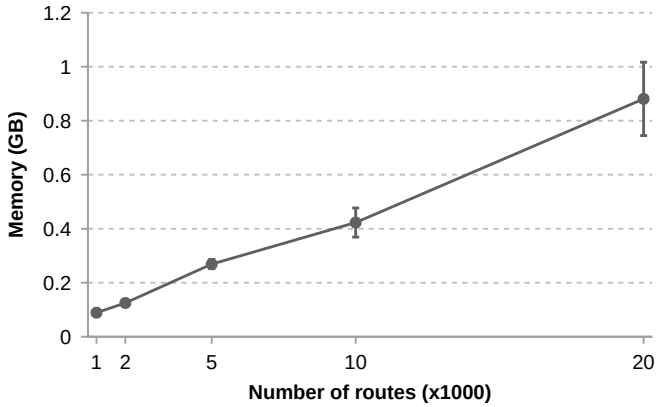


Figure 5.12: Memory consumption for different numbers of routes.



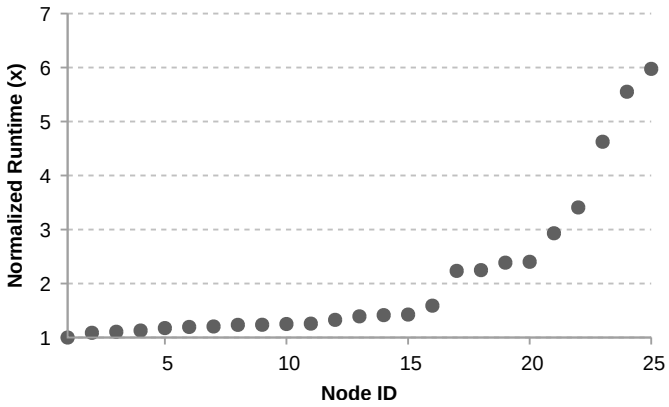
for checking a topology containing 8x more nodes[4] (see "Cogentco" in Figure 5.9), a much lower rate that shows how fast memory consumption still grows in the presence of "branchy" constructs such as match-action tables.

Ingress node. In addition to checking reachability properties with respect to the least connected node in a topology, we also study the effect of taking other nodes as ingress node when building our network models. The intuition is simple: verification time is a function of the number of branches in the model. Although in practice the number of rules (or routes) is the dominant factor in determining the number of branches, the connectivity of the ingress node can also affect this attribute because it makes the symbolic execution tree more flattened. Ultimately, this hinders the chance of cutting off unfeasible branches caused by packet drops before reaching (and thus checking) them.

To test the sensitivity of the verification time to different ingress nodes, we use P4box to verify the reachability from every possible node in the ATT topology configured with 2.5K routes. Figure 5.13 shows the normalized runtime (or verification time) of each node with respect to the least connected node. For the sake of simplicity, we ordered

[4]For approximately the same number of routes.

Figure 5.13: Normalized verification time with respect to the least connected node (Node ID = 0) for all the remaining nodes in the ATT topology.



the nodes (i.e., x axis) according to their verification times. As we can see, although the verification time can grow up to 6x in the worst case, most of the nodes (16 out of 25) have an increase lower than 1.6x showing P4box is feasible even for more densely connected (or loaded) nodes where the relative impact is expected to be greater.

6 CONCLUSION

This chapter presents some final considerations. First, we summarize the main results obtained with our work, revieweing the fundamental challenges addressed and the contributions of this thesis. Second, we briefly show the achievements attained with our research, including a list of publications. Finally, we discuss directions for future investigations that can help further reduce the impact of bugs and misconfigurations in programmable networks.

6.1 Summary

Software bugs and misconfigurations represent up to 60% of all failures in large network infrastructures today (MEZA et al., 2018; GOVINDAN et al., 2016), and the introduction of greater programmability tends to make the problem even worse. Unfortunately, state-of-the-art approaches for network debugging and verification either lead to incomplete solutions or face severe scalability issues. As an example, verifying a programmable data plane can take days to complete even for a single network device (STOENESCU et al., 2018). In this thesis, we have addressed the challenge of avoiding bugs and misconfigurations in programmable networks. To fill this gap, we proposed the abstraction of data plane monitors, small isolated modules that allow programmers to enforce desired properties in a scalable and expressive way. We designed and implemented a system, called P4box, for instrumenting data plane programs with monitors, and showed that it can enforce a broad range of properties (either statically or dynamically).

6.2 Achievements

The work developed in this thesis led to the publication of the following paper in a peer-reviewed conference (full-text can be found in Appendix B). The paper presents data plane monitors, describes P4box and shows how they can be used for dynamically enforcing a broad range of properties in programmable networks. It was nominated a **best paper runner-up** (5 out of 111) of the conference.

- *Dynamic property enforcement in programmable data planes.* **Miguel Neves**, Bradley Huffaker, Kirill Levchenko and Marinho Barcellos. IFIP NETWORKING 2019.

An earlier version of this work received the **third prize in the ACM Student Research Competition at SIGCOMM**. Moreover, an extension describing our monitor correctness checking framework and reporting our main findings after running P4box to enforce properties on a commodity SmartNIC was submitted to IEEE/ACM Transactions on Networking and is under review. We list both papers below (full-text of the journal submission can be found in Appendix C).

- *Sandboxing data plane programs for fun and profit.* **Miguel Neves**, Kirill Levchenko and Marinho Barcellos. ACM SIGCOMM 2017 Poster and Demo Session.
- *Dynamic property enforcement in programmable data planes (extended version).* **Miguel Neves**, Bradley Huffaker, Kirill Levchenko and Marinho Barcellos. IEEE/ACM Transactions on Networking (ToN). *(In submission)*

The work presented in this thesis also evolved from the development of different studies related to programmable networks and network verification. These studies led to the coauthoring of the following peer-reviewed publications:

- *Verification of P4 Programs in Feasible Time using Assertions*. **Miguel Neves**, Lucas Freire, Alberto Schaeffer-Filho and Marinho Barcellos. ACM CoNEXT 2018.
- *Uncovering Bugs in P4 Programs with Assertion-based Verification*. Lucas Freire, **Miguel Neves**, Lucas Leal, Alberto Schaeffer-Filho, Kirill Levchenko and Marinho Barcellos. ACM SOSR 2018.
- *Finding Vulnerabilities in P4 Programs with Assertion-based Verification*. Lucas Freire, **Miguel Neves**, Alberto Schaeffer-Filho and Marinho Barcellos. ACM CCS 2017 Poster Session.

Together, these papers have more than 30 citations at the moment of writing this text, which exemplifies the fact P4box addresses an important problem for which network programmers/operators seek practical solutions. We hope that our available implementation can become a standard tool deployed in production networks in the near future.

Finally, this study also resulted in the development of the following Bachelor Thesis, which explored mechanisms for showing the equivalence between P4 programs and our generated C models.

- *Validating models for verification of P4 programs through symbolic execution*. Martins, G. N. Bachelor Thesis. Federal University of Rio Grande do Sul (UFRGS). 2018.

6.3 Future work

While the work presented in this thesis has made important advances towards creating more reliable programmable networks, there are many directions for future research that can help further reduce the impact of bugs and misconfigurations on these communication infrastructures. In this section, we present some of them.

High-level abstractions. Currently, operators specify their intended properties (or policies) to P4box using a low-level, device-oriented assertion language. While this language is somewhat similar to P4 and thus facilitate its usage by non-experts in formal methods, we believe it is worthwhile to provide operators a more intuitive way to express their network-wide objectives. For example, operators could take advantage of the "one big switch" abstraction (KANG et al., 2013) to express the desired network behavior assuming a single, centralized switch that directly connects all hosts together and then relieve to P4box the task of translating those high-level policies into low-level assertions. Another option could be adopting a more human-oriented network intent language (TIAN et al., 2019; Riftadi; Kuipers, 2019; JACOBS et al., 2018).

Optimizations. Although P4box optimizations can greatly reduce static enforcement times, they are still not enough for allowing operators to check properties in large production networks involving hundreds of devices each containing thousands of lines of code. As such, new optimizations are necessary. Recent studies have focused on developing novel data structures (BJØRNER et al., 2016; KHURSHID et al., 2013) and exploiting topological symmetries (PLOTKIN et al., 2016) to reduce the complexity of the network verification problem. An interesting direction for future work is to create equivalent solutions in the P4 domain (e.g., modeling forwarding rules and match-action tables as trees rather than sequential branches). Another possibility is to investigate symbolic execution specific optimizations such as cutting off branches in our C models by performing data flow or range analysis (SIMON, 2008; KHEDKER; SANYAL; KARKARE, 2009).

Alternative verification techniques. Recent studies have found that symbolic execution often outperforms tools based on deductive verification, except when the model being verified has too many branches (KASSIOS; MÜLLER; SCHWERHOFF, 2012; STOENESCU et al., 2016; LIU et al., 2018; STOENESCU et al., 2018). Unfortunately, this is exactly the case with P4 networks. A broad direction for future work, therefore, is to try to check P4 networks using other verification techniques such as verification condition generation (RAKAMARIć; EMMI, 2014), abstract interpretation (QIAN; XU, 2007)

and symbolic model checking (YANG et al., 2009). State-of-the-art approaches from the program verification domain have also shown promising results when considering a combination of these techniques (SU et al., 2015; YU, 2018), so this is an interesting direction too.

Stateful verification. Current P4 verifiers (including P4box) are restricted to modeling the processing of a single packet and thus can not find property violations caused by sequences of packets (also known as stateful verification or the verification of stateful properties (YUAN et al., 2020)). Enabling programmers to check stateful properties in programmable networks is an important direction for future research. In particular, there is a rich literature covering similar challenges in the context of traditional networks (PANDA et al., 2017; PEDROSA et al., 2018) and/or protocol implementations (Song; Cadar; Pietzuch, 2014; CHI et al., 2017). Extending these efforts to also encompass P4 networks can be a good starting point. In addition, the fact P4box converts P4 programs and their configurations into C code allows us to take advantage of many years of research on C program verification.

Automated repair. While P4box finds policy violations, the network programmer is still responsible to fix them. Moreover, repairing a bug or misconfiguration can be extremely challenging due to the intertwined nature of network policies and protocols (e.g., a fix for a policy violation may trigger another violation for a different traffic class). Assisting programmers in fixing reported violations in P4-based networks is an interesting direction for future work. Interestingly, there is a rich literature on automatically repairing network control planes (GEMBER-JACOBSON et al., 2017; ZHOU et al., 2018; WU et al., 2017) as well as C programs (CHEN; KOMMRUSCH; MONPERRUS, 2019; HAJIPOUR; BHATTACHARYA; FRITZ, 2019; GUPTA et al., 2017) that could be used as starting points for further investigations in our domain.

Network failures. Currently, P4box does not model network failures (e.g., link failures) and thus network operators can only check policies in the presence of failures by manually enumerating each failure scenario. This is very inefficient and even impractical for large topologies. Exploring how P4box can automatically model and check policies in the presence of network failures can alleviate this burden.

REFERENCES

ANDERSON, C. J. et al. Netkat: Semantic foundations for networks. In: **Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 2014. (POPL '14), p. 113–126.

BALL, T. et al. Vericon: Towards verifying controller programs in software-defined networks. In: **Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2014. (PLDI '14), p. 282–293.

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: Association for Computing Machinery, 2017. (SIGCOMM '17), p. 155–168.

BJØRNER, N. et al. ddnf: An efficient data structure for header spaces. In: BLOEM, R.; ARBEL, E. (Ed.). **Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings**. [S.l.: s.n.], 2016. (Lecture Notes in Computer Science, v. 10028), p. 49–64.

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.

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.

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.

CHEN, Z.; KOMMRUSCH, S.; MONPERRUS, M. **Using Sequence-to-Sequence Learning for Repairing C Vulnerabilities**. 2019.

CHI, A. et al. A system to verify network behavior of known cryptographic clients. In: **14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)**. Boston, MA: USENIX Association, 2017. p. 177–195.

CLARKE, E. M. et al. Model checking and the state explosion problem. In: ____. **Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 1–30.

Dahlweid, M. et al. Vcc: Contract-based modular verification of concurrent c. In: **2009 31st International Conference on Software Engineering - Companion Volume**. [S.l.: s.n.], 2009. p. 429–430. ISSN null.

DATTA, T. et al. SPINE: Surveillance protection in the network elements. In: **9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19)**. [S.l.: s.n.], 2019.

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.

EL-HASSANY, A. et al. Sdnracer: Concurrency analysis for software-defined networks. In: **Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2016. (PLDI '16), p. 402–415.

EMMERICH, P. et al. Moongen: A scriptable high-speed packet generator. In: **Proceedings of the Internet Measurement Conference (IMC)**. [S.l.: s.n.], 2015. p. 275–287.

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.

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.

GEMBER-JACOBSON, A. et al. Automatically repairing network control planes using an abstract representation. In: **Proceedings of the 26th Symposium on Operating Systems Principles**. New York, NY, USA: Association for Computing Machinery, 2017. (SOSP '17), p. 359–373.

GOVINDAN, R. et al. Evolve or die: High-availability design principles drawn from googles network infrastructure. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: ACM, 2016. (SIGCOMM '16), p. 58–72.

GUO, C. et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In: **Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2015. (SIGCOMM '15), p. 139–152.

GUPTA, A. et al. Sonata: Query-driven streaming network telemetry. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2018. (SIGCOMM '18), p. 357–371.

GUPTA, R. et al. Deepfix: Fixing common c language errors by deep learning. In: **Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence**. [S.l.]: AAAI Press, 2017. (AAAI'17), p. 1345–1351.

HAJIPOUR, H.; BHATTACHARYA, A.; FRITZ, M. **SampleFix: Learning to Correct Programs by Sampling Diverse Fixes**. 2019.

HE, M. et al. Toward consistent state management of adaptive programmable networks based on p4. In: **Proceedings of the ACM SIGCOMM 2019 Workshop on Networking**

for Emerging Applications and Technologies. New York, NY, USA: Association for Computing Machinery, 2019. (NEAT'19), p. 29–35.

HØILAND-JØRGENSEN, T. et al. The express data path: Fast programmable packet processing in the operating system kernel. In: **Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: ACM, 2018. (CoNEXT '18), p. 54–66.

HUANG, Q. et al. Sketchvisor: Robust network measurement for software packet processing. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. [S.l.: s.n.], 2017. (SIGCOMM '17), p. 113–126.

JACOBS, A. S. et al. Refining network intents for self-driving networks. In: **Proceedings of the Afternoon Workshop on Self-Driving Networks**. New York, NY, USA: Association for Computing Machinery, 2018. (SelfDN 2018), p. 15–21.

KANG, N. et al. Optimizing the "one big switch" abstraction in software-defined networks. In: **Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2013. (CoNEXT '13), p. 13–24.

KASSIOS, I. T.; MÜLLER, P.; SCHWERHOFF, M. Comparing verification condition generation with symbolic execution: An experience report. In: JOSHI, R.; MÜLLER, P.; PODELSKI, A. (Ed.). **Verified Software: Theories, Tools, Experiments**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 196–208.

KAZEMIAN, P. et al. Real time network policy checking using header space analysis. In: **Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)**. Lombard, IL: USENIX, 2013. p. 99–111.

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.

KHEDKER, U.; SANYAL, A.; KARKARE, B. **Data Flow Analysis: Theory and Practice**. 1st. ed. USA: CRC Press, Inc., 2009.

KHERADMAND, A.; ROSU, G. P4K: A formal semantics of P4 and applications. **CoRR**, abs/1804.01468, 2018.

KHURSHID, A. et al. Veriflow: Verifying network-wide invariants in real time. In: **Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)**. Lombard, IL: USENIX, 2013. p. 15–27.

KIM, C. et al. In-band network telemetry via programmable dataplanes. In: **Posters and Demos of the ACM SIGCOMM Symposium on SDN Research (SOSR 15)**. Santa Clara: [s.n.], 2015.

KIM, H. et al. Kinetic: Verifiable dynamic network control. In: **12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)**. Oakland, CA: USENIX Association, 2015. p. 59–72.

Knight, S. et al. The internet topology zoo. **IEEE Journal on Selected Areas in Communications**, v. 29, n. 9, p. 1765–1775, 2011.

LAPOLLI, A.; MARQUES, J. A.; GASPARY, L. Offloading real-time ddos attack detection to programmable data planes. In: **2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**. [S.l.: s.n.], 2019. p. 19–27.

LI, Y. et al. Flowradar: A better netflow for data centers. In: **13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)**. [S.l.: s.n.], 2016. p. 311–324.

LI, Y. et al. A survey on network verification and testing with formal methods: Approaches and challenges. **IEEE Communications Surveys & Tutorials**, PP, p. 1–1, 08 2018.

Li, Y. et al. A survey on network verification and testing with formal methods: Approaches and challenges. **IEEE Communications Surveys Tutorials**, v. 21, n. 1, p. 940–969, 2019.

LIANG, E. et al. Neural packet classification. In: **Proceedings of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGCOMM '19), p. 256–269.

LIU, J. et al. P4v: Practical verification for programmable data planes. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2018. (SIGCOMM '18), p. 490–503.

LIU, Z. et al. One sketch to rule them all: Rethinking network flow monitoring with univmon. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. [S.l.: s.n.], 2016. (SIGCOMM '16), p. 101–114.

LOPES, N. et al. **Automatically verifying reachability and well-formedness in P4 Networks**. [S.l.], 2016.

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.

MACEDO, D. F. et al. Programmable networks—from software-defined radio to software-defined networking. **IEEE Communications Surveys Tutorials**, v. 17, n. 2, p. 1102–1125, Secondquarter 2015. ISSN 1553-877X.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833.

MEZA, J. et al. A large scale study of data center network reliability. In: **Proceedings of the Internet Measurement Conference 2018**. New York, NY, USA: ACM, 2018. (IMC '18), p. 393–407.

MONSANTO, C. et al. A compiler and run-time system for network programming languages. In: **Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 2012. (POPL '12), p. 217–230.

MOSHREF, M. et al. Trumpet: Timely and precise triggers in data centers. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. [S.l.: s.n.], 2016. (SIGCOMM '16), p. 129–143.

NARAYANA, S. et al. Language-directed hardware design for network performance monitoring. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. [S.l.: s.n.], 2017. (SIGCOMM '17), p. 85–98.

NARAYANA, S. et al. Compiling path queries. In: **13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)**. [S.l.: s.n.], 2016. p. 207–222.

NELSON, T. et al. Switches are monitors too!: Stateful property monitoring as a switch design criterion. In: **Proceedings of the 15th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: ACM, 2016. (HotNets '16), p. 99–105.

Netronome. **The Joy of Micro-C**. 2014. Available from Internet: <https://open-nfp.org/m/documents/the-joy-of-micro-c_fcjSfra.pdf>. Accessed: 27-Nov-2019.

NEVES, M. et al. Verification of p4 programs in feasible time using assertions. In: **Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)**. [S.l.: s.n.], 2018. p. 73–85.

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.

P4 Consortium. **Simple router**. 2018. Available from Internet: <https://github.com/p4lang/p4app/tree/master/examples/simple_router.p4app>. Accessed: 16-Nov-2019.

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.

PEDROSA, L. et al. Automated synthesis of adversarial workloads for network functions. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2018. (SIGCOMM '18), p. 372–385.

PLOTKIN, G. D. et al. Scaling network verification using symmetry and surgery. In: **Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 2016. (POPL '16), p. 69–83.

QIAN, J.; XU, B. Formal verification for c program. **Informatica**, IOS Press, NLD, v. 18, n. 2, p. 289–304, abr. 2007. ISSN 0868-4952.

RAKAMARIć, Z.; EMMI, M. Smack: Decoupling source language details from verifier implementations. In: **Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559**. Berlin, Heidelberg: Springer-Verlag, 2014. p. 106–113.

Riftadi, M.; Kuipers, F. P4i/o: Intent-based networking with p4. In: **2019 IEEE Conference on Network Softwarization (NetSoft)**. [S.l.: s.n.], 2019. p. 438–443.

SANGER, R.; LUCKIE, M.; NELSON, R. Identifying equivalent sdn forwarding behaviour. In: **Proceedings of the 2019 ACM Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2019. (SOSR '19), p. 127–139.

SCOTT, C. et al. Troubleshooting blackbox sdn control software with minimal causal sequences. In: **Proceedings of the 2014 ACM Conference on SIGCOMM**. New York, NY, USA: ACM, 2014. (SIGCOMM '14), p. 395–406.

SHAHBAZ, M. et al. Pisces: A programmable, protocol-independent software switch. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: ACM, 2016. (SIGCOMM '16), p. 525–538.

SHARMA, N. K. et al. Evaluating the power of flexible packet processing for network resource allocation. In: **14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)**. Boston, MA: USENIX Association, 2017. p. 67–82.

SHELLY, N. et al. Destroying networks for fun (and profit). In: **Proceedings of the 14th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: ACM, 2015. (HotNets-XIV), p. 6:1–6:7.

SHI, S. et al. **Concury: A Fast and Light-weighted Software Load Balancer**. 2019.

SHUKLA, A. et al. Runtime verification of p4 switches with reinforcement learning. In: **Proceedings of the 2019 Workshop on Network Meets AI & ML**. New York, NY, USA: Association for Computing Machinery, 2019. (NetAI'19), p. 1–7.

SIMON, A. **Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities**. 1. ed. [S.l.]: Springer Publishing Company, Incorporated, 2008.

SONCHACK, J. et al. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In: **2018 USENIX Annual Technical Conference (USENIX ATC 18)**. Boston, MA: USENIX Association, 2018. p. 823–835.

Song, J.; Cadar, C.; Pietzuch, P. Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications. **IEEE Transactions on Software Engineering**, v. 40, n. 7, p. 695–709, 2014.

STEPHENS, B.; AKELLA, A.; SWIFT, M. M. Your programmable nic should be a programmable switch. In: **Proceedings of the 17th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: ACM, 2018. (HotNets '18), p. 36–42.

STOENESCU, R. et al. Debugging p4 programs with vera. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2018. (SIGCOMM '18), p. 518–532.

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.

SU, T. et al. Combining symbolic execution and model checking for data flow testing. In: **Proceedings of the 37th International Conference on Software Engineering - Volume 1**. [S.l.]: IEEE Press, 2015. (ICSE '15), p. 654–665.

TAMMANA, P.; AGARWAL, R.; LEE, M. Simplifying datacenter network debugging with pathdump. In: **Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)**. Savannah, GA: [s.n.], 2016. p. 233–248.

TAMMANA, P.; AGARWAL, R.; LEE, M. Distributed network monitoring and debugging with switchpointer. In: **15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)**. [S.l.: s.n.], 2018. p. 453–456.

TIAN, B. et al. Safely and automatically updating in-network acl configurations with intent language. In: **Proceedings of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGCOMM '19), p. 214–226.

TILMANS, O. et al. Stroboscope: Declarative network monitoring on a budget. In: **Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)**. [S.l.: s.n.], 2018. p. 467–482.

VANAUBEL, Y. et al. Network fingerprinting: Ttl-based router signatures. In: **IMC '13**. [S.l.: s.n.], 2013.

WAGNER, J.; KUZNETSOV, V.; CANDEA, G. Overify: optimizing programs for fast verification. **USENIX Workshop on Hot Topics in Operating Systems**, p. 18–18, 2013.

WANG, H. et al. P4fpga: A rapid prototyping framework for p4. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: ACM, 2017. (SOSR '17), p. 122–135.

Weiser, M. Program slicing. **IEEE Transactions on Software Engineering**, SE-10, n. 4, p. 352–357, 1984.

WU, Y. et al. Automated bug removal for software-defined networks. In: **14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)**. Boston, MA: USENIX Association, 2017. p. 719–733.

YANG, Z. et al. Model checking sequential software programs via mixed symbolic analysis. **ACM Trans. Des. Autom. Electron. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 14, n. 1, jan. 2009. ISSN 1084-4309.

YAP, K.-K. et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 432–445.

YOSHIDA, K. et al. Inferring pop-level isp topology through end-to-end delay measurement. In: MOON, S. B.; TEIXEIRA, R.; UHLIG, S. (Ed.). **Passive and Active Network Measurement**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 35–44.

YU, H. Combining symbolic execution and model checking to verify mpi programs. In: **Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings**. New York, NY, USA: Association for Computing Machinery, 2018. (ICSE '18), p. 527–530.

YU, M.; JOSE, L.; MIAO, R. Software defined traffic measurement with opensketch. In: **Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)**. [S.l.: s.n.], 2013. p. 29–42.

YUAN, Y. et al. Quantitative network monitoring with netqre. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 99–112.

YUAN, Y. et al. Netsmc: A custom symbolic model checker for stateful network verification. In: **17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)**. Santa Clara, CA: USENIX Association, 2020. p. 181–200.

ZAOSTROVNYKH, A. et al. A formally verified nat. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 141–154.

ZENG, H. et al. Automatic test packet generation. In: **Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies**. New York, NY, USA: ACM, 2012. (CoNEXT '12), p. 241–252.

ZHANG, P. et al. Apkeep: Realtime verification for real networks. In: **17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)**. Santa Clara, CA: USENIX Association, 2020. p. 241–255.

ZHOU, W. et al. Automatically correcting networks with neat. In: **15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)**. Renton, WA: USENIX Association, 2018. p. 595–608.

APPENDIX A — RESUMO EXPANDIDO

Neste capítulo, apresentamos um resumo expandido da tese de doutorado. O capítulo está dividido em quatro partes: primeiro, contextualizamos redes programáveis e descrevemos o desafio de se garantir determinadas propriedades nesse tipo de infraestrutura. Segundo, apresentamos uma visão geral do sistema proposto, denominado P4box. Terceiro, discutimos dois casos de uso (verificação dinâmica e análise estática) e os respectivos resultados obtidos através de avaliação experimental. Por fim, indicamos direções relevantes de pesquisa.

Introdução

Redes programáveis permitem a operadores utilizar programas convencionais (ao invés de configurações de baixo nível) para modificar o comportamento da rede, seja através da implantação de novos protocolos no plano de dados ou de novas funções e serviços de rede no plano de controle. Infelizmente, a introdução de novas camadas de *software* também aumenta as chances de ocorrerem *bugs* e erros de configuração nessas infraestruturas. De fato, estudos recentes tem mostrado que *bugs* e erros de confuguração estão entre as principais causas de falhas em redes de grandes provedores (MEZA et al., 2018; GOVINDAN et al., 2016). Diante desse cenário, a presente tese propõe responder à seguinte pergunta de pesquisa:

Como evitar a ocorrência de falhas oriundas de bugs e erros de configuração em redes programáveis?

Propostas do estado-da-arte podem ser agrupadas em duas grandes categorias: verificação e depuração de redes. Verificação de redes busca provar que uma dada propriedade (p.ex., atingibilidade ou isolamento entre hospedeiros) é válida para qualquer pacote, normalmente através da aplicação de métodos tradicionais de verificação formal (p.ex., dedução lógica (ANDERSON et al., 2014), resolução de fórmulas SAT/SMT (BALL et al., 2014) ou verificação de modelos (CANINI et al., 2012)). Embora resulte em uma solução completa (i.e., válida para todos os casos possíveis), tal técnica apresenta sérios problemas de escalabilidade podendo levar horas ou mesmo dias para provar determinadas propriedades em redes de grande porte (i.e., contendo centenas de roteadores) (STOENESCU et al., 2018; LOPES et al., 2015; NEVES et al., 2018).

Depuração de redes, por outro lado, busca validar o comportamento da rede para

um subconjunto de todos os seus possíveis estados. Nesse caso, estratégias de depuração funcionam a partir da geração de eventos de interesse (p.ex. envio de sequências de pacotes ou emulação de falhas em *links*) e do monitoramento da respectiva resposta da rede (p.ex., através da coleta de estatísticas de tráfego) (GUO et al., 2015; FAYAZ et al., 2016; NöTZLI et al., 2018; SHELLY et al., 2015). Embora depuração de redes ofereça maior escalabilidade em comparação à verificação, a mesma possui a desvantagem de não ser capaz de identificar todos os *bugs* ou erros de configuração presentes uma vez que testa somente uma parte de todos os possíveis estados da rede.

P4box

Diante do cenário descrito, esta tese propõe um sistema escalável e completo (i.e., capaz de identificar todo *bug* e/ou erro de configuração quando estes estão presentes) para assegurar propriedades em redes programáveis, denominado P4box. Mais especificamente, o sistema proposto foca em planos de dados programáveis utilizando a linguagem P4 e baseia-se em três ideias-chave: i) o conceito de monitores de planos de dados; ii) a especificação de propriedades através de uma abstração também baseada em P4; e iii) a instrumentação de programas P4 com monitores em tempo de compilação.

Monitores de planos de dados são, por definição, pequenos blocos de código ligados a estruturas programáveis dos dispositivos de rede (p.ex., *parsers* e *pipelines* de processamento de pacotes). Nesse caso, um monitor é capaz de verificar a validade de certas condições imediatamente antes e após a execução do respectivo bloco programável. Além disso, monitores de planos de dados são estruturas garantidamente isoladas (em nível de linguagem) do restante do programa em execução no dispositivo, o que aumenta sua segurança e confiabilidade. A Figura A.1 ilustra esse conceito, onde o monitor é responsável por mediar a interação do bloco programável tanto com cabeçalhos de pacote quanto com os demais componentes do dispositivo (p.ex., sistema operacional e *firmware*).

Para especificar um monitor de plano de dados, programadores de rede podem usar uma estrutura de linguagem definida especificamente para esse fim, cuja sintaxe é mostrada na Figura A.2. Cada monitor possui um identificador único ($< name >$) e deve estar associado a um bloco programável ($< object >$). Pré e pós-condições são especificadas através dos atributos **before** e **after**, respectivamente. Note que ambos atributos podem conter blocos de código P4 (p.ex., tabelas, ações, *parser states*, entre outros). Por fim, é possível declarar variáveis locais a um dado monitor, às quais não poderão ser

Figure A.1: Arquitetura de um monitor de plano de dados.

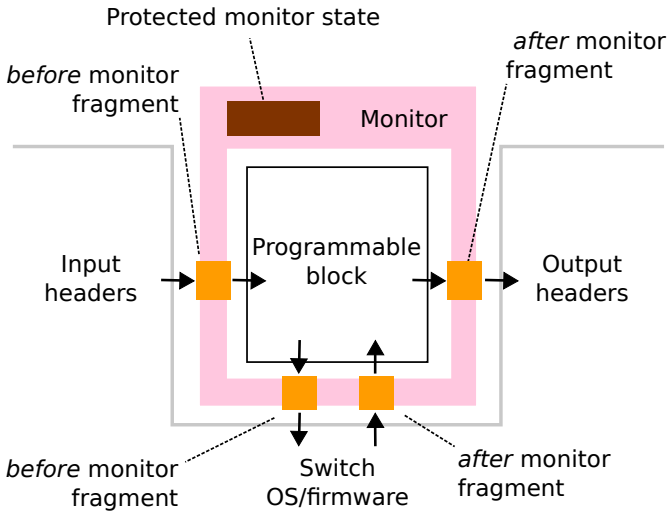


Figure A.2: Sintaxe para especificação de monitores de planos de dados.

```
monitor <name> ( [param-list] ) on <object> {
    [local-declarations]
    (before | after) { <p4-statements> }
}
```

acessadas por nenhuma outra estrutura de código. Para especificar propriedades de interesse, programadores podem usar pré e pós-condições associadas a um ou mais monitores. Uma vez especificados, monitores de planos de dados são automaticamente inseridos por P4box em programas P4 durante o processo de compilação do programa.

Caso de uso: Verificação dinâmica

Verificação dinâmica busca assegurar que uma propriedade é válida numa rede programável em tempo de execução. Essa técnica é particularmente útil quando não se faz possível analisar tal propriedade estaticamente (i.e., em tempo de compilação), ou ainda quando programadores de rede precisam utilizar código (potencialmente não confiável) escrito por terceiros. Nesse caso, monitores de planos de dados contém pequenos trechos de código responsáveis por implementar a verificação de uma propriedade ao invés de um serviço ou protocolo de rede propriamente dito.

Como forma de validação do P4box, este trabalho mostra como utilizar monitores de planos de dados para assegurar quantro diferentes propriedades: proteção de cabeçalhos, formulação correta de pacotes, localidade de tráfego e *waypointing*. Além disso, o trabalho também avalia a sobrecarga de monitores de planos de dados em dispositivos

de rede através de medições de latência e vazão em SmartNICs para quatro diferentes aplicações: roteamento IP (P4 Consortium, 2018), balanceamento de carga (SHI et al., 2019), proteção contra vigilância de redes (DATTA et al., 2019) e detecção de ataques de negação de serviço distribuídos (LAPOLLI; MARQUES; GASPARY, 2019).

As conclusões encontradas são que P4box e monitores de planos de dados geram uma sobrecarga baixa em dispositivo de redes, permitindo a programadores garantirem propriedades de interesse sem afetar o desempenho das aplicações. Mais especificamente, observa-se que há uma queda na vazão de pacotes de no máximo 9%, sendo que em alguns casos essa sobrecarga é negligível (por exemplo, para aplicações demasiadamente simples como roteamento IP não houve variação na vazão com ou sem a presença de monitores de planos de dados). Com relação à latência, observa-se um aumento de até 19% na mediana e de cerca de 15% no percentil 99, mostrando que P4box também não acarreta em um acréscimo significativo no tempo de processamento dos pacotes.

Caso de uso: Análise estática

Ao contrário de verificação dinâmica, análise estática busca assegurar que uma propriedade de rede é válida completamente em tempo de compilação. Nesse caso, é possível utilizar monitores de planos de dados para implementar parte da configuração da rede (p.ex., um subconjunto de suas funções ou protocolos de comunicação) e assegurar que a propriedade é válida perante o conjunto de monitores especificados através de técnicas de verificação formal (p.ex., execução simbólica (STOENESCU et al., 2016) ou prova de teoremas (ZAOSTROVNYKH et al., 2017)). Tal processo também é conhecido como *slicing* e tem sido amplamente estudado na área de desenvolvimento de software para validar programas complexos (Weiser, 1984).

P4box converte um conjunto de monitores associados a uma topologia de rede (ou *slice*) em um programa C equivalente, e utiliza asserções e execução simbólica para garantir a validade de propriedades de interesse perante esse modelo. Como forma de avaliar o desempenho do mecanismo proposto ao efetuar análise estática de redes programáveis, este trabalho analisa entre outros o tempo necessário para se verificar atingibilidade entre hospedeiros em diferentes topologias, configuradas por meio de monitores de planos de dados para realizar roteamento IP.

P4box é avaliado sob diferentes aspectos. Primeiro, estuda-se o impacto do tamanho do *slice* no tempo de verificação através da adição progressiva de instâncias

de funções de rede (NAT) ao *slice*. Observa-se que o tempo de verificação cresce significativamente conforme a quantidade de funções de rede aumenta, chegando a mais de 10 minutos com apenas 8 instâncias em alguns casos (topologia ATT com mil rotas instanciadas). Conclui-se portanto que monitores de planos de dados podem diminuir consideravelmente o tempo de verificação de uma propriedade por meio de análise estática através da criação de *slices* de menor tamanho.

Em seguida, avalia-se a escalabilidade do P4box com relação a diferentes topologias. Considera-se 5 topologias diferentes variando entre 25 e 197 nodos, com uma quantidade de rotas instanciadas pelo menos 100 vezes maior que o tamanho de cada topologia (i.e., cada nodo em uma topologia é responsável por encaminhar ao menos 100 fluxos distintos). Observa-se que P4box é capaz de analisar atingibilidade entre hospedeiros para todas as topologias consideradas em menos de 2 minutos, evidenciando alta escalabilidade com relação a esse fator.

Por fim, analisa-se a escalabilidade do P4box com relação ao número de rotas instanciadas (ou ao tamanho da configuração da rede). Para esse cenário, fixa-se uma determinada topologia (ATT) e varia-se o número de rotas instanciadas entre 1 e 20 mil, uniformemente distribuídas entre pares de nodos. Note que cada rota determina a existência de ao menos uma regra de encaminhamento em cada dispositivo de rede onde passa. Observa-se que o tempo de verificação cresce exponencialmente nesse cenário, consequência do grande número de estruturas condicionais presentes no modelo C resultante. Ainda assim, é possível analisar uma rede configurada com 20 mil rotas em pouco mais de 10 minutos.

Direções de pesquisa

Ao mesmo tempo que esta tese faz avanços importantes em direção a redes programáveis mais confiáveis, há várias direções de pesquisa que ainda podem ser exploradas a fim de continuar reduzindo o impacto de *bugs* e erros de configuração nessas infraestruturas. Por exemplo, atualmente P4box não é capaz de assegurar propriedades envolvendo estado (p.ex., sequências de pacotes). Além disso, apesar das otimizações implantadas, o tempo necessário para se analisar propriedades estaticamente ainda é alto em alguns casos, cabendo a investigação de técnicas de otimização adicionais (p.ex., o uso de diagramas de decisão binária para representar tabelas de encaminhamento (SANGER; LUCKIE; NELSON, 2019)). Por fim, P4box não é capaz de corrigir violações de pro-

priedades (apenas de identificá-las). Nesse caso, extender a ferramenta a fim de auxiliar programadores de rede a reparar programas defeituosos mostra-se uma direção promissora.

APPENDIX B — PAPER AT IFIP NETWORKING 2019

Title: Dynamic property enforcement in programmable data planes

Conference: IFIP Networking 2019

Qualis: A2

Date: May 20-22, 2019

Location: Warsaw, Poland

Abstract: Network programmers can currently deploy an arbitrary set of protocols in forwarding devices through data plane programming languages such as P4. However, as any other type of software, P4 programs are subject to bugs and misconfigurations. Network verification tools have been proposed as a means of ensuring that the network behaves as expected, but these tools typically require programmers to manually model P4 programs, are limited in terms of the properties they can guarantee and frequently face severe scalability issues. In this paper, we argue for a novel approach to this problem. Rather than statically inspecting a network configuration looking for bugs, we propose to enforce networking properties at runtime. To this end, we developed *P4box*, a system for deploying runtime monitors in programmable data planes. Our results show that *P4box* allows programmers to easily express a broad range of properties. Moreover, we demonstrate that runtime monitors represent a small overhead to network devices in terms of latency and resource consumption.

Dynamic Property Enforcement in Programmable Data Planes

Miguel Neves*, Bradley Huffaker†, Kirill Levchenko‡ and Marinho Barcellos*
UFRGS*, CAIDA/UCSD†, UIUC‡

***Abstract*—Network programmers can currently deploy an arbitrary set of protocols in forwarding devices through data plane programming languages such as P4. However, as any other type of software, P4 programs are subject to bugs and misconfigurations. Network verification tools have been proposed as a means of ensuring that the network behaves as expected, but these tools typically require programmers to manually model P4 programs, are limited in terms of the properties they can guarantee and frequently face severe scalability issues. In this paper, we argue for a novel approach to this problem. Rather than statically inspecting a network configuration looking for bugs, we propose to enforce networking properties at runtime. To this end, we developed *P4box*, a system for deploying runtime monitors in programmable data planes. Our results show that *P4box* allows programmers to easily express a broad range of properties. Moreover, we demonstrate that runtime monitors represent a small overhead to network devices in terms of latency and resource consumption.**

I. Introduction

Programmable data planes allow network operators to modify the packet processing pipeline of network devices to quickly deploy new protocols, customize network behavior, and implement advanced network services. The introduction of the P4 [1] programming language has greatly lowered the barriers to doing so, bringing data plane programming into the mainstream. Over the last years, an ecosystem of data plane software has emerged (e.g., [2], [3]), and we can expect to see network devices running code written by teams of developers across multiple organizations, assembled by a network operator from libraries and modules, in the near future.

Despite the simplicity of its programming model, P4 programs have demonstrated to be prone to a variety of bugs and misconfigurations [4], [5]. As a result, network operators need ways to ensure that the programs they produce behave correctly in order to reap the benefits of a data plane software ecosystem. Decades of progress in software engineering have produced mature tools and methodologies for ensuring that certain properties hold in a program, and this idea has been gradually extended to the networking domain. State-of-the-art network verification tools can take a model of the network, its configuration, and a set of properties specified using traditional formalisms (e.g., temporal logic or Datalog rules) and automatically check whether these properties hold for any packet [6], [7].

ISBN 978-3-903176-16-4 ©2019 IFIP

Although these tools have helped network operators to identify bugs before they manifest, they still face important issues that hinder their adoption in production networks. First, most of these tools require programmers to manually model data plane programs, which is a cumbersome and error-prone task [7]. Second, these tools are usually restricted in terms of the properties they can guarantee. For example, some of them are specialized to the verification of reachability properties in order to reduce verification times [8]. Third, more expressive tools capable of verifying multiple properties frequently face severe scalability issues (e.g., checking conformance with a protocol specification can take days even for a single data plane program [4]). Finally, programmers usually have to be proficient in formal verification techniques for correctly specifying their properties.

In this paper, we propose a novel approach to this problem which is based on dynamic (or runtime) enforcement rather than static verification. While the former cannot always provide the kind of strong correctness guarantees that the latter can, it has several practical advantages. First, we do not need to wait for the outcome of a long verification process in order to push a new configuration out to the network switches. In addition, runtime enforcement can promptly intervene if problematic situations actually occur. It means we can still extract some useful work from buggy code when it behaves correctly, and perhaps repair problems without disturbing any network service (see an example in Section IV-B3).

In contrast to static verification, run-time enforcement also lets the developer express policy and mechanism using the same programming environment as the rest of the program. The value of this should not be underestimated: not only does it make life easier for the developer, it also prevents translation errors between implementation and policy domains. That is, rather than expressing a property, such as loop-free forwarding using a separate modeling or formal reasoning language, the programmer can write code to enforce and verify the desired properties in the language of the program (i.e., P4 in our case).

To realize the benefits of our dynamic enforcement approach we developed P4box, a system for deploying runtime monitors in programmable data planes. A *program monitor* is a language construct we developed (as an extension to P4) inspired by the *Aspect-Oriented Programming* (AOP) paradigm [9] which provides language-level constructs for attaching code to designated points in an existing program without modifying the program itself. Programmers can use monitors to modify or verify the behavior of control blocks, parsers, and external

functions of P4 programs, and thus ensure they respect a set of desired properties. Monitors are particularly well-suited to the context in which data plane programs are assembled from externally-maintained modules, where it may be desirable to alter or verify the behavior of these modules without modifying their code.

P4box instruments a P4 program with monitors at compile-time in such a way that the former cannot circumvent or interfere with the latter. Moreover, monitors can be combined to enforce more complex properties such as the ones involving extraction and emition of labels on packets (see an example in Section IV-B1). In summary, we make the following contributions:

- ❖ We design an extension to the P4 data plane programming language, called a *monitor*, that allows a programmer to specify properties about the network (using P4) in the form of pre- and post-conditions to control-blocks, parsers and extern functions (Section III).
- ❖ We develop P4box, a system for deploying runtime monitors in programmable data planes by instrumenting P4 programs at compile-time in such a way that the former cannot be hindered, tampered or circumvented (Section III).
- ❖ We show how P4box can be used to enforce several networking properties, including packet well-formedness, header protection, and waypointing (Section IV).
- ❖ We show that monitors impose low overhead to network devices in terms of latency and memory consumption (Section V).

The remaining of this paper is organized as follows. Section II reviews the architecture of programmable network devices, summarize the main aspects of P4 programs, and motivates the development of property enforcement mechanisms in programmable data planes. Section VI discusses key aspects of runtime enforcement, P4box and program monitors. Section VII compares our proposal with related work, and finally Section VIII concludes the paper.

II. Background and Motivation

A. Programmable network devices

Programmable network devices (a.k.a. *targets*) are packet processing elements (i.e., switches, SmartNICs, NetFPGAs) that allow network programmers to configure their data plane. These devices implement variations of an architecture known as PISA (Protocol Independent Switch Architecture)[1]. PISA-based devices contain multiple programmable blocks, which can be parsers, deparsers, match-action stages or queueing systems. Figure 1 presents an example of a PISA-based switch containing three programmable blocks (dashed boxes): a parser, a match-action pipeline and a deparser. Each programmable block can be configured by developers using a data plane programming language (typically P4), and the organization and capabilities of these blocks are abstracted to P4 programs as an interface or *architecture model*.

[1]https://p4.org/assets/p4-ws-2017-p4-architectures.pdf

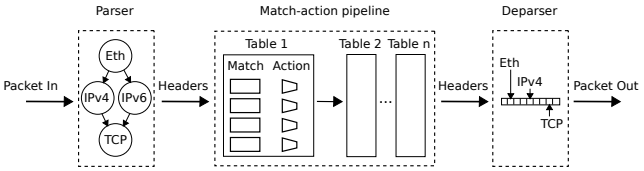


Fig. 1. Example of PISA-based switch. Dashed blocks can be programmed in P4.

```
1  parser ParserImpl( packet_in packet ){...}
2
3  control Pipeline( inout headers hdr ){
4    ...
5    action route( bit<9> iface ){ ... }
6
7    /* Route IPv4 packets */
8    table route_packet {
9      actions = { route; }
10     key = {
11       hdr.ipv4.srcAddr : ternary;
12       hdr.ipv4.dstAddr : ternary;
13     }
14     size = 1024;
15   }
16
17   apply{ route_packet.apply(); }
18 }
19
20 control DeparserImpl( packet_out packet ){...}
21
22 Switch(ParserImpl(), Pipeline(), DeparserImpl())
```

Fig. 2. Example P4 program

B. P4 Programs

As a domain specific language, P4 offers many constructs to facilitate the specification of packet processing tasks. Programmers can, for example, declare packet headers, parsers, tables, actions to modify packets, and control blocks to compose sequences of tables. These abstractions are used to configure different programmable blocks in network devices, and the configuration of all blocks comprises a P4 program. Figure 2 shows an example of a program for configuring the PISA-based switch described in Section II-A. In this example, the match-action pipeline block implements a single table that routes packets based on their IPv4 addresses (l.8-15).

C. Data Plane Bugs

Although the simplicity of its programming model (e.g., P4 programs have no loops or dynamic memory allocation [1]), data plane programs have demonstrated to be prone to many bugs and misconfigurations. Bugs in P4 vary in nature, but overall they can be both generic bugs (i.e. well-known from other programming languages) such as information overwriting[2] and data use-before-initialization[3], and also network specific bugs such as the creation of malformed packets [8], incorrect implementation of protocol specifications [5] or policy violations due to bad table configurations. In this context, it is essential to develop mechanisms that support the development of secure and correct network data planes.

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

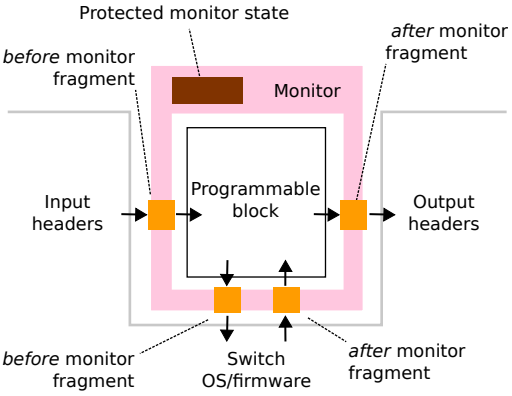


Fig. 3. P4box programming model.

III. P4box

P4box is a system that allows network programmers to deploy runtime *monitors* in programmable data planes. Using P4box programmers can attach monitors before and after control blocks, parser state transitions, and calls to external functions of a P4 program. Each monitor can modify the input and output of the code block or function it monitors. This enables the verification of pre- and post-conditions which can be used to enforce specific properties or modify the behavior of the monitored block. P4box inclines monitor code into the monitored P4 program at the intermediate representation level (i.e., during the compilation of the latter). The resulting program (original code plus monitors) then continues the compilation as before, which allows P4box to be used with any backend compiler based on the $P4_{16}$ reference implementation. In the rest of this section, we provide an overview of P4box and its runtime monitors (Section III-A), describe the three kinds of monitors P4box can deploy in detail (Sections III-B, III-C, and III-D) and present our prototype implementation (Section III-E).

A. Overview

A runtime monitor interposes on the interaction of a P4 control block or parser with the rest of the execution environment (Figure 3), allowing the monitor programmer to modify the behavior of the enclosed P4 block with the rest of the environment. A P4 programmable block (either a control block or parser) interfaces with the rest of the P4 execution environment at entry into the block, return from the block, and at calls to architecture-supplied external functions. In the P4box programming model, when a programmable block is invoked, control first passes to a monitor, also written in P4, before passing to the intended programmable block. Similarly, when a programmable block completes processing, control first passes to the monitor before returning to the device. This allows a monitor to modify the behavior of programmable blocks in a well-defined way.

Monitors can also interpose on calls to external functions: when a programmable block invokes an external function, control first passes to the monitor, then the function, and then back to the monitor again, before returning to the programmable block. A monitor can thus modify the apparent behavior of a external function. Monitors are declared and defined at the top level of a P4 program, alongside control blocks, parser blocks, and other top-level declarations. The syntax for a monitor is:

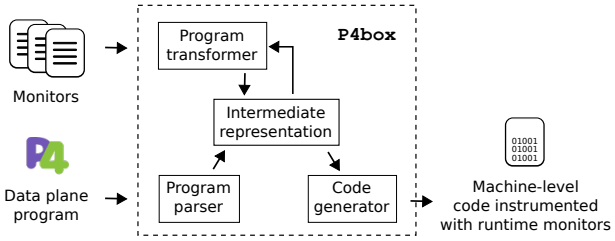


Fig. 4. P4box workflow.

```
monitor <name> ( [param-list] ) on <object> {
    [local-declarations]
    (before | after) { <p4-statements> }
}
```

Each monitor is identified by a unique <*name*> and may receive additional parameters (<*param-list*>) containing headers and metadata in addition to the parameters of the monitored object. Every monitor must be associated with a data plane <*object*>, which can be a parser, control block or extern function. The resource type defines the set of <*p4-statements*> elements the monitor supports. Monitors can have two types of methods, namely: *before* and *after*, which specify code fragments that are executed before and after the monitored resource, respectively. Finally, they can also contain local declarations (e.g., actions, tables) visible inside the monitor but not the monitored block.

Figure 4 shows the P4box workflow. The original P4 program and P4 source files defining runtime monitors are provided to P4box which combines the original program with the monitors at the intermediate level to produce a new program suitable for further compilation. At the end, machine-level code containing all monitors is generated for a variety of targets. During the instrumentation process, P4box takes advantage of language features provided by P4 such as separate scopes and namespaces in addition to static analysis to provide the following guarantees for each monitor:

- **Complete mediation:** The flow of execution of the original data plane program will always pass through a monitor (when one is defined by the programmer). This means it is not possible for the original program to circumvent a monitor.
- **Non-interference:** The original program cannot interfere in the operation of a monitor (e.g., by modifying its local variables or headers), which means monitors are completely isolated from the data plane program.

Together, the complete mediation and non-interference

properties allow monitors to restrict what the original P4 program is allowed to do even when the latter is *untrusted* (e.g., a third-party program). Monitors are thus not only an aspect-oriented P4 program structuring mechanism, but also a software sandbox that can be used to encapsulate untrusted or buggy P4 code. Next, we show examples and describe each of the three kinds of monitors P4box supports in more detail.

B. Control block monitors

P4box can attach monitors to top-level control blocks. In this case, *before* and *after* contain statements that will be executed at the beginning and the end of block, respectively. Figure 5 shows an example of a control block monitor, which could be used to detect and process information overwriting bugs[2]. This monitor is responsible for ensuring that a header is not erroneously modified by the data plane program. The monitor is attached to the processing pipeline and has two elements: i) before the programmable block, it collects state from the original packet as soon as it is parsed (l.5-8); and ii) after the block, it tests whether monitored headers were modified (l.10-17). Local variables (i.e., visible only to the monitor) are used to store protected headers (l.2-3). If the monitor detects a violation, different actions can be performed to enforce the desired property (e.g., restore the original header value, notify the network controler, log an event), being up to the programmer to decide what to do.

P4box performs the instrumentation of control blocks in three steps: first, monitor parameters containing headers and metadata are merged with parameters of the monitored block (e.g., joining the fields of two structs to create a super struct). If during this process P4box identifies there is no feasible mapping (e.g., because there is no parameter in the monitored block that supports the merge operation), a message is emitted and the instrumentation process is aborted; second, *before* and *after* blocks as well as local declarations are inserted in the monitored block; finally, a name resolution pass maps monitor names to their new namespaces. The left part of Figure 6 illustrates this transformation, where a generic control block is instrumented with its monitoring primitives. A corresponding example is shown on the right, representing the instrumentation performed to the monitor specified in Figure 5. As a result of this transformation, all packets crossing the control block also pass through the monitor since P4 assumes network devices execute statements in order.

C. Parser monitors

Parser monitors, on their turn, can be attached to top-level parsers. As such, *before* and *after* can contain finite state machines and both of them must have a start and accept state. It is possible to specialize a parser monitor to a specific parser state, in which case *before* and *after* are associated only to the latter. An example of a parser monitor is shown in Figure 11-lines 6 to 17, where the monitor is attached to the parse_ethernet state and used to extract an enforcement header. Parser monitors are also particularly useful for skipping the extraction of packet

```
1  monitor hdrInvMonitor() on Pipeline {
2    ipv4_t protec_ipv4;
3    udp_t  protec_udp;
4
5    before {
6      protec_ipv4 = hdr.inner_ipv4;
7      protec_udp = hdr.inner_udp;
8    }
9
10   after {
11     if( protec_ipv4 != hdr.inner_ipv4 ||
12         protec_udp != hdr.inner_udp ){
13       /*Run enforcement action
14         (e.g., restore original header
15         value, notify the control plane,
16         write log) */
17   }}
15 }
```

Fig. 5. Example of control block monitor to enforce header protection.

bits that for some reason (e.g., confidentiality) should not be visible to the data plane program.

To instrument parsers, P4box takes into account if *before* and *after* are attached to states or not. If not, it assumes the start and end (i.e., accept) states of the monitored parser as its hooking points. The left part of Figure 7 shows the transformations P4box applies. Assuming state S_k is being monitored, P4box links the finite state machine specified inside *before* (before_FSM) between states S_{k-1} and S_k by modifying state transitions. An analogous process is performed for the finite state machine specified inside *after* (after_FSM), linking it between states S_k and S_{k+1}. The right part of Figure 7, on its turn, shows an example of these transformations, where P4box performs the instrumentation to the parser monitor specified in Figure 11. Instead of transitioning directly from state parse_ethernet to parse_ipv4, the execution flow goes through states _M_START_ and parse_wp_header.

D. Extern monitors

Extern monitors are attached to extern calls. Their capabilities are restricted to what actions can do in P4 because of limitations the latter have on extern callers (e.g., it is not possible to make local declarations or invoke a table from inside an action). Similar to parser monitors, extern monitors can also be specialized to subgroups of a resource. In this case, a type signature is used to apply a monitor only to a subset of the extern calls. An example is presented in Figure 11-lines 20 to 24, where the extern monitor is applied only to calls for emitting headers of type ethernet_t. Extern monitors are useful to mediate how the data plane program interacts with the platform underlying it.

P4box instruments extern calls by adding *before* and *after* blocks right before and after every monitored call, respectively. The left part of Figure 8 illustrates this transformation, where the same extern call appears twice (inside an action and directly in the control block body). For the particular case in which a monitor has a type signature, only calls with that signature are instrumented. As an example, the right part of Figure 8 shows the instrumentation to the extern monitor specified in Figure 11.

```
control <control_name>
    ( <combined-params> ){
  [local_elements]
  [monitor_local_elements]

  apply{
    [before_statement]
    ...
    [block_statement]
    ...
    [after_statement]
  }
}
```

```
control pipeline(inout newHeaders hdr,
                 inout metadata meta){
  ipv4_t protec_ipv4;
  ...
  apply {
    protec_ipv4 = hdr.inner_ipv4;
    ...
    if(protec_ipv4 != hdr.inner_ipv4
     || protec_udp != hdr.inner_udp){
      ...
    }
  }
}
```

Fig. 6. Instrumentation of control blocks.

```
parser <parser_name>
     ( <combined-params> ){
  [local_elements]
  [monitor_local_elements]
  ...
  state <s_k-1> {
    transition [before_FSM];
  }
  [state before_FSM {
    transition <s_k> }]
  state <s_k> {
    transition [after_FSM];
  }
  [state after_FSM {
    transition <s_k+1> }]
  state <s_k+1> {
    transition <s_k+2>
  }
  ...
}
```

```
parser pipeline(packet_in packet,
                out newHeaders hdr){
  ...
  state parse_ethernet {
    transition _M_START_;
  }
  state _M_START_ {
    transition select(...){
      16w0xFFFF : parse_wp_header;
      ...
    }
  }
  state parse_wp_header {
    transition parse_ipv4;
  }
  state parse_ipv4 {
    transition parse_tcp;
  }
  ...
}
```

Fig. 7. Instrumentation of parsers.

```
control <control_name>
    ( <combined-params> ){
  action <action_name>(){
    ...
    [before_statement]
    [extern_A_call]
    [after_statement]
    ...
  }

  apply{
    ...
    [before_statement]
    [extern_A_call]
    [after_statement]
    ...
  }
}
```

```
control DeparserImpl(
          packet_out packet,
          in newHeaders hdr){
  apply{
    ...
    packet.emit(hdr.ethernet);
    packet.emit(hdr.wp_header);
    packet.emit(hdr.ipv4);
    ...
  }
}
```

Fig. 8. Instrumentation of extern calls.

E. Implementation

We implemented a prototype of P4box by extending the $P4_{16}$ reference compiler[4]. Our system has around 1.5K lines of C++ code and is publicly available[5]. We modified the front-end compiler to instrument programs by adding additional passes over their intermediate representation. Our examples and the workloads used in our experiments are also available online.

[4]https://github.com/p4lang/p4c

[5]https://github.com/mcnevesinf/p4box

IV. Enforcing Properties

The value of a mechanism like P4box is best seen through examples. In this section, we show how P4box can be used to enforce several kinds of properties in the data plane. Generally, these fall into two categories: program properties, which are properties of a single program's behavior, and network-wide properties, which are properties of several network devices' behavior.

A. Program Properties

Program properties concern the behavior of a program running on an individual device. These properties must hold regardless of how the device is configured or connected in a topology. They are also referred to as *network function properties* in the literature [10]. In this work, we consider two types of program properties: *generic safety* properties, which correspond to low-level properties related to the correct operation of a data plane program (e.g., packet formation properties and use-after-initialization), and *functional* or semantic properties, which guarantee the program conforms to a given user-specification (e.g., an RFC). Below we show how we enforce some program properties of interest, well-formedness and header protection.

1) Well-formedness: The output of a data plane program is *well-formed* if it complies with relevant protocol standards. *Well-formedness* determines the interoperability between multiple implementations of a protocol stack. In terms of programmable data planes, this means that the packets produced by one data plane program can be processed by another, and vice-versa. Enforcing well-formedness invariants is particularly useful in hybrid networks (i.e., networks containing both P4-enabled and legacy devices), where the elements may not support the same set of protocols. P4box can enforce well-formedness properties (e.g., packets do not contain both an IPv4 and IPv6 header, ICMP packets always have an IPv4 header) with simple checks of header validity at the end of the processing pipeline.

2) Header protection: In some cases, it may be desirable to ensure that a header is not modified by a forwarding device or programmable block. For example, in an deployment where VLANs are used to isolate potentially untrusted domains, it may be necessary to provide strong assurance that a VLAN tag is not modified by a forwarding device. P4box can be used to ensure that headers are not modified by collecting the appropriate packet state at the beginning of the processing pipeline (e.g., the value of a VLAN tag), and comparing it against the emitted headers. Such properties can be easily extended to allow only transformations to a pre-defined domain (e.g., source MAC can be modified only to a set of output interface addresses).

B. Network-Wide Properties

Network-wide properties concern forwarding devices when configured and connected in a particular topology [10]. These properties may involve basic predicates (e.g., A can reach B) as well as state and quantities (e.g., express desired behaviors

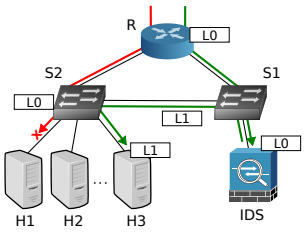


Fig. 9. Example topology for waypointing.

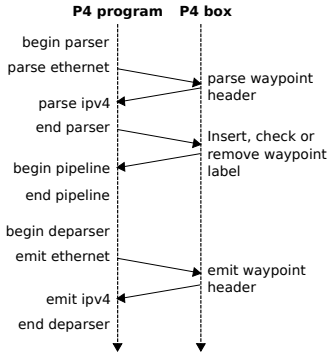


Fig. 10. Interaction between P4box and the P4 program to enforce waypointing.

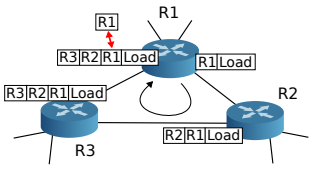


Fig. 12. Example topology for loop detection.

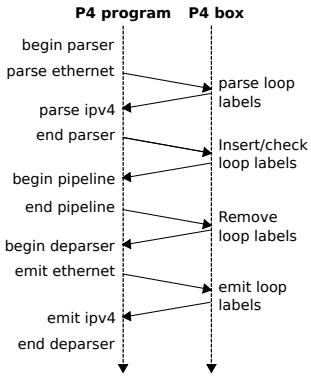


Fig. 13. Interaction between P4box and the P4 program to enforce loop detection.

```
1  struct p4boxState {
2    waypoint_t wp_header;
3  }
4
5  //Parser monitor to extract enforcement header
6  monitor wpParser(inout p4boxState pstate) on ParserImpl {
7    after parse_ethernet {
8      state start {
9        transition select(packet.lookahead<bit<32>>()){
10         16w0xFFFF : parse_wp_header;
11         default : accept;
12       }
13     }
14     state parse_wp_header {
15       packet.extract(pstate.wp_header);
16       transition accept;
17 }}}
18
19 //Extern monitor to emit enforcement header
20 monitor wpExtern(inout p4boxState pstate)
21                              on emit<ethernet_t>{
22   after {
23     packet.emit(pstate.wp_header);
24 }}
25
26 monitor wpControl(inout p4boxState pstate) on Pipeline {
27   ...
28   table check_waypoint {...}
29   ...
30
31   before {
32     //Enforce waypointing property
33     insert_label.apply();
34     check_waypoint.apply();
35     remove_label.apply();
36 }}
```

Fig. 11. Supervisor to enforce waypointing.

```
1  struct p4boxState {
2    ...
3    //Header stack to store sequence of labels
3    loop_header_t[10] loopHeader;
4  }
5
6  monitor loopMonitor(inout p4boxState pstate)
7                                       on Pipeline{
8    ...
9    action loop_detected(){ ... }
10   action insert_label( bit<32> label ){ ... }
11
12   /*Check if sequence of labels in a packet
13     contains router ID (i.e., has a loop)*/
14   table check_loop {
15     actions = { insert_label; loop_detected; }
16     key = {
17       pstate.loopHeader[0].label : ternary;
18       ...
19       pstate.loopHeader[9].label : ternary;
20     }
21     size = 10;
22   }
23
24   before {
25     check_loop.apply();
26   }
27 }
```

Fig. 14. Supervisor to detect forwarding loops.

for networks containing middleboxes or having latency constraints). We now describe how P4box can enforce common network-wide properties.

1) Waypointing: Network operators may want to force packets to pass through a sequence of devices (waypoints) before the network delivers them to an end host. P4box can enforce waypoint properties by checking and updating labels whenever these packets cross a device in the chain. As an example, Figure 9 shows a scenario where packets coming from an external network (i.e., through router R) must first be inspected by an IDS system before arriving at a web server (hosts H1–H3). In this case, a P4box monitor in R introduces labels in each packet in order to enforce waypointing. These labels are then updated by another monitor at switch S1, and a third monitor checks them at switch S2 for dropping packets that are destined to the web servers and do not contain the updated tag (L1). Figure 10 shows how P4box interacts with the P4 program to enforce waypointing, where vertical arrows represent the flow of execution. Note that P4box traps the program at three points: first, between the parsing of the Ethernet and IPv4 headers, to check whether the packet contains a label and extract the latter; second, right before the beginning of the match-action pipeline, to operate on the label (e.g., check, updates or remove) depending on how the device is connected in the topology; finally, to emit the label during

the deparsing phase.

Figure 11 shows a summary (with some parts omitted due to space constraints) of the code used to enforce waypoint properties. Each trap is programmed as a separate monitor. Parser (l.6-17) and extern (l.20-24) monitors are employed to extract and emit labels, which are declared in the wp_header (l.2). Moreover, a control block monitor uses match-action tables to insert, check/update and remove labels according to the incoming/outgoing ports of the packet. P4box monitors can be configured (proactive or reactively) to reroute packets on-the-fly and correct property violations. Moreover, we can extrapolate the labeling mechanism described above to enforce path conformance (i.e., to guarantee that the actual path taken by a packet conforms to the operator policy). In this case, P4box monitors check and update packet labels on every hop.

2) Loop detection: P4box can also detect forwarding loops by adding labels to packets. However, unlike waypointing, it appends a new label rather than updating a single one whenever the packet traverses a different hop. Figure 12 illustrates this idea, where labels contain router IDs. To detect a loop, a P4box monitor compares the sequence of labels already in the packet with the new one. If there is a match, then a loop is identified. Figure 13 shows the interaction between P4box and the P4 program in order to enforce loop detection. Similar to waypointing, P4box first hooks the program parser in order to extract the sequence of labels attached to the packet. However, two (rather than one) traps are needed during the match-action processing, one before and another after the pipeline. The former ensures the device does not waste time processing a packet that is in a loop and will be discarded anyway, while the latter is used to guarantee that the labels are only removed after a packet gets its output port in the last hop.

Figure 14 summarizes monitors for enforcing loop detection. Parser and extern monitors, which are used to extract and emit the sequence of labels, are omitted due to space constraints. Moreover, the sequence of labels is manipulated using a header stack (l.3). A control block monitor contains the match-action tables to check, insert and remove labels (l.6-27). Entries to these tables place the router ID in each position of the stack in order to detect a loop.

3) Traffic locality: Sometimes operators want to preserve traffic locality, e.g., packets flowing between two VMs in the same rack must not leave the top-of-rack switch in a data center, or traffic between two hosts in the same autonomous system should not leave its borders [7]. P4box can enforce traffic locality by controlling the set of output ports a packet can take. For example, packets from host A to B in Figure 15 are not allowed to be forwarded to upper ports. Figure 16 shows how P4box interacts with the P4 program to enforce traffic locality. First, it hooks the flow of execution at the beginning of the processing pipeline to save the state of required headers (e.g., MPLS or IPv4) before the program can modify them. Then, at the end of the pipeline, it uses the saved state as well as information about the outgoing port to check whether the packet can be forwarded. Figure 17 shows

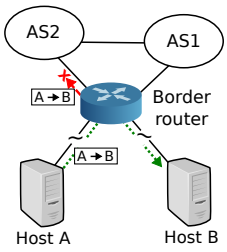


Fig. 15. Example topology for traffic locality.

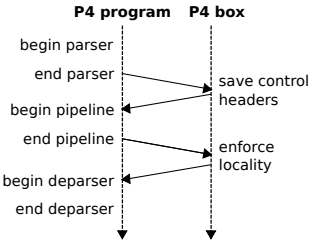


Fig. 16. Interaction between P4box and the P4 program to enforce traffic locality.

```
1   monitor tlMonitor(inout p4boxState pstate)
2                                    on Pipeline {
3     //Run enforcement action
4     action enforce_locality(){ ... }
5
6     //Check if packet violates locality
7     //(i.e., tries to leave AS)
8     table traffic_locality_table {
9       actions = { NoAction; enforce_locality; }
10      key = {
11        hdr.ipv4.srcAddr : ternary;
12        hdr.ipv4.dstAddr : ternary;
13        standard_metadata.egress_port : exact;
14      }
15      size = 512;
16    }
17
18    after { traffic_locality_table.apply(); }
19  }
```

Fig. 17. Supervisor to enforce traffic locality.

relevant parts of the monitor used to enforce traffic locality. It contains a single table that matches a set of control headers and the outgoing port (l.8-16), and runs an enforce_locality action (e.g., send the packet to a different outgoing port) when a violation is detected (l.4).

V. Performance

Because dynamic enforcement happens at run time, it may impose a performance penalty compared with static verification techniques. In this section, we analyze the performance overhead of P4box in terms of logical resources (i.e., tables, actions, headers) required for enforcing each property. We favor this kind of evaluation in a preliminary analysis because these metrics are target-independent and thus can be used to estimate the overhead for different types of network devices (e.g., hardware and software switches, SmartNICs and NetFPGAs). Moreover, they are not associated with any specific data plane program running on these devices, which could affect metrics such as latency and throughput. Overall, the higher the number of logical units in a P4 program, the higher the overhead in the data plane. For example, packet parsing latency increases with the number of headers (or bits) to be extracted, and a match-action stage takes longer to process a packet if we increase the number of tables or the complexity of the actions to be performed.

Table I summarizes the overhead of P4box for enforcing the properties described in Section IV. The column *key size*

TABLE I
P4BOX PERFORMANCE OVERHEAD. n = #CHECKS, m = #PROTECTED HEADERS, p = LABEL SIZE, q = #LABELS, s = LENGTH OF CONTROL FIELDS

Property	#Parsed bits	#Tables	Key size (bits)	#Field writes	#Lines of code
Well formedness (Sec. IV-A1)	0	0	0	1	$n+4$
Header protection (Sec. IV-A2)	0	0	0	m	$2m+12$
Waypointing (Sec. IV-B1)	p	3	$p+s$	5	80
Loop detection (Sec. IV-B2)	qp	3	qp	$4q$	$5q+80$
Traffic locality (Sec. IV-B3)	0	1	s	2	25
switch.p4 - IPv4	384	40	280	≈ 50	$\approx 6K$

reflects the size of the largest matching key when multiple tables are applied, and the column *field writes* corresponds to operations such as adding and removing headers as well as field assignments in actions. We use variables to indicate parameters that can be adjusted when enforcing each property. For example, header protection requires one field write for saving the state of each protected header (see lines 5-8 in Figure 5), in which case we represent the number of protected headers as m. This number may change from program to program. Other variables include the number of header validity checks for enforcing well-formedness, n, the size of the labels attached to packets for enforcing waypointing and loop detection, p, the maximum amount of labels, q, and the total length (in bits) of the fields used to control the operation of a monitor (e.g., IP addresses in traffic locality), s.

To put the numbers from Table I in perspective, we compare them with switch.p4[6], a widespread data plane program that implements a top-of-rack switch for data centers. Switch.p4 has more than 6K lines of code, and requires parsing 384 bits and applying 40 tables to process a traditional IPv4 packet. In order to enforce waypointing for example, P4box requires parsing only 8 bits (assuming $p = 8$) and applying 3 tables which are specified in 80 lines of code. In practice, this represents an increase lower than 5% in the packet processing latency according to the experiments we performed in a software switch[7]. Regarding resource consumption, if we consider hardware-based devices such as NetFPGAs, waypointing requires less than 3% additional memory blocks, flip-flops and lookup tables according to the literature [11] (assuming key sizes of 72 bits and a hash-based associative memory implementation).

In our ongoing work, we are investigating optimizations for enforcing each property (e.g., combining tables among them) in order to reduce even more these overheads. Moreover, P4box could benefit from parallelizations available in network devices to process monitors concurrently [12]. We plan to extend the evaluation for including measurements performed on high-performance P4-enabled devices (e.g., SmartNICs and NetFPGAs) as a future work.

VI. DISCUSSION

Monitor correctness. Although monitors can also contain bugs themselves, that is less likely to happen compared to the original program due to their intentionally small code base. Moreover, their simplicity makes them suitable to formal analysis (e.g., model checking or theorem proving) when security or reliability are important. In our ongoing work, we are exploring automatically converting monitors into an equivalent model in C and using an off-the-shelf symbolic execution engine (e.g., KLEE [13]) to prove their correctness.

High-level abstractions. While P4box allows programmers to use P4 for specifying properties, it is still necessary to think about each monitor individually. For example, programmers may need to create multiple monitors to enforce a network-wide property (e.g., a monitor for inserting and other for removing a label from packets). This can easily become a tedious process in large networks containing thousands of devices. Recent research efforts have proposed to automatically synthesize network configurations from higher-level abstractions (e.g., graphs or intents) [14]. We plan to extend P4box to support these abstractions in order to facilitate the enforcement of more complex properties or their combination.

VII. RELATED WORK

Network verification. Many tools have been proposed for verifying that a network behaves as expected. Moreover, these tools focus on either the control or the data plane. ERA [15] and Minesweeper [6] use models of networking protocols (e.g., BGP and OSPF) to analyze the network control plane. Although they can check multiple data plane configurations with this approach (i.e., the ones resulting from different protocol interactions), they are restricted to a limited number of protocols. Veriflow [16], NoD [7] and SymNet [17], on the other hand, are data plane verifiers. They take a single data plane configuration (i.e., set of forwarding rules) as input, and check whether certain properties hold for all possible packets. Data plane verification approaches are typically not tied to any specific protocol, but network programmers need to manually build a separate model for each data plane program, which may be a cumbersome and error prone task.

P4v [18] and ASSERT-P4 [5] can automatically verify P4 programs, but they are able to check only program-specific properties. Finally, Vera [4] and P4Nod [8] create models for data plane programs that can be used as input to SymNet and NoD, respectively. Although they can quickly verify small data plane programs (i.e., in the order of seconds), the verification time grows exponentially with both the program and the network size.

[6]https://github.com/p4lang/switch/

[7]https://github.com/p4lang/behavioral-model

Network debugging. Another dynamic approach to ensure security and correctness properties in networks is debugging. This approach is essentially based on monitoring and collecting statistics from network devices to perform an offline analysis. For example, Marple [19] proposes a query language for specifying monitoring tasks. Stroboscope [20] extends this idea and also considers scheduling to meet resource constraints. Instead of monitoring and collecting data, P4box processes information embedded on packets in switches at runtime. This design enables our mechanism to promptly react to property violations, containing them before they compromise a network policy. In-band Network Telemetry (INT)[8] provides flexibility similar to ours. However, it assumes information embedded on packets can not be compromised by buggy or malicious data plane programs. P4box, on the other hand, creates an isolated environment that can be used by network programmers to securely enforce policies of interest.

Runtime enforcement. The idea of using runtime monitors to enforce properties was first introduced by [21] in the context of system security more than forty years ago. In computer networks, FlowTags is a seminal work that proposed to extend middleboxes to add tags on packets which would be used by switches to enforce path conformance and origin binding [22]. However, unlike P4box, it does not take data plane programs and all possible bugs that come with them into account.

VIII. Conclusion

P4 and programmable data planes lowered the barrier for innovation in networking, but at the same time also made networks more prone to bugs and misconfigurations. To solve this problem we proposed P4box, a system for dynamically enforcing properties in programmable data planes through runtime monitors. P4box can enforce both program and network-wide properties while requiring a small effort from network programmers. Moreover, it represents a modest overhead to network devices in terms of latency and memory consumption. As future work, we plan to combine static verification and dynamic enforcement to build efficient, correct-by-construction programmable data planes.

Acknowledgments. This work has been supported by grants from NSF (CNS-1740911), RNP/CTIC (P4Sec), CNPq (140317/2017-1), and also by CAPES/Brazil – Finance Code 001.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

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

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

[8]https://p4.org/assets/INT-current-spec.pdf

[4] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging p4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018, pp. 518–532.

[5] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos, "Verification of p4 programs in feasible time using assertions," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2018, pp. 73–85.

[6] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the ACM SIGCOMM Conference*, 2017, pp. 155–168.

[7] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 499–512.

[8] N. Lopes, N. Bjorner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese, "Automatically verifying reachability and well-formedness in p4 networks," Tech. Rep., September 2016.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the European Conference on Object-Oriented Programming*, 1997, pp. 220–242.

[10] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A formally verified nat," in *Proceedings of the ACM SIGCOMM Conference*, 2017, pp. 141–154.

[11] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4fpga: A rapid prototyping framework for p4," in *Proceedings of the ACM Symposium on SDN Research (SOSR)*, 2017, pp. 122–135.

[12] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 103–115.

[13] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Conference on Operating Systems Design and Implementation (OSDI 08)*, 2008, pp. 209–224.

[14] A. Abhashkumar, J.-M. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu, "Supporting diverse dynamic intent-based policies using janus," in *Proceedings of the International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2017, pp. 296–309.

[15] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 217–232.

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

[17] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proceedings of the ACM SIGCOMM Conference*, 2016, pp. 314–327.

[18] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018, pp. 490–503.

[19] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, 2017, pp. 85–98.

[20] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever, "Stroboscope: Declarative network monitoring on a budget," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 467–482.

[21] J. P. Anderson, "Computer security technology planning study," Air Force Electronic Systems Division, Tech. Rep., 1972.

[22] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 543–546.

APPENDIX C — PAPER SUBMITTED TO IEEE/ACM TON

Title: Dynamic property enforcement in programmable data planes

Journal: IEEE/ACM Transactions on Networking (ToN)

Qualis: A1

Submission Date: March 18, 2020

Abstract: Network programmers can currently deploy an arbitrary set of protocols in forwarding devices through data plane programming languages such as P4. However, as any other type of software, P4 programs are subject to bugs and misconfigurations. Network verification tools have been proposed as a means of ensuring that the network behaves as expected, but these tools frequently face severe scalability issues. In this paper, we argue for a novel approach to this problem. Rather than statically inspecting a network configuration looking for bugs, we propose to enforce networking properties at runtime. To this end, we developed *P4box*, a system for deploying runtime monitors in programmable data planes. *P4box* allows programmers to easily express a broad range of properties (both program-specific and network-wide). Moreover, we provide an automated framework based on assertions and symbolic execution for ensuring monitor correctness. Our experiments on a SmartNIC show that *P4box* monitors represent a small overhead ($<$20%) to network devices in terms of latency, throughput and power consumption.

Dynamic Property Enforcement in Programmable Data Planes

Miguel Neves*, Bradley Huffaker†, Kirill Levchenko‡ and Marinho Barcellos§
UFRGS*, CAIDA/UCSD†, UIUC‡, University of Waikato§

***Abstract*—Network programmers can currently deploy an arbitrary set of protocols in forwarding devices through data plane programming languages such as P4. However, as any other type of software, P4 programs are subject to bugs and misconfigurations. Network verification tools have been proposed as a means of ensuring that the network behaves as expected, but these tools frequently face severe scalability issues. In this paper, we argue for a novel approach to this problem. Rather than statically inspecting a network configuration looking for bugs, we propose to enforce networking properties at runtime. To this end, we developed *P4box*, a system for deploying runtime monitors in programmable data planes. *P4box* allows programmers to easily express a broad range of properties (both program-specific and network-wide). Moreover, we provide an automated framework based on assertions and symbolic execution for ensuring monitor correctness. Our experiments on a SmartNIC show that *P4box* monitors represent a small overhead (<20%) to network devices in terms of latency, throughput and power consumption.**

I. Introduction

Programmable data planes allow network operators to modify the packet processing pipeline of network devices to quickly deploy new protocols, customize network behavior, and implement advanced network services. The introduction of the P4 [1] programming language has greatly lowered the barriers to doing so, bringing data plane programming into the mainstream. Over the last years, an ecosystem of data plane software has emerged (e.g., [2], [3]), and we can expect to see network devices running code written by teams of developers across multiple organizations, assembled by a network operator from libraries and modules, in the near future.

Despite the simplicity of its programming model, P4 programs have demonstrated to be prone to a variety of bugs and misconfigurations [4], [5]. As a result, network operators need ways to ensure that the programs they produce behave correctly in order to reap the benefits of a data plane software ecosystem. Decades of progress in software engineering have produced mature tools and methodologies for ensuring that certain properties hold in a program, and this idea has been gradually extended to the networking domain. State-of-the-art network verification tools can take a model of the network, its configuration, and a set of properties specified using traditional formalisms (e.g., temporal logic or Datalog rules) and automatically check whether these properties hold for any packet [6], [7].

Although these tools have helped network operators to identify bugs before they manifest, they still face important issues that hinder their adoption in production networks. First, most of these tools require programmers to manually model data plane programs, which is a cumbersome and error-prone task [7]. Second, these tools are usually restricted in terms of the properties they can guarantee. For example, some of them are specialized to the verification of reachability properties in order to reduce verification times [8]. Third, more expressive tools capable of verifying multiple properties frequently face severe scalability issues (e.g., checking conformance with a protocol specification can take days even for a single data plane program [4]). Finally, programmers usually have to be proficient in formal verification techniques for correctly specifying their properties.

In this paper, we propose a novel approach to this problem which is based on dynamic (or runtime) enforcement rather than static verification. While the former cannot always provide the kind of strong correctness guarantees that the latter can, it has several practical advantages. First, we do not need to wait for the outcome of a long verification process in order to push a new configuration out to the network switches. In addition, runtime enforcement can promptly intervene if problematic situations actually occur. It means we can still extract some useful work from buggy code when it behaves correctly, and perhaps repair problems without disturbing any network service (see an example in Section IV-B2).

In contrast to static verification, run-time enforcement also lets the developer express policy and mechanism using the same programming environment as the rest of the program. The value of this should not be underestimated: not only does it make life easier for the developer, it also prevents translation errors between implementation and policy domains. That is, rather than expressing a property, such as loop-free forwarding using a separate modeling or formal reasoning language, the programmer can write code to enforce and verify the desired properties in the language of the program (i.e., P4 in our case).

To realize the benefits of our dynamic enforcement approach we developed P4box, a system for deploying runtime monitors in programmable data planes. A *program monitor* is a language construct we developed (as an extension to P4) inspired by the *Aspect-Oriented Programming* (AOP) paradigm [9] which provides language-level constructs for attaching code to designated points in an existing program without modifying the program itself. Programmers can use monitors to modify or verify the behavior of control blocks, parsers, and external functions of P4 programs, and thus ensure they respect a set of desired properties. Monitors are particularly well-suited to the context in which data plane programs are assembled from externally-maintained modules, where it may be desirable

to alter or verify the behavior of these modules without modifying their code.

P4box instruments a P4 program with monitors at compile-time in such a way that the former cannot circumvent or interfere with the latter. Moreover, monitors can be combined to enforce more complex properties such as the ones involving extraction and emition of labels on packets (see an example in Section IV-B1). In summary, we make the following contributions:

- ❖ We design an extension to the P4 data plane programming language, called a *monitor*, that allows a programmer to specify properties about the network (using P4) in the form of pre- and post-conditions to control-blocks, parsers and extern functions (Section III).
- ❖ We develop P4box, a system for deploying runtime monitors in programmable data planes by instrumenting P4 programs at compile-time in such a way that the former cannot be hindered, tampered or circumvented (Section III).
- ❖ We show how P4box can be used to enforce several networking properties, including packet well-formedness, header protection, and waypointing (Section IV).
- ❖ We provide an automated framework based on assertions and symbolic execution for allowing programmers to check properties of interest on monitors (Section V).
- ❖ We evaluate P4box on various applications running in a SmartNIC and show that monitors impose low overhead (<20% in the worst case) to network devices in terms of latency, throughput and power consumption (Section VI).

This paper extends our earlier conference paper [10] by describing our automated framework for ensuring monitor correctness as well as the extensive set of experiments we performed on a commodity SmartNIC. Also, we have updated the related work to reflect the most recent advances we found in the literature. The remaining of this paper is organized as follows. Section II reviews the architecture of programmable network devices, summarize the main aspects of P4 programs, and motivates the development of property enforcement mechanisms in programmable data planes. Section VII compares our proposal with related work, and finally Section VIII concludes the paper.

II. Background and Motivation

A. Programmable network devices

Programmable network devices (a.k.a. *targets*) are packet processing elements (i.e., switches, SmartNICs, NetFPGAs) that allow network programmers to configure their data plane. These devices implement variations of an architecture known as PISA (Protocol Independent Switch Architecture)[1]. PISA-based devices contain multiple programmable blocks, which can be parsers, deparsers, match-action stages or queueing systems. Figure 1 presents an example of a PISA-based switch containing three programmable blocks (dashed boxes): a parser, a match-action pipeline and a deparser. Each programmable block can be configured by developers using a data plane programming language (typically P4), and the organization and capabilities of these blocks are abstracted to P4 programs as an interface or *architecture model*.

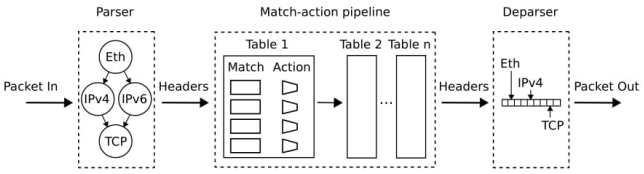


Fig. 1. Example of PISA-based switch. Dashed blocks can be programmed in P4.

```
1  parser ParserImpl( packet_in packet ){...}
2
3  control Pipeline( inout headers hdr ){
4    ...
5    action route( bit<9> iface ){ ... }
6
7    /* Route IPv4 packets */
8    table route_packet {
9      actions = { route; }
10     key = {
11       hdr.ipv4.srcAddr : ternary;
12       hdr.ipv4.dstAddr : ternary;
13     }
14     size = 1024;
15   }
16
17   apply{ route_packet.apply(); }
18 }
19
20 control DeparserImpl( packet_out packet ){...}
21
22 Switch(ParserImpl(), Pipeline(), DeparserImpl())
```

Fig. 2. Example P4 program

B. P4 Programs

As a domain specific language, P4 offers many constructs to facilitate the specification of packet processing tasks. Programmers can, for example, declare packet headers, parsers, tables, actions to modify packets, and control blocks to compose sequences of tables. These abstractions are used to configure different programmable blocks in network devices, and the configuration of all blocks comprises a P4 program. Figure 2 shows an example of a program for configuring the PISA-based switch described in Section II-A. In this example, the match-action pipeline block implements a single table that routes packets based on their IPv4 addresses (l.8-15).

C. Data Plane Bugs

Although the simplicity of its programming model (e.g., P4 programs have no loops or dynamic memory allocation [1]), data plane programs have demonstrated to be prone to many bugs and misconfigurations. Bugs in P4 vary in nature, but overall they can be both generic bugs (i.e. well-known from other programming languages) such as information overwriting[2] and data use-before-initialization[3], and also network specific bugs such as the creation of malformed packets [8], incorrect implementation of protocol specifications [5]

[1]https://p4.org/assets/p4-ws-2017-p4-architectures.pdf

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

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

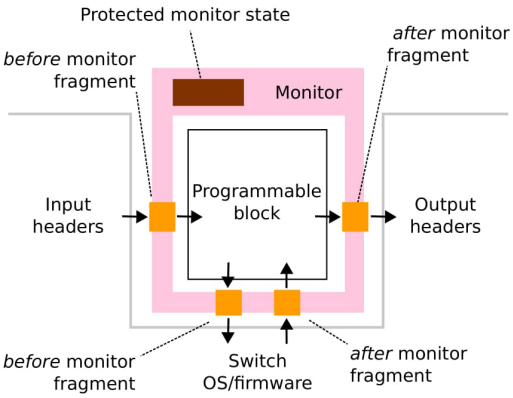


Fig. 3. P4box programming model.

or policy violations due to bad table configurations. In this context, it is essential to develop mechanisms that support the development of secure and correct network data planes.

III. P4box

P4box is a system that allows network programmers to deploy runtime *monitors* in programmable data planes. Using P4box programmers can attach monitors before and after control blocks, parser state transitions, and calls to external functions of a P4 program. Each monitor can modify the input and output of the code block or function it monitors. This enables the verification of pre- and post-conditions which can be used to enforce specific properties or modify the behavior of the monitored block. P4box inclines monitor code into the monitored P4 program at the intermediate representation level (i.e., during the compilation of the latter). The resulting program (original code plus monitors) then continues the compilation as before, which allows P4box to be used with any backend compiler based on the $\mathrm{P4}_{16}$ reference implementation. In the rest of this section, we provide an overview of P4box and its runtime monitors (Section III-A), describe the three kinds of monitors P4box can deploy in detail (Sections III-B, III-C, and III-D) and present our prototype implementation (Section III-E).

A. Overview

A runtime monitor interposes on the interaction of a P4 control block or parser with the rest of the execution environment (Figure 3), allowing the monitor programmer to modify the behavior of the enclosed P4 block with the rest of the environment. A P4 programmable block (either a control block or parser) interfaces with the rest of the P4 execution environment at entry into the block, return from the block, and at calls to architecture-supplied external functions. In the P4box programming model, when a programmable block is invoked, control first passes to a monitor, also written in P4, before passing to the intended programmable block. Similarly, when a programmable block completes processing, control first passes to the monitor before returning to the device. This allows a monitor to modify the behavior of programmable blocks in a well-defined way.

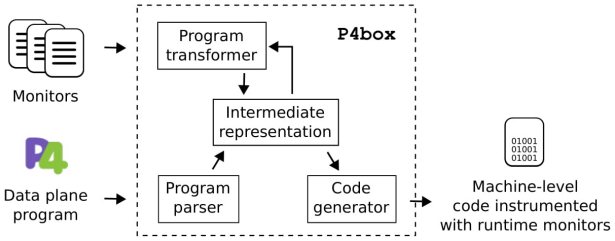


Fig. 4. P4box workflow.

Monitors can also interpose on calls to external functions: when a programmable block invokes an external function, control first passes to the monitor, then the function, and then back to the monitor again, before returning to the programmable block. A monitor can thus modify the apparent behavior of a external function. Monitors are declared and defined at the top level of a P4 program, alongside control blocks, parser blocks, and other top-level declarations. The syntax for a monitor is:

```
monitor <name> ( [param-list] ) on <object> {
    [local-declarations]
    (before | after) { <p4-statements> }
}
```

Each monitor is identified by a unique *<name>* and may receive additional parameters (*<param-list>*) containing headers and metadata in addition to the parameters of the monitored object. Every monitor must be associated with a data plane *<object>*, which can be a parser, control block or extern function. The resource type defines the set of *<p4-statements>* elements the monitor supports. Monitors can have two types of methods, namely: *before* and *after*, which specify code fragments that are executed before and after the monitored resource, respectively. Finally, they can also contain local declarations (e.g., actions, tables) visible inside the monitor but not the monitored block.

Figure 4 shows the P4box workflow. The original P4 program and P4 source files defining runtime monitors are provided to P4box which combines the original program with the monitors at the intermediate level to produce a new program suitable for further compilation. At the end, machine-level code containing all monitors is generated for a variety of targets. During the instrumentation process, P4box takes advantage of language features provided by P4 such as separate scopes and namespaces in addition to static analysis to provide the following guarantees for each monitor:

- **Complete mediation:** The flow of execution of the original data plane program will always pass through a monitor (when one is defined by the programmer). This means it is not possible for the original program to circumvent a monitor.
- **Non-interference:** The original program cannot interfere in the operation of a monitor (e.g., by modifying its local variables or headers), which means monitors are completely isolated from the data plane program.

Together, the complete mediation and non-interference

properties allow monitors to restrict what the original P4 program is allowed to do even when the latter is *untrusted* (e.g., a third-party program). Monitors are thus not only an aspect-oriented P4 program structuring mechanism, but also a software sandbox that can be used to encapsulate untrusted or buggy P4 code. Next, we show examples and describe each of the three kinds of monitors P4box supports in more detail.

B. Control block monitors

P4box can attach monitors to top-level control blocks. In this case, *before* and *after* contain statements that will be executed at the beginning and the end of block, respectively. Figure 5 shows an example of a control block monitor, which could be used to detect and process information overwriting bugs[2]. This monitor is responsible for ensuring that a header is not erroneously modified by the data plane program. The monitor is attached to the processing pipeline and has two elements: i) before the programmable block, it collects state from the original packet as soon as it is parsed (l.5-8); and ii) after the block, it tests whether monitored headers were modified (l.10-17). Local variables (i.e., visible only to the monitor) are used to store protected headers (l.2-3). If the monitor detects a violation, different actions can be performed to enforce the desired property (e.g., restore the original header value, notify the network controler, log an event), being up to the programmer to decide what to do.

P4box performs the instrumentation of control blocks in three steps: first, monitor parameters containing headers and metadata are merged with parameters of the monitored block (e.g., joining the fields of two structs to create a super struct). If during this process P4box identifies there is no feasible mapping (e.g., because there is no parameter in the monitored block that supports the merge operation), a message is emitted and the instrumentation process is aborted; second, *before* and *after* blocks as well as local declarations are inserted in the monitored block; finally, a name resolution pass maps monitor names to their new namespaces. The left part of Figure 6 illustrates this transformation, where a generic control block is instrumented with its monitoring primitives. A corresponding example is shown on the right, representing the instrumentation performed to the monitor specified in Figure 5. As a result of this transformation, all packets crossing the control block also pass through the monitor since P4 assumes network devices execute statements in order.

C. Parser monitors

Parser monitors, on their turn, can be attached to top-level parsers. As such, *before* and *after* can contain finite state machines and both of them must have a start and accept state. It is possible to specialize a parser monitor to a specific parser state, in which case *before* and *after* are associated only to the latter. An example of a parser monitor is shown in Figure 11-lines 6 to 17, where the monitor is attached to the parse_ethernet state and used to extract an enforcement header. Parser monitors are also particularly useful for skipping the extraction of packet bits that for some reason (e.g., confidentiality) should not be visible to the data plane program.

```
1  monitor hdrInvMonitor() on Pipeline {
2    ipv4_t protec_ipv4;
3    udp_t  protec_udp;
4
5    before {
6      protec_ipv4 = hdr.inner_ipv4;
7      protec_udp = hdr.inner_udp;
8    }
9
10   after {
11     if( protec_ipv4 != hdr.inner_ipv4 ||
12         protec_udp != hdr.inner_udp ){
13       /*Run enforcement action
14         (e.g., restore original header
15         value, notify the control plane,
16         write log) */
17   }}
15 }
```

Fig. 5. Example of control block monitor to enforce header protection.

To instrument parsers, P4box takes into account if *before* and *after* are attached to states or not. If not, it assumes the start and end (i.e., accept) states of the monitored parser as its hooking points. The left part of Figure 7 shows the transformations P4box applies. Assuming state S_k is being monitored, P4box links the finite state machine specified inside *before* (before_FSM) between states S_{k-1} and S_k by modifying state transitions. An analogous process is performed for the finite state machine specified inside *after* (after_FSM), linking it between states S_k and S_{k+1}. The right part of Figure 7, on its turn, shows an example of these transformations, where P4box performs the instrumentation to the parser monitor specified in Figure 11. Instead of transitioning directly from state parse_ethernet to parse_ipv4, the execution flow goes through states _M_START_ and parse_wp_header.

D. Extern monitors

Extern monitors are attached to extern calls. Their capabilities are restricted to what actions can do in P4 because of limitations the latter have on extern callers (e.g., it is not possible to make local declarations or invoke a table from inside an action). Similar to parser monitors, extern monitors can also be specialized to subgroups of a resource. In this case, a type signature is used to apply a monitor only to a subset of the extern calls. An example is presented in Figure 11-lines 20 to 24, where the extern monitor is applied only to calls for emitting headers of type ethernet_t. Extern monitors are useful to mediate how the data plane program interacts with the platform underlying it.

P4box instruments extern calls by adding *before* and *after* blocks right before and after every monitored call, respectively. The left part of Figure 8 illustrates this transformation, where the same extern call appears twice (inside an action and directly in the control block body). For the particular case in which a monitor has a type signature, only calls with that signature are instrumented. As an example, the right part of Figure 8 shows the instrumentation to the extern monitor specified in Figure 11.

```
control <control_name>
    ( <combined-params> ){
  [local_elements]
  [monitor_local_elements]

  apply{
    [before_statement]
    ...
    [block_statement]
    ...
    [after_statement]
  }
}
```

```
control pipeline(inout newHeaders hdr,
                 inout metadata meta){
  ipv4_t protec_ipv4;
  ...
  apply {
    protec_ipv4 = hdr.inner_ipv4;
    ...
    if(protec_ipv4 != hdr.inner_ipv4
     || protec_udp != hdr.inner_udp){
      ...
    }
  }
}
```

Fig. 6. Instrumentation of control blocks.

```
parser <parser_name>
     ( <combined-params> ){
  [local_elements]
  [monitor_local_elements]
  ...
  state <s_k-1> {
    transition [before_FSM];
  }
  [state before_FSM {
    transition <s_k> }]
  state <s_k> {
    transition [after_FSM];
  }
  [state after_FSM {
    transition <s_k+1> }]
  state <s_k+1> {
    transition <s_k+2>
  }
  ...
}
```

```
parser pipeline(packet_in packet,
                out newHeaders hdr){
  ...
  state parse_ethernet {
    transition _M_START_;
  }
  state _M_START_ {
    transition select(...){
      16w0xFFFF : parse_wp_header;
      ...
    }
  }
  state parse_wp_header {
    transition parse_ipv4;
  }
  state parse_ipv4 {
    transition parse_tcp;
  }
  ...
}
```

Fig. 7. Instrumentation of parsers.

```
control <control_name>
    ( <combined-params> ){
  action <action_name>(){
    ...
    [before_statement]
    [extern_A_call]
    [after_statement]
    ...
  }

  apply{
    ...
    [before_statement]
    [extern_A_call]
    [after_statement]
    ...
  }
}
```

```
control DeparserImpl(
          packet_out packet,
          in newHeaders hdr){
  apply{
    ...
    packet.emit(hdr.ethernet);
    packet.emit(hdr.wp_header);
    packet.emit(hdr.ipv4);
    ...
  }
}
```

Fig. 8. Instrumentation of extern calls.

E. Implementation

We implemented a prototype of P4box by extending the $P4_{16}$ reference compiler[4]. Our system has around 1.5K lines of C++ code and is publicly available[5]. We modified the front-end compiler to instrument programs by adding additional passes over their intermediate representation. Our examples and the workloads used in our experiments are also available online.

IV. Enforcing Properties

The value of a mechanism like P4box is best seen through examples. In this section, we show how P4box can be used to enforce several kinds of properties in the data plane. Generally, these fall into two categories: program properties, which are properties of a single program's behavior, and network-wide properties, which are properties of several network devices' behavior.

A. Program Properties

Program properties concern the behavior of a program running on an individual device. These properties must hold regardless of how the device is configured or connected in a topology. They are also referred to as *network function properties* in the literature [11]. In this work, we consider two types of program properties: *generic safety* properties, which correspond to low-level properties related to the correct operation of a data plane program (e.g., packet formation properties and use-after-initialization), and *functional* or semantic properties, which guarantee the program conforms to a given user-specification (e.g., an RFC). Below we show how we enforce some program properties of interest, well-formedness and header protection.

1) Well-formedness: The output of a data plane program is *well-formed* if it complies with relevant protocol standards. *Well-formedness* determines the interoperability between multiple implementations of a protocol stack. In terms of programmable data planes, this means that the packets produced by one data plane program can be processed by another, and vice-versa. Enforcing well-formedness invariants is particularly useful in hybrid networks (i.e., networks containing both P4-enabled and legacy devices), where the elements may not support the same set of protocols. P4box can enforce well-formedness properties (e.g., packets do not contain both an IPv4 and IPv6 header, ICMP packets always have an IPv4 header) with simple checks of header validity at the end of the processing pipeline.

2) Header protection: In some cases, it may be desirable to ensure that a header is not modified by a forwarding device or programmable block. For example, in an deployment where VLANs are used to isolate potentially untrusted domains, it may be necessary to provide strong assurance that a VLAN tag is not modified by a forwarding device. P4box can be used to ensure that headers are not modified by collecting the appropriate packet state at the beginning of the processing pipeline (e.g., the value of a VLAN tag), and comparing it against the emitted headers. Such properties can be easily extended to allow only transformations to a pre-defined domain (e.g., source MAC can be modified only to a set of output interface addresses).

B. Network-Wide Properties

Network-wide properties concern forwarding devices when configured and connected in a particular topology [11]. These properties may involve basic predicates (e.g., A can reach B) as well as state and quantities (e.g., express desired behaviors for networks containing middleboxes or having latency constraints). We now describe how P4box can enforce common network-wide properties.

[4]https://github.com/p4lang/p4c
[5]https://github.com/mcnevesinf/p4box

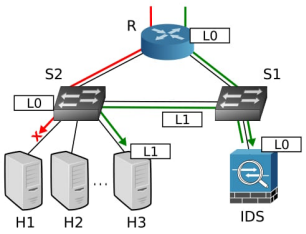


Fig. 9. Example topology for waypointing.

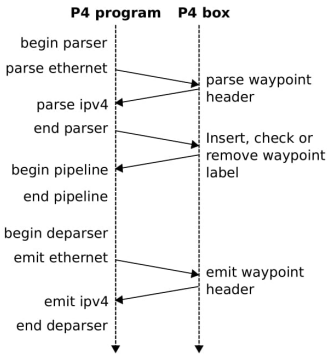


Fig. 10. Interaction between P4box and the P4 program to enforce waypointing.

```
1  struct p4boxState {
2    waypoint_t wp_header;
3  }
4
5  //Parser monitor to extract enforcement header
6  monitor wpParser(inout p4boxState pstate) on ParserImpl {
7    after parse_ethernet {
8      state start {
9        transition select(packet.lookahead<bit<32>>()){
10         16w0xFFFF : parse_wp_header;
11         default : accept;
12       }
13     }
14     state parse_wp_header {
15       packet.extract(pstate.wp_header);
16       transition accept;
17 }}}
18
19 //Extern monitor to emit enforcement header
20 monitor wpExtern(inout p4boxState pstate)
21                            on emit<ethernet_t>{
22   after {
23     packet.emit(pstate.wp_header);
24 }}
25
26 monitor wpControl(inout p4boxState pstate) on Pipeline {
27   ...
28   table check_waypoint {...}
29   ...
30
31   before {
32     //Enforce waypointing property
33     insert_label.apply();
34     check_waypoint.apply();
35     remove_label.apply();
36 }}
```

Fig. 11. Supervisor to enforce waypointing.

1) Waypointing: Network operators may want to force packets to pass through a sequence of devices (waypoints) before the network delivers them to an end host. P4box can enforce waypoint properties by checking and updating labels whenever these packets cross a device in the chain. As an example, Figure 9 shows a scenario where packets coming from an external network (i.e., through router R) must first be inspected by an IDS system before arriving at a web server (hosts H1–H3). In this case, a P4box monitor in R introduces labels in each packet in order to enforce waypointing. These labels are then updated by another monitor at switch S1, and a third monitor checks them at switch S2 for dropping packets that are destined to the web servers and do not contain the updated tag (L1). Figure 10 shows how P4box interacts with the P4 program to enforce waypointing, where vertical arrows represent the flow of execution. Note that P4box traps the program at three points: first, between the parsing of the Ethernet and IPv4 headers, to check whether the packet contains a label and extract the latter; second, right before the beginning of the match-action pipeline, to operate on the label (e.g., check, updates or remove) depending on how the device is connected in the topology; finally, to emit the label during the deparsing phase.

Figure 11 shows a summary (with some parts omitted due to space constraints) of the code used to enforce waypoint properties. Each trap is programmed as a separate monitor. Parser (l.6-17) and extern (l.20-24) monitors are employed to extract and emit labels, which are declared in the wp_header (l.2). Moreover, a control block monitor uses match-action tables to insert, check/update and remove labels according to the incoming/outgoing ports of the packet. P4box monitors can be configured (proactive or reactively) to reroute packets on-the-fly and correct property violations. Moreover, we can extrapolate the labeling mechanism described above to enforce path conformance (i.e., to guarantee that the actual path taken by a packet conforms to the operator policy). In this case, P4box monitors check and update packet labels on every hop.

2) Traffic locality: Sometimes operators want to preserve traffic locality, e.g., packets flowing between two VMs in the same rack must not leave the top-of-rack switch in a data center, or traffic between two hosts in the same autonomous system should not leave its borders [7]. P4box can enforce traffic locality by controlling the set of output ports a packet can take. For example, packets from host A to B in Figure 12 are not allowed to be forwarded to upper ports. Figure 13 shows how P4box interacts with the P4 program to enforce traffic locality. First, it hooks the flow of execution at the beginning of the processing pipeline to save the state of required headers (e.g., MPLS or IPv4) before the program can modify them. Then, at the end of the pipeline, it uses the saved state as well as information about the outgoing port to check whether the packet can be forwarded. Figure 14 shows relevant parts of the monitor used to enforce traffic locality. It contains a single table that matches a set of control headers and the outgoing port (l.8-16), and runs an enforce_locality action (e.g., send the packet to a different outgoing port) when a violation is detected (l.4).

V. Checking Monitor Correctness

Monitors are less likely to contain bugs compared to P4 programs due to their smaller size. For example, a monitor to enforce header protection has no more than a dozen of lines of code while traditional P4 programs usually have hundreds to thousands of lines (two to three orders of magnitude larger) [4], [12]. Despite their simplicity, monitors are still subject to bugs and misconfigurations though. For this reason, we developed an automated framework for allowing programmers to check invariants in their specified monitors.

Our framework is inspired in assert-p4 [5], a state-of-the-art tool for checking invariants in P4 programs. As for

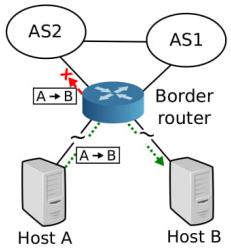


Fig. 12. Example topology for traffic locality.

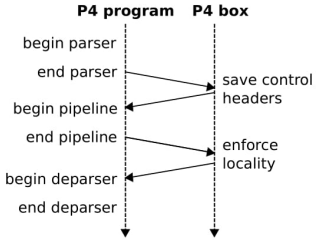


Fig. 13. Interaction between P4box and the P4 program to enforce traffic locality.

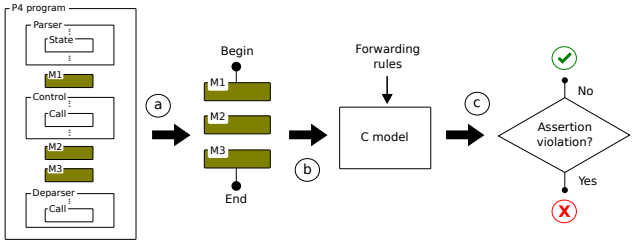


Fig. 15. Workflow for checking monitor correctness. M1, M2, M3 = annotated monitors. a = monitor assembling. b = model extraction. c = symbolic execution.

```
1   monitor tlMonitor(inout p4boxState pstate)
2                                   on Pipeline {
3     //Run enforcement action
4     action enforce_locality(){ ... }
5
6     //Check if packet violates locality
7     //(i.e., tries to leave AS)
8     table traffic_locality_table {
9       actions = { NoAction; enforce_locality; }
10      key = {
11        hdr.ipv4.srcAddr : ternary;
12        hdr.ipv4.dstAddr : ternary;
13        standard_metadata.egress_port : exact;
14      }
15      size = 512;
16    }
17
18    after { traffic_locality_table.apply(); }
19  }
```

Fig. 14. Supervisor to enforce traffic locality.

assert-p4, our framework is also based on assertions and symbolic execution (see Figure 15 for its workflow). First, programmers annotate monitors with assertions expressing properties of interest. For that, we consider the same assertion language as proposed in [5], which is also a good fit to our problem since monitors are comprised of P4 constructs. The language includes elements for specifying logical, relational and arithmetic expressions, as well as conditional statements and basic tests involving packet headers (e.g., whether a header was extracted from a packet or not).

Once annotated, monitors are assembled in a "virtual program" respecting the same order of execution as the monitored code. This means if monitors A and B are monitoring programmable blocks X and Y, respectively, and X runs before Y, then A will precede B. In addition, the assembled code also contains all header and metadata definitions from the original program, which are treated as symbolic inputs by the verification engine and enable programmers to check invariants on monitors that manipulate program state (e.g., change a header value). After the assembling phase, the new virtual program is translated into an equivalent model in C, and assertions are checked using a symbolic execution tool.

Translating monitors to C allows us to use an off-the-shelf symbolic execution engine, e.g., KLEE [13], to check the desired properties. Moreover, tools to ensure the correctness of the translation process are also available[6]. As an example, Figure 16 shows the resulting model for the monitor described in Section III-B (we omit some parts for the sake

[6]https://github.com/gnmartins/assert-p4

```
1  #include "klee.h"
2
3  //Model monitor locals
4  ipv4_t protec_ipv4;
5  udp_t protec_udp;
6
7  //Make monitor inputs symbolic
8  void symbolizeInputs(){
9    klee_make_symbolic(&hdr, sizeof(hdr), "hdr");
10   klee_make_symbolic(&meta, sizeof(meta), "meta");
11 }
12
13 //Model monitor logic
14 void hdrInvMonitor_before(){
15   protec_ipv4 = hdr.inner_ipv4;
16   constant_protec_var = protec_ipv4;
17   protec_udp = hdr.inner_udp;
18 }
19
20 void hdrInvMonitor_after(){
21   if( protec_ipv4 != hdr.inner_ipv4 ||
22       protec_udp != hdr.inner_udp ){ ... }
23 }
24
25 int main(){
26   symbolizeInputs();
27   hdrInvMonitor_before();
28   hdrInvMonitor_after();
29   //Model assertions
30   hasChanged( constant_protec_var, protec_ipv4 );
31   return 0;
32 }
```

Fig. 16. Equivalent model in C to the monitor described in Section III-B.

of simplicity). The *main* code (lines 25-32) controls the call order for the monitors, which are on their turn modeled as additional functions (lines 14-23). We make all monitor inputs (i.e., packet headers, metadata and protected state) symbolic (lines 8-11), so that they can be comprehensively checked by the symbolic execution engine. Local monitor definitions (e.g. variables and match-action tables) are modeled as unique global constructs (lines 4-5). Finally, each assertion is modeled independently, and usually involves variables that are set and tested at relevant points in the program. For example, the assertion modeled in lines 16 and 30 checks whether the monitor, which should ensure a packet header is not modified, is not itself erroneously modifying the header. We refer to [5] for more details on the translation process.

Performance. One of the key concerns in automated testing is performance as not rarely the cost of checking a program invariant becomes prohibitive in practice. For example, symbolic execution is particularly known for its path explosion issue [14] and other techniques also have their own drawbacks (e.g., the state space explosion problem in model checking [15] or

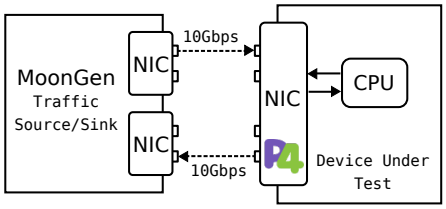


Fig. 17. Testbed topology. Dashed arrows represent the data flow. Solid arrows indicate control traffic (e.g., for programming the NIC firmware using P4 and collecting statistics).

Property	Latency (us)		
	Avg	5th	95th
Well formedness	1.91	1.24	3.61
Header protection	1.32	1.02	2.30
Traffic locality	1.25	1.02	1.80
Waypointing	0.97	0.87	1.40
All 4 properties	2.35	1.74	3.12

TABLE I
AVERAGE, 5TH AND 95TH-PERCENTILE LATENCY COST OF THE PROPERTIES DESCRIBED IN SECTION IV.

large logical formulas in SMT solving [12]). A few techniques (e.g., program slicing and directed symbolic execution) have been proposed to reduce this burden in the context of P4 programs, but it still takes hours or even days to check a relatively complex program instance [4], [5].

To demonstrate the scalability of checking monitor invariants using our approach, we applied our framework to check basic semantic properties (i.e., show that monitors in fact meet their specification) on the monitors described in Section IV. We run our experiments in a single-core virtual machine equipped with 4GB of RAM and Ubuntu 18.04. We used KLEE 2.0, the Z3 solver, and LLVM 6.0 as the symbolic execution engine. In each case, our framework was able to check the whole input space in less than a second. This is mainly because of the small size of monitors, which typically result in no more than a few hundred execution paths.

VI. EVALUATION

Because dynamic enforcement happens at run time, it may impose a performance penalty compared with static verification. In this section, we analyze the performance overhead of P4box and show it is small for many useful properties and applications.

Figure 17 shows the topology of the setup for evaluating P4box. The device under test (DuT) is equipped with a 4-core Intel Core i3 530 2.93GHz CPU and a single-port 40G Agilio CX smart NIC running in breakout mode (i.e., 4x10G virtual interfaces). The traffic generator, on its turn, contains a 4-core Intel Xeon E31220 3.1GHz CPU and two dual-port 10G Agilio CX NICs. We configure the traffic generator with MoonGen [16] and use a single interface in each NIC for sending and receiving traffic respectively, leaving the other interfaces unused. Unless explicitly mentioned otherwise, our analyses consider the traffic generator creates a 10 Gbps stream of 64-byte UDP packets (∼14.8 million packets per second).

All P4 programs run as embedded firmware in the DuT NIC and are isolated from other end host resources (e.g., CPU, memory and operating system). We use P4box to create instrumented P4 programs and then the Netronome P4 compiler with MAC timestamps and shared content stores enabled to convert instrumented programs into target specific code. Except for Section VI-A, in which we analyze the cost of enforcing each property separately, all our experiments assume P4box instruments data plane programs with the four properties described in Section IV, so that we could measure overheads in more demanding conditions.

We measure throughput, latency and power consumption to compare the forwarding performance of the device under test with and without P4box. To measure throughput, we count the number of packets processed in the NIC each second using a P4 counter. We report the average of 10 runs where each run lasts for 30 seconds. To measure the packet processing latency, we collect NIC ingress/egress timestamps and report results over 100 packets. Finally, we use the automated script provided by Netronome (*nic-power*) to read the board power consumption every 100 milliseconds, and similarly to latency measurements also report results over 100 reads. All measurements are performed after a 5 seconds warm-up interval.

A. Property overhead

We start looking at the overhead of each property in isolation. To evaluate this overhead, we instrumented a very simple data plane program (L3 routing – see Table II) with P4box configured to enforce a single property, and measure the performance drawback compared to a baseline (i.e., the same program without any instrumentation). Table I shows the latency overhead, in microseconds, for enforcing the properties described in Section IV. As we can see, the overhead is under 5 μs even when we consider all properties together – last line in the table. This is at least one order of magnitude smaller than the latency cost for processing a packet in many data plane applications (see Section VI-B). Also, the overhead is clearly not additive, meaning the cost for enforcing a combination of properties is not the same as the sum of the cost for enforcing the individual ones. This is because P4box can employ resource sharing among monitors in order to optimize their performance.

B. Application performance

Next, we evaluate the forwarding performance of the device under test while running real-world applications instrumented with P4box. We select instances of 4 popular applications across different domains: (1) L3 routing, which forwards packets based on destination IP addresses [17]; (2) Load balancing, which uses Othello hashes for mapping virtual IPs (VIPs) to destination servers (DIPs) [18]; (3) DDoS detection, which adopts counting sketches to identify malicious flows [19]; and (4) Surveillance protection, which encrypts IP addresses to obfuscate information about Internet users and devices [20]. Table II summarizes the P4 programs implementing these applications. Each program has a distinct number of matching tables, which results in different pipeline depths. Moreover,

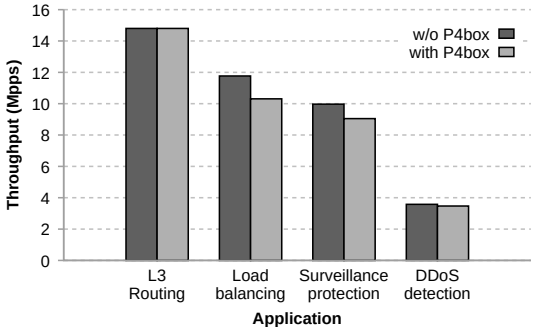


Fig. 18. Average throughput for the evaluated applications. Standard deviation is less than 0.1 Mpps.

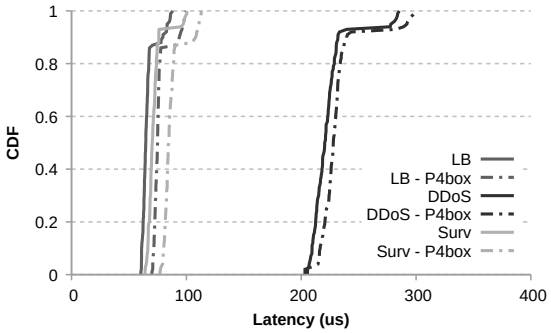


Fig. 19. CDF of the packet latency for the evaluated applications.

three of the programs do not manipulate any persistent state in the device while the remaining one uses registers for storing packet counts.

Figure 18 compares the throughput of the device under test for the evaluated applications. In all cases, the overhead for running P4box is small, representing a throughput drop of about 9% (1.4 Mpps) for load balancing, 6% (0.9 Mpps) for surveillance protection and 0.7% (0.1 Mpps) for DDoS detection. Interestingly, there was no noticeable overhead for L3 routing as this application was able to achieve line rate in both scenarios.

Figure 19 compares the cumulative distribution of the packet processing latency for the different applications. As can be seen, P4box implies a small latency overhead for packets. For example, the increase in the median latency is below 20% in all cases (4% for DDoS detection, 15% for load balancing and 19% for surveillance protection). Results are similar when we look at the tail latencies, with an overhead smaller than 15% at the 99th percentile in the worst case (for load balancing). Overall, the more complex the application the lower the penalty for running P4box.

C. Effect of packet rate

We now turn our attention to examining how different packet rates affect P4box. We consider a maximum load scenario in which the traffic generator sends traffic at the constant rate of 10Gbps, but changes the packet size and consequently the number of packets sent per unit of time. For example, the traffic generator can send up to 14.8 million 64-byte packets

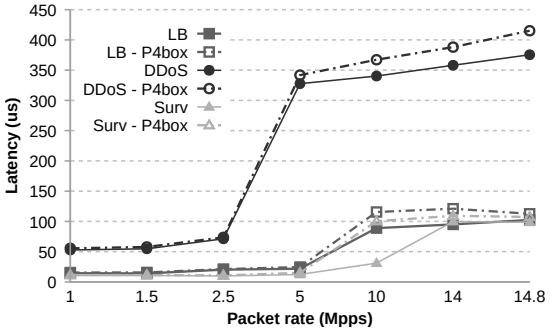


Fig. 20. 95-percentile tail latencies at different packet rates.

Application	#Tables	Stateful	LoC
L3 routing [17]	3	N	160
Load balancing [18]	11	N	420
Surveillance protection [20]	6	N	480
DDoS detection [19]	2	Y	540

TABLE II
EVALUATED APPLICATIONS. LOC = LINES OF CODE.

per second, but this number reduces to approximately 1 million if it instead sends packets of 1500 bytes.

Figure 20 compares the 95-percentile tail latency for different applications as a function of the packet rate. P4box overhead is negligible up to 5 Mpps. This is because NIC resources are not overloaded at low rates. Above 5 Mpps, P4box increases tail latencies around 20% as a result of bottleneck on NIC. This bottleneck is more prominent in computing-intensive applications such as DDoS detection, where higher processing demands per packet induce a head-of-line (HOL) blocking and consequently queueing formation at input ports [21].

D. Power consumption

Finally, we evaluate how P4box affects the SmartNIC power consumption. First, we measure the overhead for different link utilizations. We start with an idle system, and gradually increase the input rate until it achieves full link capacity (10 Gbps). Figure 21 shows the results for the L3 routing application. As we can see, P4box overhead is smaller than 5% (0.4W) even in the worst case (i.e., when link utilization is maximum). Moreover, this overhead slightly decreases for lower utilizations.

We also measure the overhead for different applications and packet rates. In this case, we consider a line rate scenario where different packet sizes result in different packet rates, but do not affect the link utilization (always 100%) - similarly to the analysis performed in Section VI-C. Table III shows that P4box increases power consumption less than 0.5W for all applications. Interestingly, the overhead is smaller for higher packet rates. We believe this is because of the increased packet processing demand, which keeps more processing units (called Micro Engines - MEs in Netronome ASICs [22]) active/occupied along time for both approaches (i.e., with and without P4box).

Application	Packet size / rate					
	64 bytes / 14 Mpps			1500 bytes / 900 Kpps		
	w/o P4box	with P4box	%	w/o P4box	with P4box	%
Load balancing	12.79	12.92	+1.01	11.25	11.25	0
Surveillance protection	12.70	12.74	+0.31	11.21	11.32	+0.98
DDoS detection	12.63	12.71	+0.63	11.21	11.77	+4.99

TABLE III
AVERAGE POWER CONSUMPTION (IN WATTS) AT LINE RATE FOR DIFFERENT APPLICATIONS. STANDARD DEVIATION IS LESS THAN 0.1W.

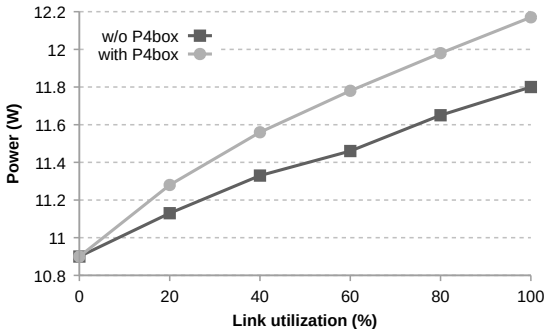


Fig. 21. Average SmartNIC power consumption for different link utilizations. Standard deviation is less than 0.1W.

VII. RELATED WORK

Network verification. Many tools have been proposed for verifying that a network behaves as expected. Moreover, these tools focus on either the control or the data plane. Minesweeper [6], Tiramisu [23] and Plankton [15] use models of networking protocols (e.g., BGP and OSPF) to analyze the network control plane. Although they can check multiple data plane configurations with this approach (i.e., the ones resulting from different protocol interactions), they are either restricted to a limited number of protocols or require long times for verifying large networks. Veriflow [24], APKeep [25], NoD [7] and SymNet [26], on the other hand, are data plane verifiers. They take a single data plane configuration (i.e., set of forwarding rules) as input, and check whether certain properties hold for all possible packets. Data plane verification approaches are typically not tied to any specific protocol, but network programmers need to manually build a separate model for each data plane program, which may be a cumbersome and error prone task.

P4v [12] and ASSERT-P4 [5] can automatically verify P4 programs, but they are able to check only program-specific properties. Vera [4], P4Nod [8] and P4K [27] create models for data plane programs that can be used as input to SymNet, NoD and the K framework, respectively. Although they can quickly verify small data plane programs (i.e., in the order of seconds), the verification time grows exponentially with both the program and the network size. Finally, p4pktgen [28] and p4rl [29] generate test packets for P4 programs. As for P4box, they can detect runtime bugs that are hard to find using static analysis techniques. However, both approaches are focused on a single data plane program.

Network debugging. Another dynamic approach to ensure security and correctness properties in networks is debugging. This approach is essentially based on monitoring and collecting statistics from network devices to perform an offline analysis. For example, Marple [30] proposes a query language for specifying monitoring tasks. Stroboscope [31] extends this idea and also considers scheduling to meet resource constraints. Instead of monitoring and collecting data, P4box processes information embedded on packets in switches at runtime. This design enables our mechanism to promptly react to property violations, containing them before they compromise a network policy. In-band Network Telemetry (INT)[7] provides flexibility similar to ours. However, it assumes information embedded on packets can not be compromised by buggy or malicious data plane programs. P4box, on the other hand, creates an isolated environment that can be used by network programmers to securely enforce policies of interest.

Runtime enforcement. The idea of using runtime monitors to enforce properties was first introduced by [32] in the context of system security more than forty years ago. In computer networks, FlowTags is a seminal work that proposed to extend middleboxes to add tags on packets which would be used by switches to enforce path conformance and origin binding [33]. However, unlike P4box, it does not take data plane programs and all possible bugs that come with them into account.

VIII. CONCLUSION

P4 and programmable data planes lowered the barrier for innovation in networking, but at the same time also made networks more prone to bugs and misconfigurations. To solve this problem we proposed P4box, a system for dynamically enforcing properties in programmable data planes through runtime monitors. P4box can enforce both program and network-wide properties while requiring a small effort from network programmers. Moreover, it represents a small overhead to network devices in terms of latency, throughput and power consumption.

Acknowledgments. This work has been supported by grants from NSF (CNS-1740911), RNP/CTIC (P4Sec), CNPq (140317/2017-1), and also by CAPES/Brazil – Finance Code 001.

REFERENCES

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[7] https://p4.org/assets/INT-current-spec.pdf

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

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

[4] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging p4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018, pp. 518–532.

[5] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos, "Verification of p4 programs in feasible time using assertions," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2018, pp. 73–85.

[6] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the ACM SIGCOMM Conference*, 2017, pp. 155–168.

[7] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 499–512.

[8] N. Lopes, N. Bjorner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese, "Automatically verifying reachability and well-formedness in p4 networks," Tech. Rep., September 2016.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the European Conference on Object-Oriented Programming*, 1997, pp. 220–242.

[10] M. Neves, B. Huffaker, K. Levchenko, and M. Barcellos, "Dynamic property enforcement in programmable data planes," in *2019 IFIP Networking Conference (IFIP Networking)*, 2019, pp. 1–9.

[11] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A formally verified nat," in *Proceedings of the ACM SIGCOMM Conference*, 2017, pp. 141–154.

[12] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018, pp. 490–503.

[13] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 209–224.

[14] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[15] "Plankton: Scalable network configuration verification through model checking," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, 2020, pp. 953–967.

[16] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the Internet Measurement Conference (IMC)*, 2015, pp. 275–287.

[17] P4 Consortium. (2018) Simple router. [Online]. Available: https://github.com/p4lang/p4app/tree/master/examples/simple_router.p4app

[18] S. Shi, C. Qian, Y. Yu, X. Li, Y. Zhang, and X. Li, "Concury: A fast and light-weighted software load balancer," 2019.

[19] . C. Lapolli, J. Adilson Marques, and L. P. Gaspary, "Offloading real-time ddos attack detection to programmable data planes," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 19–27.

[20] T. Datta, N. Feamster, J. Rexford, and L. Wang, "SPINE: Surveillance protection in the network elements," in *9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19)*, 2019.

[21] B. Stephens, A. Akella, and M. M. Swift, "Your programmable nic should be a programmable switch," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets)*, 2018, pp. 36–42.

[22] Netronome. (2014) The Joy of Micro-C. [Online]. Available: https://open-nfp.org/m/documents/the-joy-of-micro-c_fcjSfra.pdf

[23] "Tiramisu: Fast multilayer network verification," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, 2020, pp. 201–219.

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

[25] "Apkeep: Realtime verification for real networks," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, 2020, pp. 241–255.

[26] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proceedings of the ACM SIGCOMM Conference*, 2016, pp. 314–327.

[27] A. Kheradmand and G. Rosu, "P4k: A formal semantics of p4 and applications," *arXiv preprint arXiv:1804.01468*, 2018.

[28] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated test case generation for p4 programs," in *Proceedings of the Symposium on SDN Research*, ser. SOSR 18, 2018.

[29] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid, "Runtime verification of p4 switches with reinforcement learning," in *Proceedings of the 2019 Workshop on Network Meets AI and ML (NetAI)*, 2019, p. 17.

[30] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, 2017, pp. 85–98.

[31] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever, "Stroboscope: Declarative network monitoring on a budget," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 467–482.

[32] J. P. Anderson, "Computer security technology planning study," Air Force Electronic Systems Division, Tech. Rep., 1972.

[33] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 543–546.

Miguel Neves is currently a Ph.D. student and Part-time Professor at the Federal University of Rio Grande do Sul (UFRGS), Brazil. He received his B.Eng. in Computer Engineering from the same university in 2014. His research interests are in the interplay of program analysis, networking, security and distributed systems.

Bradley Huffaker has been working as a senior research programmer at CAIDA, Center for Applied Internet Data Analysis in the UC San Diego, since 2015. He received his Master Degree from UCSD in Mathematics and Computer Science. His focus is on visualization, DNS, geolocation, and Internet topology.

Kirill Levchenko is an Associate Professor at the University of Illinois at Urbana-Champaign. He received his Ph.D. from the University of California, San Diego in 2008 and his B.A. in Mathematics and Computer Science from the University of Illinois at Urbana-Champaign in 2001. His research applies evidence-based techniques to a broad range of computer and network security domains.

Marinho Barcellos has been with the University of Waikato, NZ, since October 2019. Prior to that, Marinho was a professor at the Federal University of Rio Grande do Sul - UFRGS (2010-2019). Marinho has contributed with research in a wide range of topics, including multicast, peer-to-peer, software-defined networks, programmable data planes, network security, and Internet measurements. As for public service, he has dedicated time to several conference organisations, and is presently a member of the ACM SIGCOMM executive committee, a co-chair of the CARES committee, and a member of two steering committees, ACM CoNEXT and PAM.