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

RICARDO PARIZOTTO

Consistent Code Composition and Modular Data Plane Programming

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

Advisor: Prof. Dr. Alberto Egon Schaeffer-Filho

Porto Alegre
May 2020

CIP — CATALOGING-IN-PUBLICATION

Parizotto, Ricardo

Consistent Code Composition and Modular Data Plane Programming / Ricardo Parizotto. – Porto Alegre: PPGC da UFRGS, 2020.

82 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Alberto Egon Schaeffer-Filho.

1. Software Defined Networks. 2. Programmability. 3. Data Plane. 4. Consistency. I. Schaeffer-Filho, Alberto Egon. II. Título.

“Nothing is so painful to the human mind as a great and sudden change.”

— MARY SHELLEY

ACKNOWLEDGEMENTS

First, I would like to thank my family: my mom, Natalina Parizotto, my father Nilvo Parizotto and my sisters Rafaela and Luciana Parizotto, for their love and support. Thank you for being so understanding when I missed countless family meetings because I was distant.

I would like to thank my advisor, Alberto Egon Schaeffer-Filho. Alberto has a remarkable ability to shape ideas and helped me keep focused on this project. He always expects the best from me, and this leads me to make a work that now I am really proud of. I will always be grateful for his guidance.

I would like to express my gratitude to my collaborators during the making of this thesis. I would like to give a special thanks to Lucas Castanheira, my co-author, who has special technical contributions to this work. It has been a pleasure to work with Arthur Jacobs, Fernanda Bonetti, and Luciano Zembruzki. I would like to give my special thanks to Rafael Riberio, my colleague since my time in undergraduate school and lab-mate. I also greatly enjoyed having Leonardo Lauryel friendship while I wrote this thesis.

ABSTRACT

Programmable Data Planes (PDP) enable more flexibility for the operation of networks. The various benefits of programmability have led the community to develop new software on both academic and industrial capacities. To fully reap the benefits of programmability, it should be feasible to compose and operate multiple PDP functions into a single target switch as needed. However, existing techniques are not suitable in the sense that they use an excessive number of parser states and tables, and lack abstractions for the steering of packets through the control flows. As such, they do not support modular composition of PDP functions. This thesis proposes PRIME, a composition mechanism of in-network functions that also addresses the fundamental needs of packet steering between PDP program modules. PRIME enables network operators to specify compositions of network functions written in P4 and how traffic traverses them. The composition employs a verification phase to identify ambiguities at source code level and avoid loops inside the switch pipeline. An additional table and a control plane management system enforce the steering of packets through control flows. We present a prototype of PRIME, along with a proof of the steering correctness. The results shows that it is possible to achieve module-wide compositions at little additional cost in terms of delay and throughput.

Keywords: Software Defined Networks. Programmability. Data Plane. Consistency.

Composição Consistente de Código e Programação Modular do Plano de Dados

RESUMO

Planos de dados programáveis (PDP) permitem mais flexibilidade para a operação de redes. Os vários benefícios da programabilidade levaram a comunidade a desenvolver novos softwares, tanto na academia quanto na indústria. Para aproveitar plenamente os benefícios da programabilidade, deve ser possível compor e operar várias funções do PDP em um único switch de destino, conforme necessário. No entanto, as técnicas existentes não são adequadas no sentido em que usam um número excessivo de estados e tabelas de encaminhamento e não possuem abstrações para o direcionamento interno de pacotes através dos fluxos de controle. Portanto, as técnicas existentes não suportam a composição modular de funções ao PDP. Esta dissertação propõe PRIME, um mecanismo de composição de funções em rede que também atende às necessidades fundamentais do direcionamento interno de pacotes entre os módulos de um programa PDP. PRIME permite que os operadores de rede especifiquem composições de funções de rede escritas em P4 e como o tráfego as atravessa. A composição emprega uma fase de verificação para identificar ambiguidades em nível do código fonte e evitar loops dentro do pipeline do switch. Uma tabela adicional e um sistema de gerenciamento para o plano de controle garantem o direcionamento de pacotes através dos fluxos de controle. Apresentamos um protótipo do PRIME, juntamente com uma prova da corretude do módulo de direcionamento de tráfego. Os resultados mostram que é possível obter composições de módulos com pouco custo adicional em termos de atraso e taxa de transferência.

Palavras-chave: Redes Definidas por Software, Programabilidade, Plano de Dados, Consistência.

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic Logic Unit
ASIC	Application-specific integrated circuit
CPU	Central Processing Unit
DPI	Deep Packet Inspection
FPGA	Field Programmable Gate Array
ICF	In-Network Computing Function
INT	In-Band Network Telemetry
MAT	Match-Action Tables
MAU	Match-Action Units
NFV	Network Function Virtualization
NF	Network Function
P4	Programming Protocol-independent Packet Processor
PDP	Programmable Data Planes
PISA	Protocol Independent Switch Architecture
SDN	Software Defined Network
SRAM	Static Random-Access Memory
SFC	Service Function Chaining
TCAM	Ternary Content Addressable Memory

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Software-based paradigms for networking enable decoupling software solutions from the hardware in which they execute, making the management and operation of the network infrastructure more flexible and adaptive. Software-Defined Networking (SDN) (FEAMSTER; REXFORD; ZEGURA, 2014) promotes the separation of the control logic from the forwarding behavior of network devices. More recently, Programmable Data Planes (PDP) offer more flexibility in the development of protocols and network functionality by allowing packet processing at the line rate in the switch itself. This motivated many emerging applications, such as NetCache (JIN et al., 2017) and P4xos (DANG et al., 2016), to bring part of the processing back to the data plane to achieve economies of scale and lower operating costs. As such, operators can leverage programmable hardware to, for instance, process or analyze data (MUSTARD et al., 2019), thereby enabling faster reactions in contrast to packet mirroring to middleboxes or controller-based applications (SONCHACK et al., 2016; ERAN et al., 2019).

1.1 Problem Statement

Rather than writing monolithic functions, it should be straightforward for PDP software to be shared and composed into switches as needed (FREIRE et al., 2018)(LIU et al., 2018)(BENSON, 2019). However, existing languages for data plane programming do not support modular development. P4 (“Programming Protocol-independent Packet Processors”) (BOSSHART et al., 2014), one of the most popular languages for PDPs, requires developers to perform extensive modifications into the function source code to deploy it on existing applications. For instance, if a network operator wants to install a new program in a switch that is already running a P4 program, both programs would require modifications. As a result, researchers have responded by offering composition instances that dedicate multiple PDP functions to the same physical target (HANCOCK; MERWE, 2016). Composition typically refers to code merging techniques or virtualization techniques (SAQUETTI et al., 2019b; SAQUETTI et al., 2019a; KRUDE et al., 2019; ZHANG et al., 2019; LI et al., 2017; LI et al., 2018), which can both be utilized as a programming model (ZHOU; BI, 2017) or for the automation of development. Hence, composition try to avoid rewriting code from different functions manually and maintains the semantics of the system.

Unfortunately, while composing multiple functions may promote better usage of network resources, the management becomes more complex and error-prone. Current efforts to compose various programs in a single target switch make use of an excessive number of flow tables and parser states (HANCOCK; MERWE, 2016; ZHANG et al., 2017; ZHENG; BENSON; HU, 2018). Consequently, these techniques can severely limit throughput and increase latency in general-purpose hardware or do not fit in specialized hardware, such as netFPGAs (SAQUETTI et al., 2019b) or ASICs (DANG et al., 2017).

Moreover, state-of-the-art techniques do not suffice to provide transitional consistency between steering configurations. Without transitional consistency, changes in the steering of flows through the program modules can create intermediary states, which may cause misrouting and security holes (HAN et al., 2015) (REITBLATT et al., 2012) (MATTOS; DUARTE; PUJOLLE, 2016). Thus, new techniques are required to allow new applications to be composed, preserving transitional packet-consistency of traffic steering and without degrading the performance of the data plane operation (HE et al., 2019).

1.2 Research Goals

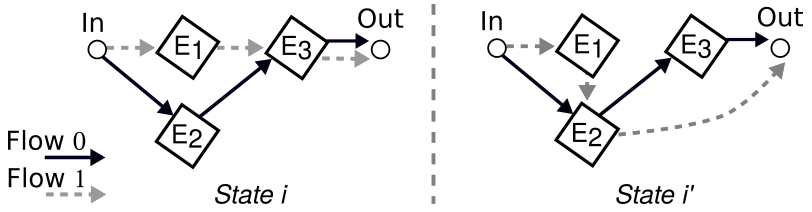
We summarize the research goals of this thesis as follows:

Composition of Programs. We need to provide a data plane composition mechanism, allowing operators to use the mechanism on different scenarios. The composition must allow different data plane programs to share the same switch resources, allowing better resource usage compared to placing monolithic functions on sequences of different switches. The process of sharing resources must provide intuitions about source code merging to improve resource utilization.

Steering Definition. Composing multiple P4 programs into devices brings together the necessity of abstractions to steer flows through the composed modules. This, in turn, creates new difficulties for the network operation. We must provide ways to steer packets internally between composed programs. The steering configuration must be easy to manage and semantically coherent with the policy specified by the network operator (HAN et al., 2015), i.e., we must support dynamic steering of internal functions avoiding that each update creates state configurations with intermediary states. For example, Figure 1.1 presents two different states of traffic steering. In the example, network state i steers Flow 1 packets through programs $(E1, E3)$ and Flow 0 through programs $(E2, E3)$, respectively. For some reason, it might be desirable to achieve a transition between the state

configuration i to state i', in which $(E1, E2)$ process Flow 1. However, this change of configuration is error-prone and can create undesirable intermediary states, *i.e.,* a packet may see part of state *i* and part of state *i'*. In the example, an intermediary state can be created by performing the update of $E1$ before updating $E2$, leading a new packet to reach $E2$ without having the proper instructions to process it.

Figure 1.1: Switch state transition



Source: The Author

1.3 Summary of Contributions

As programmers may want to instantiate programs without rewriting their constructs (e.g., tables, actions or parser states), we present a new mechanism to compose P4 modules, which we call PRIME (Programming In-Network Modular Extensions). The developer can perform the composition of functions placing an ordered set of programs (e.g., security functions, including firewalls, access controls, and DPIs) and isolating resources between them. A source code analysis phase detects and corrects ambiguities between the control flow of modules, consequently avoiding undesired loops inside the switch pipeline. Dynamically, PRIME allows network administrators to specify the steering of traffic through the composed programs. The key insight is to deploy programs statically and use the per-packet state to steer flows using one single additional table. PRIME then provides a control plane interface to specify steering updates and send the necessary table entries to switches.

Overall, this thesis makes the following contributions:

- Identifies a set of requirements that network operators would require to compose multiple functions at a single switch. Besides, it describes an architecture that can be implemented without changes to the network hardware.
- Explores state machine composition techniques for providing an extension operator for the P4 language that merges independent parsers.
- Designs a composition model for P4 programs and provides a control plane in-

terface to steer flows through the composed programs. The interface provides the means to update the configuration without creating intermediary states.

1.4 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 presents a brief overview of Software Defined Networks and Programmable Data Planes. Chapter 3 presents the main design principles of PRIME, as well as its architecture and algorithms. Chapter 4 presents implementation details. Chapter 5 presents the evaluation methodology, use cases we built using existing P4 applications, and results obtained using the bmv2 software switch. Chapter 6 presents the main related work and discussions. Finally, Chapter 7 presents the conclusions and future work.

2 BACKGROUND

In this chapter, we present the essential background for this work. Section 2.1 presents Software-Defined Networking (SDN) and its architecture. Next, Section 2.2 presents technical concepts about Programmable Data Planes (PDP). We finish the section showing details of the P4 language and its configuration.

2.1 Software-Defined Networking

Software-Defined Networking (SDN) is a networking paradigm that brings software abstractions to build network behavior. The main difference of SDN to traditional networks is the decoupling of the control plane from the forwarding devices (data plane) (FEAMSTER; REXFORD; ZEGURA, 2014). In SDN, network devices become simple forwarding devices while the "brain" is implemented on the controller.

This paradigm brings several advantages compared to legacy methods. First, an advantage of SDN is the simplification of the instantiation of new applications in the control plane when compared to traditional networks, where the data and control planes are tightly coupled and typically operate in a distributed design (JARRAYA; MADI; DEBBABI, 2014). Second, with SDN, it is much easier to introduce new ideas to the network through a software program, as it is easier to manipulate the network behavior through a fixed set of commands (KIM; FEAMSTER, 2013). To this end, the architecture defines conceptual planes and communication interfaces. Next, we discuss the architecture in detail and how the conceptual planes are related.

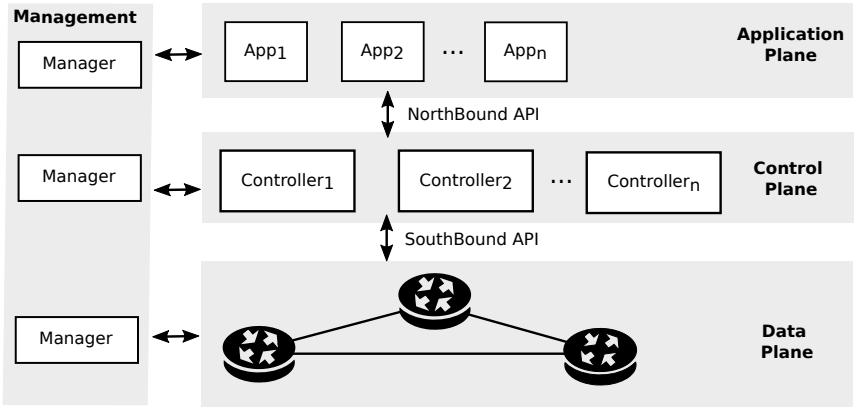
2.1.1 SDN Architecture & Conceptual Planes

The SDN architecture, depicted in Figure 2.1, defines four conceptual planes and communication interfaces.

The Management Plane is responsible for monitoring, configuring, and managing the network application behavior on each plane, e.g., making decisions according to the network state. The management plane can be utilized to configure the data plane; however, it does so infrequently.

The Application Plane is responsible for executing applications within the net-

Figure 2.1: SDN High-level architecture



Source: The Author

work infrastructure. Applications can build on an abstract view of the network to gather information and make decisions. Examples of such applications are interfaces for network visualization, load balancers, or analytics applications to detect suspicious network behaviors for security.

The Control Plane defines the control logic, such as the implementation of routing mechanisms. It is important to say that a logically centralized programming model given by SDN does not imply a physically centralized system (KOPONEN et al., 2010). In general, the control plane is composed of one or more controllers that monitor and configure the data plane behavior (KREUTZ et al., 2014). In fact, such distribution usually is made to obtain adequate levels of performance and reliability for the network (HELLER; SHERWOOD; MCKEOWN, 2012).

The Data Plane includes devices responsible for forwarding packets (generally referred to as switches). The data plane only forwards packets according to rules received from the control plane, i.e., they only take one packet from an input port and send to an output port according to the forwarding rules. Traditionally, switches have the functionality fixed by application-specific integrated circuits (ASICs).

The controller operating system can offer an API to application developers. This API is a common interface for developing applications. Typically we refer to this API as the Northbound interface. The northbound interface abstracts low-level details to program forwarding devices. Besides, the Southbound API defines the instruction set of forwarding devices, which composes the Southbound Interface. Furthermore, the Southbound Interface formalizes the way that the control and data plane interact (KREUTZ et al., 2014).

2.1.2 OpenFlow

OpenFlow (MCKEOWN et al., 2008a) proposes a new protocol to perform experimental tests on networks operators use every day. OpenFlow is the most widely accepted and deployed open southbound standard for SDN (JAIN et al., 2013). The protocol provides three sources of information: First, event messages are sent when a link or port change is triggered. Second, flow statistics are generated by the switch and collected by the SDN controller. Thirty, a warning is sent to the controller when a new flow arrives, and the switch does not know what to do with it, i.e., it still does not have rules to process it (KREUTZ et al., 2014).

An OpenFlow switch is composed of three main parts: (1) the forwarding table, which is associated with processing actions for each table entry; (2) a secure channel for the switch to communicate with the control plane; and (3) the OpenFlow protocol, which provides an open and standard way for the control plane to interact with the OpenFlow switch. Each flow entry has simple actions associated with it: (1) forward, (2) encapsulate and send to a controller, and (3) drop. The forward of packets map a flow to a given output port. Such characteristics of the forwarding enable packets to be routed by the network. Encapsulating packets to the control plane is typically performed for the first packet from a flow in order to the controller to calculate a new route and install it on the forwarding table (FOSTER et al., 2011). Finally, drop packets can be used for security reasons to eliminate denial of service attacks or reduce discovery attacks to end hosts (SONCHACK et al., 2016).

OpenFlow, however, is limited to a strict set of header fields and actions. To overcome this limitation, the community is making efforts to make reconfigurable switches (BIFULCO; RÉTVÁRI, 2018). Emerging switch architectures are enabling programmers to reconfigure header fields and write their actions (HALEPLIDIS et al., 2015). The reconfigurable architectures result in promising ways to leverage the network hardware and their APIs to management. The main aspects of Programmable Data Planes are synthetized next.

2.2 Programmable Data Planes

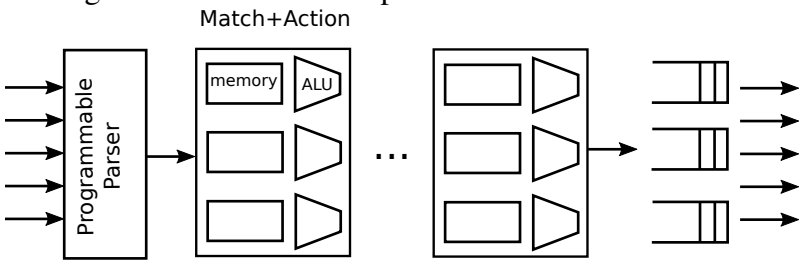
Data Plane Programmability has been proposed as a means to deploy new features on forwarding devices without the need to buy new hardware. The development of speci-

fication languages such as P4 (BOSSHART et al., 2014) enabled operators to change the behavior of programmable switches without rewriting low-level instructions (*e.g.*, the kernel of OvS (PFAFF et al., 2015), integrated circuits of hardware switches, or components of simulation environments).

2.2.1 Protocol Independent Switch Architecture

The Protocol Independent Switch Architecture (PISA) architecture is one of the technologies which enabled programmability. Figure 2.2 presents the standard PISA architecture. The architecture divides the forwarding model into a programmable packet parser, a pipeline of *match+action* logic, and a packet deparser.

Figure 2.2: Protocol Independent Switch Architecture



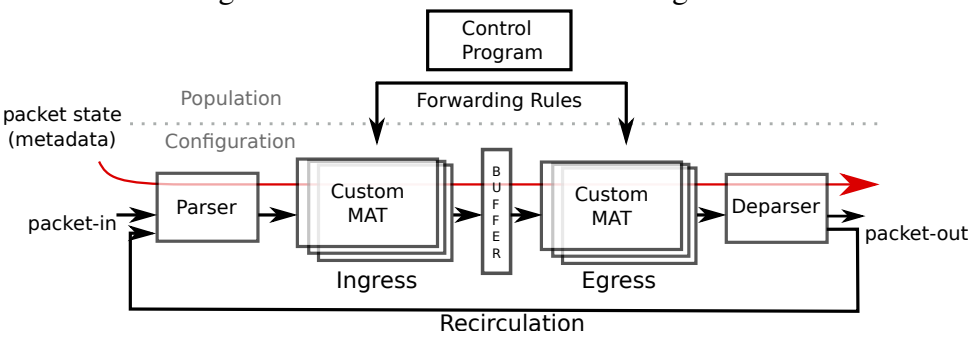
Source: The Author

The *match+action* logic is a mix of SRAM and TCAM for lookup tables, counters, meters, and generic hash tables. The action logic is composed of ALU's and stands for standard boolean and arithmetic operations, header modification operations, hashing operations, etc. Programmable forwarding devices do not understand any protocols until they get programmed. Programming languages, such as P4 (BOSSHART et al., 2014), Domino (SIVARAMAN et al., 2016) and POX (LI et al., 2017), can specify a logic data plane view and be mapped to the physical resources. In this work, we focus on P4, which is the most popular language for the PDP.

2.2.2 *P4: High-Level Data Plane Programming Language*

P4 allows programming and configuration of forwarding devices, including specific actions or control calls. In contrast to standard OpenFlow switches (MCKEOWN et al., 2008b), P4 enables network developers to build programs that modify the structure of packet headers and can store complex network state on the data plane.

Figure 2.3: Abstract Packet Processing Model



Source: The Author

A P4 architecture model is a contract between the P4 developer and the switch manufacturer. Therefore, each switch manufacturer must provide a P4 compiler as well as an architecture pattern for their hardware. Figure 2.3 shows an example of a P4 abstraction to configure a PISA-based switch. The abstraction divides the data plane behavior into three main blocks: the Parser, Control flows, and the Deparser:

Parser. P4 declares a parser state machine that describes how to extract headers from incoming packets. The programmer declares names for each header field, so the parser can reference bit fields into typed data, which the programmer can reference next (JOSE et al., 2015).

Figure 2.4: Example of Packet Parser

```
1  parser Parser(packet_in packet,...) {
2      state start {transition parse_ethernet;}
3
4      state parse_ethernet {
5          packet.extract(hdr.ethernet);
6          transition select(hdr.ethernet.etherType) {
7              TYPE_IPV4: parse_ipv4;
8              default: accept;
9          }
10     }
11     state parse_ipv4 {...}
12 }
```

Source: The Author

An example P4 parser is presented in Figure 2.4, where an `ethernet` header is extracted, and a transition can be made to extract `IPv4` header using the `etherType` field value. After the parser processes a packet, the packet follows to a pipeline of control flows.

Control Flows. Each control flow is composed of a set of logical *match+action* tables implemented using *match+action units* (MAUs). An *apply* block specifies the se-

mantics and order that each MAU processes packets, reads, and modifies the content of header attributes instantiated by the parser.

Figure 2.5: Example Ingress Control Flow

```
control Ingress(inout headers hdr, ...) {
    action ipv4_forward(macAddr_t dstAddr) {...}

    table ipv4_lpm {
        key = {hdr.ipv4.dstAddr: lpm;}
        actions = {ipv4_forward;}
        size = 1024;
    }
    apply {ipv4_lpm.apply();}
}
```

Source: The Author

In the example of Figure 2.5, the ingress *match+action* pipeline implements a single table that routes 32-Bit IPv4 addresses using least prefix matching with at most 1024 entries. After the pipeline of control flows processes a packet, it follows to the deparser.

Deparser. The Deparser writes internal variables to the packet header and emits the packet to an output port (or recirculates it back to the parser). In practice, typed data are assembled into bit fields of the packet header. In the example of Figure 2.6, the switch first assembles `ethernet` header and then `IPv4`.

Figure 2.6: Example Deparser

```
control MyDeparser(packet_out packet, in headers
    hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}
```

Source: The Author

The PDP abstraction divides the forwarding model into two stages: the *configuration* and the *population*.

During the *configuration*, developers can program the parser state machine, the structure of MAUs, and the semantics of control flows. In this phase, the developer also defines header structures and internal registers. Packets can carry variables during its processing, called *metadata*. Examples of metadata are output ports, timestamps of opera-

tions, and table-to-table states. Such examples are common in the underlying architecture of P4, but developers can create their own metadata to develop new applications.

The *population* stage allows the operator to insert, remove, or modify entries of the stateful objects, such as tables and registers that were created during the *configuration* phase. In the case of P4, the language does not dictate table update behavior. Therefore it is necessary to build tools on top of P4 to provide an update command for a different target switch, *i.e.*, when a packet matches a rule, an action is invoked with parameters supplied by a control program.

3 PRIME: DESIGN AND ALGORITHMS

This chapter presents the system design and the algorithms used in this work. First, Section 3.1 presents an overview of PRIME, as well as the system architecture and terminology. Next, Section 3.2 presents the composition of packet parsers and deparsers. Section 3.3 describes the composition of control flows and a source code analysis method. Finally, Section 3.4 presents how to perform the steering of packets, and a proof of consistency.

3.1 Overview

In this section, we describe an overview of PRIME (Programming In-Network Modular Extensions), a mechanism for network administrators to compose different PDP programs in each switch of the network.

PRIME is responsible for two main tasks: providing an P4 program as a basis for the composition of functions and generating a programming interface (API) to steer packets through program modules (e.g., tables) at run-time. As such, the system enables network operators to easily deploy only the necessary modules in each switch without rewriting code to build different configurations with the available programs. PRIME has a controller interface, which interacts through an API with an base program. The components of the base program were based in P4 and provide a programming model to host compositions of in-network functions written as P4 programs for the data plane. Finally, this is all supported by the PISA architecture on a target switch.

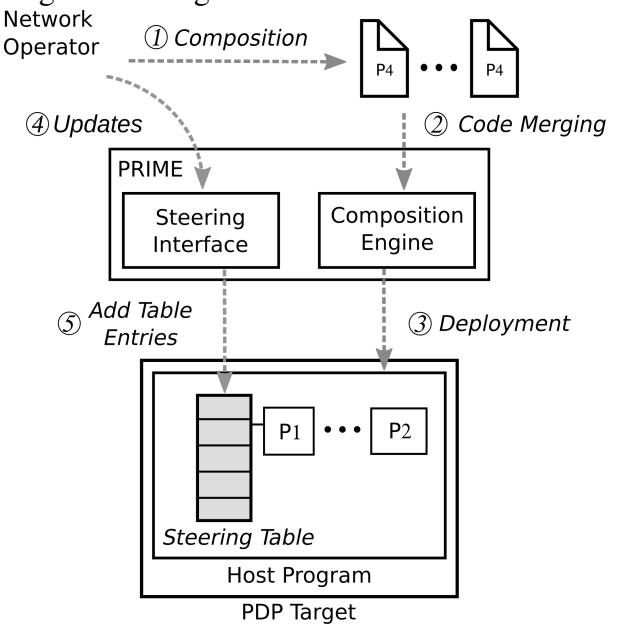
The function composition must preserve the following constraints:

- **Function Isolation:** the arrangement and composition must isolate network functions. The isolation ensures that variables from different programs do not overlap.
- **Loop-Freedom:** both the steering and composition must ensure packets do not loop inside the switch.
- **Consistent Updates:** updating the set of compositions of flows must be made consistently, i.e., must not create intermediary states.

To identify the elements of the system as unequivocally as possible, we now present the essential terminology used in this work: A *Program* is a syntactically correct P4 program, which can be verified by the language compiler. A *Program Module*

is a P4 program control block. A program module can be an ingress control or only a packet parser that will be composed. *Extensions* are program modules required to be on the host program. Each specific computation that can be invocated as part of a P4 program is a *Function*. A Program may contain more than one function, and such information is transparent for the composition mechanism.

Figure 3.1: High-Level Architecture of PRIME.



Source: The Author

Figure 3.1 presents the high-level architecture of PRIME. Firstly, network operators write separated and independent programs, running independently from each other (Figure 3.1, Step 1). Secondly, the source code of multiple programs is merged into a host program (Figure 3.1, Step 2), which provides *primitives* to steer packets through them. The composition performs an analysis of the packet parsers and control flows to ensure the program will operate with no loops or ambiguous states. If the merged code passes this analysis, the new program is deployed on the switch (Figure 3.1, Step 3). Network operators may define which sequences of programs will process a flow dynamically by using a steering interface. The interface updates the switch state to multiplex packets to a specific set of program modules (functions) (Figure 3.1, Step 4). Specifically, the steering interface produces switch table entries and installs them on the switch (Figure 3.1, Step 5).

After discussing the design of the system, we now present in detail how to compose data plane programs.

3.2 Combining Header Instances

Given a set of program extensions and the host program, the composition aggregates the functionalities of the set of extensions to the host program. The system assumes that each extension is syntactically correct and verified by the standard P4 compiler to perform the composition. Then, the system computes the composition by scanning parsers and control flows and merges the respective structure definitions according to the semantics of the composition and the characteristics of the modules themselves. These aspects are explained in detail below.

3.2.1 Extending Parsing Trees

We assume for the extensions a general definition of packet parser trees, which can fit most usual composition techniques. A parser tree is defined as an oriented graph $PG = (V, E)$, where each node represents a packet header, and each transition represents the next protocol header. Let the composition operation on packet parsers of an extension E and a host program H be $C : \Gamma_E \times \Gamma_H \mapsto \Gamma_L$. We define the composition of parsers as the union of the set of terminal states, non-terminals, transitions, and header definitions of the extension.

Algorithm 1: Handling the extension of parsers

Data: $PG_E = (S_E, T_E)$, $PG_H = (S_H, T_H)$
Result: $parser_{out}$

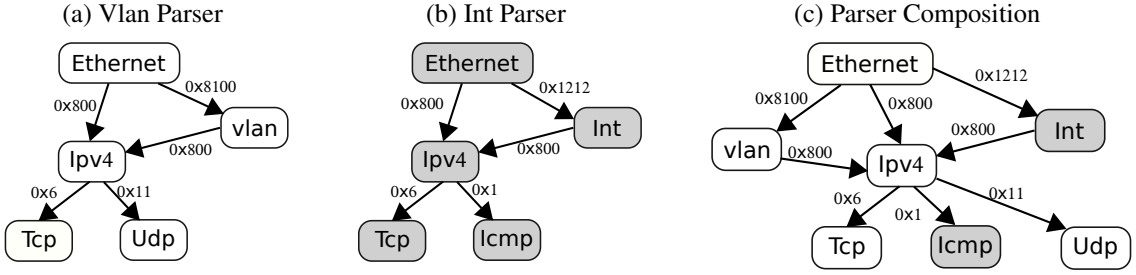
```
1 begin
2     for state ∈ S_E do
3         if (state ∉ S_H) then
4             S_H ← S_H ∪ {state}
5         for transition ∈ T_E do
6             if transition ∉ T_H then
7                 T_H ← T_H ∪ {transition}
8     Use in-depth search algorithm to identify loops and non-determinism
9     return Parser_out
```

Algorithm 1 presents a pseudo-code for handling the extension between two parser-state machines. The composition result is a new parser state machine Γ_L, which merges states with the same ID; and performs the union of state transitions from the extension and the host parser.

Conflict-Free State Machines. To ensure that the composition is correct, we constrain the scope of extensions, requiring the resulting parser to be deterministic and loop-free. It is difficult to find the equivalence between two headers, therefore we consider that two vertices are equivalent when they have the same header ID. These are restrictions we need to enforce in order to ensure the composition of parser operates correctly. We say that $\Gamma_E \ extends \ \Gamma_H$ if Γ_E satisfies the restrictions imposed by Γ_H. For this, after the modules are interpreted and merged, PRIME performs a custom verification phase (SCHWERDFEGER; WYK, 2009).

The verification performs an in-depth search in the result of the union of packet parsers. Once each packet parser is a tree, the composition can create a graph with loops and non-determinism, and in this case, the verification will find existing loops. If the composed code passes this analysis, *i.e.,* $\Gamma_E \ extends \ \Gamma_H$, then $\Gamma_L \ = \ \Gamma_E \ C \ \Gamma_H$, it can be considered "certified" and safely composed with the host program. Otherwise, the administrator is notified with a warning. After receiving a warning, the administrator should manually rename states and transitions that created the inconsistency. This would also require to rewrite control flows invocations in the case that the object that created the error is a header field. We intend to investigate ways to repair those cases automatically, *e.g.*, recirculating packets through a new independent parser that could not be merged because of the host program restrictions (PRAKASH et al., 2015).

Figure 3.2: Graphic representation of packet parser trees composition



Source: The Author

Figure 3.2 presents an example of the composition of the two parser state machines that are shown in Figure 3.2a and Figure 3.2b. The composition result, depicted in Figure 3.2c, merges Ethernet, which now has the transition `0x8100` to Vlan and `0x1212` to INT. Finally, State ICMP is included in the parser with a transition `0x1` from already known State IPv4. The inclusion of a new state also carries its header definitions, *i.e.*, the composition merges the definitions of packet header and the state ICMP into the composed program (GIBB et al., 2013).

3.2.2 Deparsing

Each state composed with the program must be carefully emitted to ensure packets are well-formed (LOPES et al., 2016). For instance, the system should not emit IPv4 headers before emitting TCP, which could impact on out-of-order read/writes on the next network hop. Once the structure of deparsers does not convey sufficient information to establish a dependency among them, we cannot infer the order in which packet headers must be emitted in the composed program. To avoid that disruption, the composition of deparsers must (1) unite the set of emitted headers from both programs, and (2) create a new deparser that emits headers in the same order as they are instantiated by the parser (SONI; TURLETTI; DABBOUS, 2018).

3.3 Control Flow Arrangement

Control flows of P4 programs include additional definitions of actions, tables and conditional branches (*if-else* statements) inside of control blocks. To extend functionalities of two control blocks, we present a programming operator to compose P4 programs, enabling the network administrator to isolate control flow blocks in a static manner (JIN et al., 2015)(SUN et al., 2017).

3.3.1 Function Cataloging

The composition aggregates ingress and egress modules into an additional table in the host program, which we call the *"steering table"* according to the semantics of the composition operator and the constructs of the P4 program. The composition operator can be utilized between two P4 programs to place the control flow of a new extension to the beginning of the pipeline of the steering table. In practice, control flows of the programs appear in the host program in the order in which they were composed.

Merging tables may promote space optimization, but creates the possibility of violating target-independent constraints, such as the equivalence between table structures, table dependencies, and loop-freeness (which is a restriction imposed by the P4 language and the data plane itself) (ZHENG; BENSON; HU, 2018). To ensure the merging does not break target-independent constraints, the merging operator provides isolation of tables,

registers, and actions. Therefore, the semantics of the host program is preserved by not allowing a function to rewrite it.

3.3.2 Constructs Disambiguation

PRIME performs a source code analysis step to identify the equivalence of structures between the composed tables and ensure they do not violate table dependencies. For tables with ambiguous IDs, PRIME renames their IDs and rewrites the *"apply"* construct for the merged structure to use the proper ID and preserve dependencies of both modules (SAHA; SAMANTA; SARANGI, 2009). The same isolation is performed for registers, actions, and metadata definitions with ambiguous IDs.

Algorithm 2: Modules merging and verification

Data: $host.p4, module.p4$
Result: $host$
begin
  $Ambiguous = \{\}$
  //for all tables
  for $t \in pipeline$ **do**
    if $t \in H_t$ **then**
      $Ambiguous = Ambiguous \cup \{t\}$;
    else
      $H_t = H_t \cup \{t\}$;
  //for all actions
  for $a \in pipeline$ **do**
    if $a \in H_a$ **then**
      $Ambiguous = Ambiguous \cup \{a\}$;
    else
      $H_a = H_a \cup \{a\}$
  //for all registers
  for $r \in pipeline$ **do**
    if $r \in H_t$ **then**
      $Ambiguous = Ambiguous \cup \{r\}$;
    else
      $H_r = H_r \cup \{r\}$;
  return $host$

Algorithm 2 traverses all tables, actions, and registers defined on the new function and associates them with the host program. In order to do this, we first collect all defined registers, actions, and tables in both pipelines. Afterward, we disambiguate repeated IDs and rewrite the statements in the apply struct (Algorithm 3).

Algorithm 3: Constructs Disambiguation

```
Data: newprogram.p4, Ambiguous
Result: host
1 begin
2     for each node n in pipeline do
3         if n is START or not a invocation statement then
4             continue;
5         if n ∈ Ambiguous then
6             rename.invocation(n, newprogram);
```

With the aid of the *"steering table"*, the composition produces a sequence of program modules whose execution order can be altered dynamically. For instance, the composition can change the order of execution of a firewall and a load balancing. Specifically, a firewall must be applied before load balancing incoming packets as the firewall must consider the original IP addresses. Conversely, the load balancer must first restore the original IP address before the firewall handles outgoing packets (MONSANTO et al., 2013).

The structure of the host program and the composition assures a data plane structure that allows the configuration of both directions. Each composition translates into a configuration that works as a link for a sequence of program control flows. The steering table is positioned at the beginning of the switch pipeline and intercepts all incoming packets. The table specification can match packets using wildcards, lpm or exact and works as a large catalog of pointers from specific sets of packets to sequences of program modules merged during the composition (CHEN; BENSON, 2017; SONCHACK et al., 2016) (a process similar to service function chaining (SFC) in the context of Network Function Virtualization (MIJUMBI et al., 2016)).

3.4 Traffic Control

When the network administrator wishes to steer packets for a specific sequence of programs, s/he describes the identifier of the flow and the sequence of modules that must process this flow. A configuration C is defined as a pair of switch programs P and the steering function S. A switch state N is a pair (Q, C) containing a set of flows Q and the configuration C. The system has two kinds of transitions: recirculations (rounds) and updates transitions.

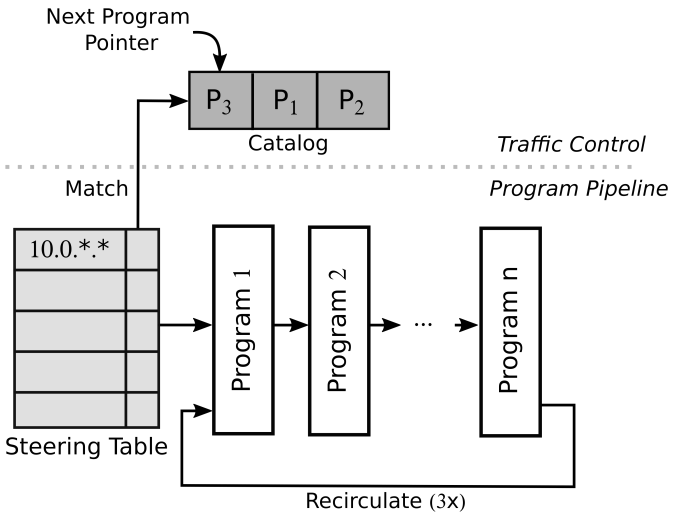
During a recirculation, a packet is retrieved from the egress and sent to the next

program using the set of switch programs P, and the steering function S. A recirculation denotes the traversing of a packet through the pipeline of programs. In each round, only one of the programs process the packet. The host program utilizes a traffic control module to deliver the packet to the program indexed by the next program of the catalog. After a packet reaches the egress, the next program indicator is updated, and the packet recirculates to start another round. This repeats until the packet is processed by all the programs indexed by the configuration C.

3.4.1 Consistent Update

In an update transition, the switch forwarding is updated to a new behavior for a specific flow. We represent an update as a partial function from local packets to a list of programs. To apply an update, PRIME then translates the code to the tuple of parameters of the steering table. When an incoming packet matches the table, an action that we call 'catalog' loads the parameters supplied by the administrator to the internal state. Subsequently, these user-supplied parameters will be stored as packet metadata and used by the host program to determine the order in which program modules are processed.

Figure 3.3: Traffic steering through program modules



Source: The Author

Figure 3.3 presents an example of how the steering table can map flows to sequences of programs. In the example, packets that match `10.0.*.*` are mapped to be processed by programs P_3, P_1 and P_2 respectively. For this, after matching the table, the

catalog points to the ingress control-flow of P_3 and follow to its egress. Next, the packet recirculates and follows to P_1 ingress and egress. Finally, the packet recirculates a third time to P_2. It is important to note that the same data plane structure supports the execution in a different order if the network administrator wishes.

3.4.2 Enforcing Correctness

To ensure packets will not face intermediary states of steering configurations, we reduce our problem to the *transitional per-packet-consistent updates* problem (HAN et al., 2015). Per-packet states for a given packet are consistent if the traces generated during the update are generated from the previous configuration, or from the new configuration, but not from a mixture of the two (REITBLATT et al., 2012).

We explicitly state the invariants enforced by the host program and the merging itself. The host program maintains the following invariants:

- *(H1)* All functions in the pipeline are ordered linearly in the order they are merged, and each packet follows the pipeline in order. Effectively, the set of functions IDs is the set of naturals numbers, and ordering follows directly from that.
- (*H2*) The configuration of steering is only loaded into a packet metadata in the first round, thus preventing the configuration of a transient flow from being changed by the table on other rounds.
- (*H3*) No changes are accepted into the steering configuration while it is already updating (this occurs because the action that loads the configuration is atomic).
- (*H4*) Standard metadata (*e.g*, output ports) are copied into user-metadata before re-circulating and restored into new standard values to index functions into the correct processing order.
- (*H5*) The active steering configuration and packet header are recirculated only when *function ID* $<$ *# of Programs*. The next function to be processed is then updated according to the values of steering.

To ensure the composition does not violate these invariants, the merging must act accordingly. For this, the program merging satisfies the following invariants:

- (*M1*) Metadata definitions are verified and disambiguated to ensure no function code modify the catalog structure.

- (*M2*) Each table and action is disambiguated to ensure composed function code do not rewrite the catalog of a packet or apply the steering table.

We now can use these invariants to prove transitional packet-consistent updates for the steering configuration in the switch pipeline.

Lemma 1. *If pkt carries C_1 configuration, where $|C_1| > 1$, it is not possible for any update to change its configuration until the end of its processing.*

Proof. We proceed by induction over m, the configuration length, noting that the base case, $m = 0$, is consistent. Assume that an update occurs: H2, M1, M2 ensure that when C_1 is loaded, the steering trace of pkt is not modified until emitted by the egress. H4 ensures that pkt keeps the same steering configuration C after recirculating. Therefore, to all processing rounds $r_0, ..., r_n$ of pkt, the configuration in round n is the initial configuration C_1. H5 ensures that packets do not recirculate forever. H1 ensures that a packet pkt with steering configuration C always crosses every program in the pipeline and, by the previous conclusions, finds each program in the steering configuration, C_1.

□

Theorem 2. *For all configurations C_1 and $C_2(P[p_1, ...p_k], S)$., updating from C_1 to C_2 is per-packet consistent.*

Proof. The proof proceeds by considering every trace generated during the execution of the update. There are two cases: In case (1) the transient packet creates a new state for disambiguation. In this case, PRIME has the C_1 loaded into packet metadata. In case (2) packets enter the switch with the new configuration C_2. In case (1), the traces can be generated by no update operations, and the definition of per-packet consistency holds directly from Lemma 1. In case (2) we denote pkt as the first packet entering the switch tagged with a new steering configuration C_2, written $[p_1, ...p_k]$. H3 ensures that while the steering table is loading C_2, no other update can be performed into the steering configuration of pkt. H1 proves that when pkt exists in the pipeline, all the programs in the pipeline are updated to C_2, even if pkt is marked to be dropped by a program in the pipeline. Hence pkt and all subsequent packets tagged with C_2 are processed with the new configuration. □

We claim that although the implementation of the composition of multiple programs in the same switch pipeline appears straightforward, configuring the traffic steering requires the switch to preserve certain invariants. Consistency is made possible because

P4 provides per-packet states (metadata). However, metadata still needs to be copied into user-metadata before recirculating. We hope that our work provides a good motivation to rethink the design of the metadata system to facilitate the correct steering conceptually. In this section, we have shown what invariants to preserve, and why they suffice for a correct implementation of a packet-consistent steering configuration (HAN et al., 2015).

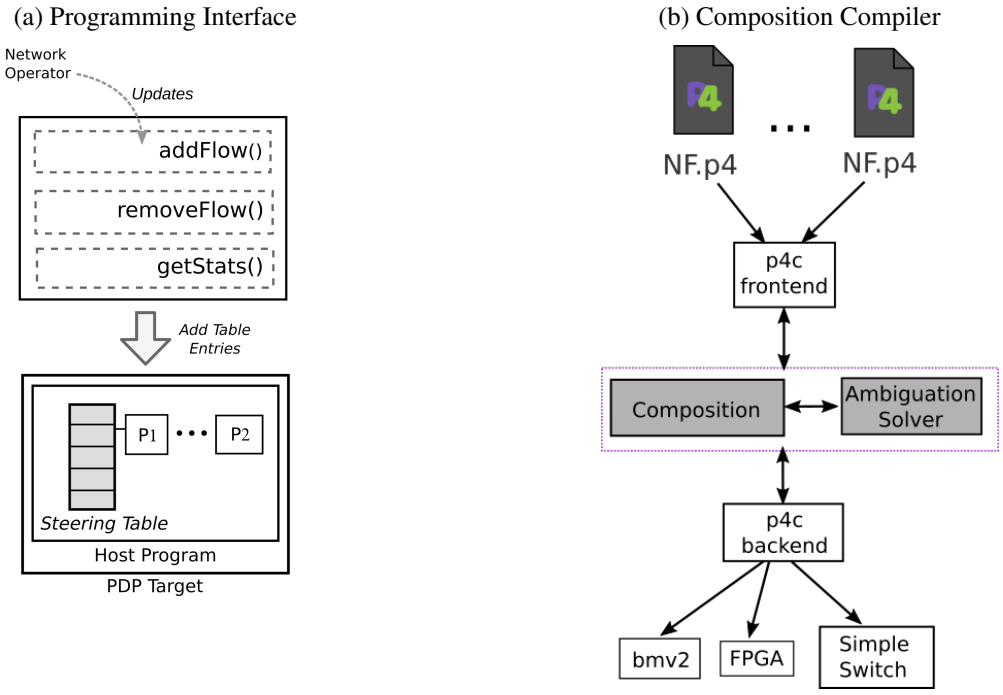
4 IMPLEMENTATION

This chapter summarizes the development details of PRIME. Section 4.1 presents an overview of the main components of PRIME. Next, Section 4.2 presents details of the programming API and the composition compiler. Finally, Section 4.3 shows how network operators can use the programming API to define the steering of flows.

4.1 Prototype Overview

We have prototyped a system called PRIME that implements the composition and steering abstractions presented earlier. Figure 4.1 illustrates the design details of PRIME. PRIME is composed of two essential modules: The steering interface and a composition compiler.

Figure 4.1: Description of Implementation of PRIME



Source: The Author

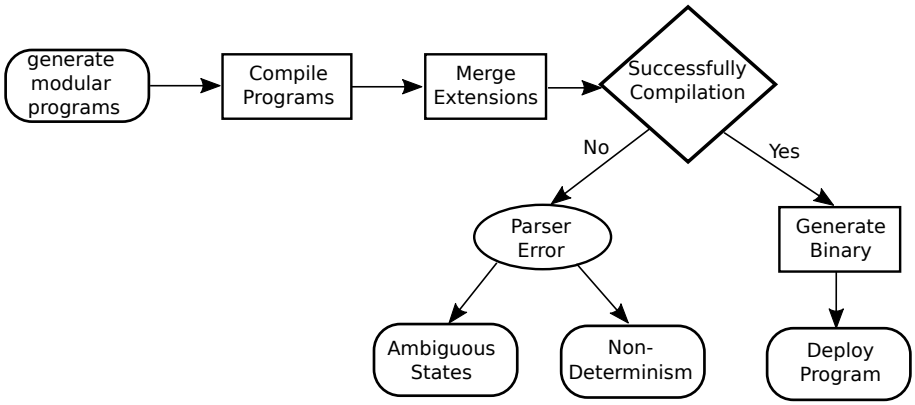
The composition compiler provides the means to assemble large P4 programs by merging smaller ones. We call these programs *"extensions"* and the merged program the *"host"* program. The host program is a P4 program that has an additional table, control blocks, and metadata control, which works as a base to compose extensions. The network operator can place network functions into the host program as a programming model.

The host program has slots for functions and their primitives to steer packets through the functions internally.

4.2 Composition Compiler

We implemented a prototype of the composition compiler to support the development of programs written in $P4_{16}$ and using the V1 Switch model. The compiler works as an extension for the p4c compiler, managing the source code between the p4c front-end to verify the properties between programs. The system parses and composes the original code to the P4 source code. Therefore it maintains compatibility with any target switch. In the end, we build the new source with the `p4c` compiler to obtain the specific target code. Thus, the output code is compatible with any programmable target with support to the language.

Figure 4.2: Compilation and Program Building Methodology



Source: The Author

Figure 4.2 represents the program compiling flow chart of PRIME. First, developers build independent programs and compile them with the standard compiler. Next, programs are linked through the algorithms presented earlier. In the case of the parser, the composition has no success, and it can raise two different instances of errors: a parser error can indicate repeated states with different structures or non-determinism.

4.3 Steering API

After the deployment, the operator can utilize the steering interface to specify the steering of specific subsets of traffic through sequences of program modules during run-

time. All insertion happens in a steering table that performs the first lookup of the traffic. To avoid misrouting during updates of the steering configuration, we provide the means to avoid intermediary states and show why they suffice for a correct implementation. These properties make PRIME valuable for supporting bi-directional configurations.

The host program contains several lines of P4 code and supports $P4_{16}$ programs written using the V1 Switch Model. The integration of programs inserts functions inside the slots of the host program and can add more slots as we compose modules. PRIME uses the standard p4runtime to provided by the P4 language consortiumm to create the programming API after we compose programs. The programming API interacts with the p4-bm target and enables control plane actions over the steering table through a python-based language.

To specify sequences of functions in each switch, PRIME offers a simple API with python-based operators. The API is based on P4runtime (P4RUNTIME, 2019) and enables us to determine which sequence of functions will process specific packets. Instead of match patterns, PRIME allows programs to write basic instructions using the proper name of the function (FOSTER et al., 2011). PRIME also includes a standard switch that specifies the rule's current physical location in the network. Finally, PRIME programmers are free to define their function fields as modules are composed. For example, a programmer may want to assign a packet to one of several functions through the network switches.

The composition treats the output of one function as the input of another. Consider a simple example:

Figure 4.3: Example flow composition

```
1   writeSteering(p4info_helper, sw_id=s1,
     dst_ip_addr="10.0.1.3", nf1=firewall, nf2=
     loadbalancer, ttl_rounds=2)
```

Source: The Author

In Figure 4.3, switch S1 will filter packets with destination 10.0.1.3 and apply the firewall and the load balancer in sequence. To this end, during the composition, the system reads configuration files with the function ID and respective names and translates to low-level rules of the S1 device.

These parameters are wrapped into gRPC requests by the P4runtime API and sent for the target switch in question. Such parameters are translated to the catalog of functions, as presented in Figure 4.4.

Figure 4.4: Example catalog written in P4

```
1     action catalogue( bit<8> nf1, bit<8> nf2, ...,
2     bit<8> nfn, bit<32> ttl_rounds) {
3         meta.custom_metadata.nf_01_id  =   nf1;
4         meta.custom_metadata.nf_02_id  =   nf2;
5         .
6         .
7         .
8         meta.custom_metadata.nf_02_id  =   nf8;
9         meta.custom_metadata.total_rounds =
    ttl_rounds;
10    }
```

Source: The Author

The figure presents the catalog parameters previously passed through P4runtime API (Lines 1-2). Next, these parameters are copied into custom metadata, which will be the steering configuration of each packet (Lines 3-9). Finally, the `ttl_rounds` is also copied to custom metadata and works as an upper bound for the number of NFs that a packet will be processed.

Figure 4.5: Example ingress host program written in P4

```
1  if(meta.custom_metadata.rounds > 0) {
2      //restore standard metadata
3      standard_metadata.egress_spec = meta.port_aux;
4  }
5  if (meta.custom_metadata.rounds == 0){
6      steering.apply();
7      meta.custom_metadata.next_function = meta.
     custom_metadata.nf_01_id;
8  }
9  //next function has ID=1?
10 if (meta.custom_metadata.next_function == 1){
11     //Function_1 code
12 }
13 if (meta.custom_metadata.next_function == 2){
14     //Function_2 code
15 }
16 .
17 .
18 .
19 if (meta.custom_metadata.next_function == N){
20     //Function_N code
21 }
```

Source: The Author

These metadata are used to define the processing order of composed functions.

Figure 4.5 presents how we arrange functions sequentially on the host program. Firstly, we ensure that standard metadata are restored before processing anything (Lines 1-4). Next, we check if we are in the first round. In this case, we load the steering parameters by performing a lookup on the steering table and setting up the first function to process the packet (Lines 6-9). Finally, we may choose between composed functions, which are those that will process the packet in the current round (Lines 10 - 21).

5 EXPERIMENTAL SETUP

This chapter summarizes the experimental evaluation scenarios of PRIME. Section 5.1 presents different use cases using PRIME. Section 5.2 presents metrics and the evaluation methodology. Section 5.3 shows the results of experiments with a software switch to show the impact of using PRIME to compose in-network functions. Finally, Section 5.4 compares PRIME with P4Visor (ZHENG; BENSON; HU, 2018) through simple compositions and comparing the host programs of both systems.

5.1 Use Cases

To validate the feasibility of PRIME, we composed existing P4 applications with a simple L2 switch program that corresponds to our Host program. The code from use cases and evaluation scenarios are available at github [1]. Next, we discuss the details of these existing applications and the final configuration of the compositions we performed.

FlowStalker. FlowStalker (CASTANHEIRA; PARIZOTTO; SCHAEFFER-FILHO, 2019) is a monitoring mechanism which encodes metrics and stores them on data plane devices. Specifically, FlowStalker monitors per-flow and per-packet metrics (*e.g.,* byte counts, packet counts, timestamps) defined by the operator. FlowStalker employs a hash table of registers to index information for the exact flow or packet. A reactive system detects if specific flows violate local thresholds and raises a warning in the case that the threshold is crossed. Thresholds are implemented as a Heavy Hitter Detection mechanism (SIVARAMAN et al., 2017), which has a pipeline of registers indexed by a 5-tuple that represents the flow. After the warning is sent, the controller can inject courier packets to collect data from data plane registers.

In-Band Network Telemetry (INT). INT (KIM et al., 2015) is a framework that allows the collection and reporting of network state by the data plane, without requiring intervention from the control plane. INT is being utilized as a tool for several security mechanisms to troubleshoot, perform congestion control, or even notify the control applications about traffic anomalies. INT extends the packet parser with a new header, which encapsulates monitored items (*e.g.,* timestamps, buffer times, and switch identification). Monitored items are appended into a new header, which is unique for each switch. This means that a new header is instantiated and emitted by all switches in the path to an end

[1] https://github.com/PRIMEb4/Prime

host. The last hop removes INT headers and sends the standard packet to the end host.

LetFlow. LetFlow (VANINI et al., 2017) is a load balancer that executes on switches. Letflow picks paths at random for each flowlet and balances traffic on different paths of the network. A flowlet is a burst of packets that is separated in time from other bursts by a sufficient gap (timeout). When a packet arrives, LetFlow uses a table to map flowlets to paths. Each table entry contains two fields: the last seen time and a path id. When a packet arrives, the program computes a hash (CRC-16) of the source IP, destination IP, source port and destination port. This hash is used as the key to the flowlet table. If the packet is part of an already existing flowlet, the packet is sent on the path identified by the path id, and last seen time is set to the current time. Otherwise, the packet begins a new flowlet and may be assigned to a new path at random.

P4Xos. P4xos (DANG et al., 2016) is a consensus protocol that runs on the data plane. P4Xos is divided into three different P4 programs: the coordinator (leader), the acceptor and the learner. The coordinator ensures only one process sends messages to instances of the protocol, guaranteeing message ordering: it writes the current instance number and an initial round number into the message header; increments the instance number for the next invocation; stores the value of the new instance number; and broadcasts the packet to acceptors. Acceptors choose a value (vote) for each instance of the consensus before forwarding them. Acceptors keep a history of votes to ensure they do not vote for the same value on the same instance of consensus. Finally, learners require a quorum of messages from acceptors and "deliver" a value.

We compose these programs incrementally to analyze the impact of each composition independently. **Scenario 1** is a composition of FlowStalker with the host program. The idea is to build a switch with support to the analysis of security threats using the metrics collected by FlowStalker.

Scenario 2 composes scenario 1 with the In-Band Network Telemetry (INT). This composition allows debugging the network state (*e.g.,* identifying the source of bugs in the network). **Scenario 3** merges LetFlow to Scenario 2. The idea to compose LetFlow is to allow flows to be balanced, mainly when the network performance is low.

Finally, we build scenarios 4, 5 and 6 by composing Scenario 3 with P4Xos. In particular, **Scenario 4** is a composition with the acceptor; **Scenario 5** is a composition with the coordinator; and **Scenario 6** is a composition with the learner. The composition of P4xos raised an error during compilation, because it uses different names for packet instances. We renamed header instances of P4xos to the composition be correct.

Table 5.1: Code metrics of the use cases

	Sce. 1	**Sce. 2**	**Sce. 3**	**Sce. 4**	**Sce. 5**	**Sce. 6**
LoC	366	489	617	779	759	649
States	3	5	5	7	7	7
Tables	4	6	12	15	15	16

Source: The Author

Table 5.1 presents the respective number of states on the packet parser, tables and lines of code (LOC) of the composed programs in each respective scenario. As PRIME merges equivalent states between different programs, the composition tries to minimize the number of states and lines of code. Next, we discuss in details a run-time analysis of each scenario.

5.2 Metrics Formulation

Evaluating programmable switches requires new methodologies. Commonly, end-to-end measurement tools are used, such as `iperf` and `ping`. However, this considers information which is not useful to assess PRIME, as the system operates exclusively on the processing pipeline of the switch. Therefore we follow a different methodology (DANG et al., 2017) to measure latency. We need to assess only the latency of the control flows, as the time spent on parsing and deparsing is not important in this case.

When a packet enters the pipeline and matches the steering table, we store a local timestamp, denoted as $Timestamp_i$. $Timestamp_i$ is stored as part of a packet state until the last program of the pipeline is concluded. On the last stage of the pipeline, $Timestamp_e$ is stored into a local register for further analysis. These correspond to ingress time and egress time, respectively.

Throughput represents the amount of data the switch can process in a given time. Similarly to what we did with the latency measurements, we do not consider parsing and deparsing time on the calculation of throughput. Thus, throughput is calculated as the effective throughput of processing a program's control flows $T_{process}$ less the time spent on the parser T_{parser}. Removing T_{parser} is justified because it is not overhead of the additional tables or the replacement actions. Therefore we define the number of packets traversing the switch as n, and model the per-packet throughput, $P_{throughput}$, as follows:

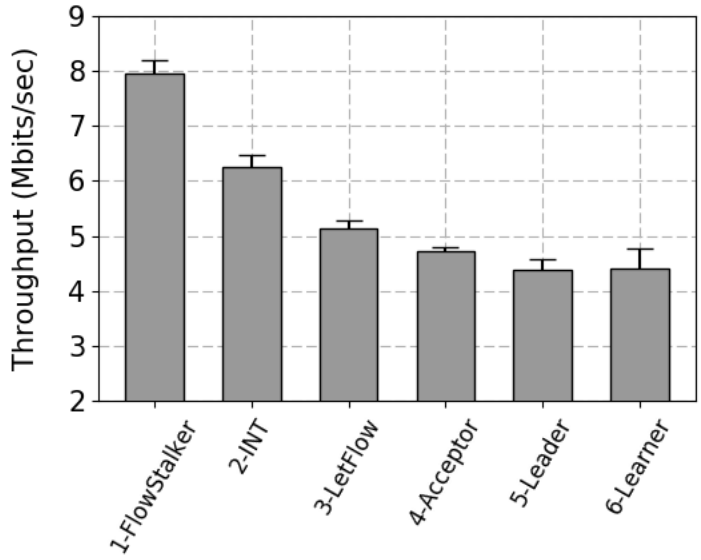
$$P_{throughput} = PacketSize \times (T_{process} - T_{parser})/n \tag{5.1}$$

All experiments below are ideally based on a single source and destination flow traversing the P4 switch. However, multiple flows with different packet sizes would traverse the switch and consequently impact throughput. Therefore we need a more in-depth monitoring mechanism to store metrics from multiple flows. We see the development of a monitoring mechanism like this and more detailed evaluation as future work.

5.3 Assessing Compositions

To evaluate PRIME we need to assess the imposed penalty of steering packets using the final program. For this, we executed each specific scenario using the behavioral model in an Intel(R) Core(TM) i3-6006U CPU 2.00GHz. We performed a thousand requests, and collected packet timestamps to measure latency and throughput following the methodology presented earlier.

Figure 5.1: Throughput

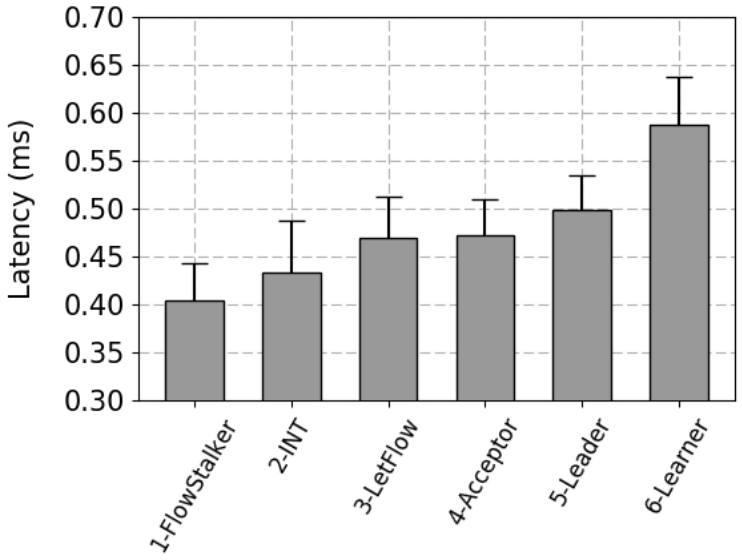


Source: The Author

Figure 5.1 presents the throughput that the composition achieves in the data plane. When the switch steers packets through scenario 1 (*i.e.*, packets match the steering table), throughput is nearly 8 Mbits/sec. The throughput reduces as we compose more modules in scenarios 2-6. The reduction occurs because of the amount of computation required by the functions.

Figure 5.2 presents the latency in ms with only one rule installed on the steering table (*i.e.*, the rule that matches the end host). As we compose more program modules, latency increases. This happens, similarly to what happens to throughput reduction, because

Figure 5.2: Latency



Source: The Author

of the insertion of additional states to the parser. Behavior model runs on general-purpose hardware, and additional states increase CPU consumption. The latency in scenario 1 is nearly 0.4ms. As we compose more program modules, such as in scenario 5, latency increases to nearly 0.5ms. We see as future work deploying PRIME compositions into high-performance packet-processing ASIC and FPGA.

5.4 Comparison with State-of-the-art

We compare PRIME with one of the state-of-the-art approaches, P4Visor (ZHENG; BENSON; HU, 2018), to compose programs. P4Visor is a system to compose P4 programs. The system provides testing operators which compose two different versions of a program using source code merging.

5.4.1 Code Metrics

Specifically, we utilized the Differential testing Operator of P4Visor to compose programs. We could not build the case studies we presented earlier because P4Visor does not currently support the composition of more than two programs. Thus we show two simple scenarios: a production version of a router with a testing version of the same program, and LetFlow with the simple router program.

Table 5.2: Resource Overhead of PRIME and P4Visor

	PRIME		**P4Visor**	
	Router	LetFlow	Router	LetFlow
Parser States	3	4	5	6
Tables	7	10	12	13

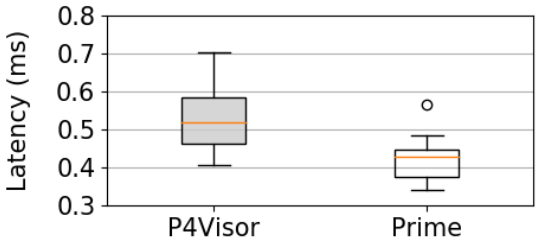
Source: The Author

Table 5.2 presents the number of states of the parser and tables of the programs. The composition of the simple router creates fewer states using the PRIME approach. P4Visor does not merge IPv4 states, therefore creating two different states for equivalent header instances. The number of tables in PRIME is also smaller for these compositions. Although PRIME does not support abstractions to merge tables between programs, the traffic steering control has only one table. Other features to steer packets are performed only by interacting on the catalog and *if-else* statements, without the need of more tables.

5.4.2 Steering Performance

Similarly to PRIME, P4Visor composes programs into a P4 base program which has control structures to steer packets internally. In this section, we show how both host programs impact on the latency of packets. Once every composition will be merged to the host, the latency of the host will always sum to the latency of compositions. To compare P4Visor with our program, we had to translate the P4Visor base program to P4v16. The translation was required to support the same measurement methodology (presented earlier on Section 5.2) to both systems, and perform a more reliable measurement.

Figure 5.3: P4Visor vs PRIME: Latency

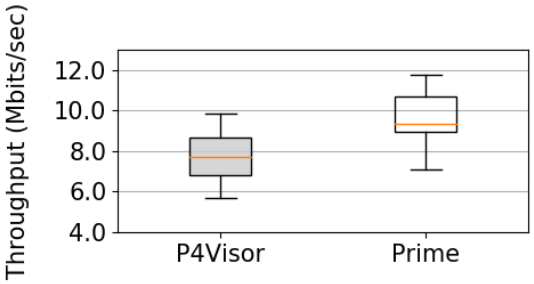


Source: The Author

We performed an experiment that traversed a thousand packets through the programs with no table entries, i.e., we only assessed the standard host program forwarding structure during the experiment. Figure 5.3 presents the latency of the base P4Visor base

program and compared it with PRIME. P4Visor takes about 0.55ms to traverse one single packet through the entire host program, while PRIME takes only about 0.45ms. This can be explained because PRIME requires fewer lookup operations than P4Visor. Achieving less latency in the host program gives evidence that composing programs using the PRIME host program has less penalty on latency than using P4Visor.

Figure 5.4: P4Visor vs PRIME: Throughput



Source: The Author

Additionally, Figure 5.4 presents the throughput of both base programs. P4Visor takes about 8 Mbits/sec, while PRIME takes about 12Mbit/sec. This gives some evidence that PRIME would attend flow demands faster.

Although we use only one table for internal packet addressing and handling, we are still dependent on the switch architecture. The software switch used in the evaluations has the table lookup with time corresponding to the number of rules. Therefore, when the steering table has multiple flows, the overall switch latency will be reduced proportionally. Therefore, we recommend operators to keep a small number of rules and, where possible, split the network load on more than one device so that the network has small delays.

6 RELATED WORK

PRIME is closely related to efforts on data plane composition. Section 6.1 reviews the main research efforts in this field. Next, Section 6.2 discuss the differences of PRIME to these efforts using a taxonomy of properties covered by each work.

6.1 Data Plane Composition

Hyper4 (HANCOCK; MERWE, 2016) is a hypervisor for programmable data planes. It provides a virtualization layer that runs in software. This virtualization layer runs inside the target switch and executes several instances of P4 programs. Although Hyper4 enables modularization, the system imposes high overhead on the forwarding of packets. The overhead occurs because composed modules are running on partially-virtualized programs and because the Hyper4 base program includes several additional tables to support composition. Conversely, PRIME runs as a single P4 program, thus avoiding the virtualization-layer and utilizes only one additional forwarding table to compose modules.

MPVisor (ZHANG et al., 2017) is a hypervisor that uses P4 but provides a base program much smaller than Hyper4. The system offers high-level operators for programming P4 targets. However, their operators produce large pipelines of programs and are not sufficient for the correct operation of steering. This hinders the deployment of configurations that support multiple steering configurations. MPVisor also reduces the number of tables required to virtualize P4 programs when compared to Hyper4, but the number is still large compared with PRIME, which uses only one additional table.

ClickP4 (ZHOU; BI, 2017) is a programming architecture for P4 programs. ClickP4 enables program decomposition into modules and improve code reuse. The system orchestrates modules dynamically, granting more flexibility for the data plane. To this end, ClickP4 uses recirculation to steer packets through programs. Although ClickP4 enables dynamic orchestration of programs, the system may create intermediary states and compromise the forwarding during updates.

P4Bricks (SONI; TURLETTI; DABBOUS, 2018) is a system for multi-processing P4 programs. The system provides parallel and sequential operators, but exclusively to allow nodes to process packets at the same time. The system compiles programs to the same target and restructures the logical pipeline according to control flow dependencies.

P4Bricks provides a low-level compilation for the target switch, which makes the system target specific and limits the utilization of the system. Although the operators proposed by P4Bricks may enable multi-processing, P4Bricks performs out-of-order readings and writes while processing packets, which can create inconsistencies and compromise the developer logic.

P4Visor (ZHENG; BENSON; HU, 2018) is a system to merge and test P4 programs. The system provides AB and Differential testing operators, which both isolate testing traffic from the composed programs. The traffic isolation requires parsers from different program modules to have disambiguation states even if the merged states are equivalent. The merging of control flows tries to minimize resource sharing between modules by merging equivalent tables. The parser composition does not merge transitions from equivalent states, and thus their approach creates a new state for disambiguation. In contrast, PRIME merges equivalent states by uniting their transitions. This feature consequently reduces the number of states necessary to parse the composed program.

Dejavu (WU et al., 2019) is a programming model to optimize resource utilization of programmable switches. The system connects and hosts several functions in a single switch. The system provides parallel and sequential operators, allowing different functions to share the same pipeline. The system leverages recirculation to route packets between chains of functions and tries to minimize the number of recirculations. However, although allowing optimization of the number of recirculation, this can perform out-of-order processing, once functions usually are composed of ingress and egress capabilities. PRIME takes a more intuitive approach, which ensures the correct ordering of read/writes between the functions.

6.2 Discussions

Among the related work, Hyper4 was the pioneer. However, the system has several limitations, mainly because of the usage of an excessive number of tables. Despite the table configuration enables the placement of functions dynamically, such configuration creates a dependency on the hardware which must have an interface do place entire programs dynamically. Such an interface is not common for all target programmable forwarding devices. Such dynamic composition is important for the network, but can be performed in different ways, such as migrating functions (LUO; YU; VANBEVER, 2017) and rerouting traffic (KRUDE et al., 2019).

Table 6.1: Scope of operation and characteristics of recent work

Work/ Goals	Multiple Operators	Dynamic Compositions	Parser Merging	Hardware Independence (Partial)	Consistent Composition
Hyper4		•			
P4Bricks	•		•		
P4Visor	•		•		
MPVisor		•			
ClickP4				•	
Dejavu	•		•	•	
PRIME			•	•	•

Source: The Author

Other approaches, such as MPVisor and ClickP4, tried to mitigate these limitations using a reduced number of tables compared to Hyper4. However, they still lack for merging optimizations, such as the parser composition. Moreover, the complexity of tables and parser states is still considerably high, once they choose to compile new NFs without data plane interruption by using an approach similar to Hyper4. While Hyper4 uses almost 400 tables to declare a program with 8 stages, MPvisor saves 5x to 8x. ClickP4, in turn, requires developers to know ClickP4 code before they deploy a new function, and additional source code is still required to deploy new modules.

Efforts such as P4Visor and P4Bricks introduced host programs with static numbers of tables. Both systems provide multiple operators with different semantics to compose programs. P4Bricks yet enables parallel execution of P4 programs but requires specialized targets, which are not conventional P4 devices. P4Visor breaks barriers by introducing optimization techniques to reduce the resource consumption of control flows and still preserve program isolation. Despite such techniques to optimize the number of tables between modules (or functions) help reducing resource consumption, P4visor uses a fixed amount of 8 tables. Conversely, PRIME uses just one table. P4Visor has limitations on packet parsing because they always make copies of identical states. PRIME covers such parser limitations by enabling the modular composition of parsers. Besides that, current efforts support a limited number of functions. P4Visor, for instance, supports only two compositions and requires modifications on the traffic control to allow more functions to be composed.

More recently, Dejavu suggests the union of equivalent parser states. The union is performed manually, creating limitations for the developer. To this end, PRIME allows that the parser union to be made automatically. Once recirculating packets can generate a

higher overhead on packet processing, Dejavu advocated that programs could divide the same ingress or egress by using sequential and parallel operators. This optimizes some recirculations and, consequently, can allow a higher throughput rate. The negative side of this strategy is that allowing the dynamic changes in the execution order would require multiple copies of the same source code and consequently have high resource consumption. Therefore, the placement of multiple functions at the same control block requires a previous analysis to avoid such a phenomenon. As PRIME composes programs sequentially, as presented in Section 3.3, avoiding such analysis, and ensuring that ordering is preserved.

Further, beyond these limitations of the composition, current research has several limitations for the operation of the network. None of the work allows the consistency of the inner steering updates, which can cause misrouting due to undesired intermediary configurations formed during the update. Table 6.1 shows the properties covered by previous work.

7 CONCLUSIONS & FUTURE WORK

Software paradigms for networks promise to transform the network architecture by softwarizing it, which can simplify network operations and simplify the development life cycle of new solutions. SDN decouples the control logic from forwarding devices, enabling simplified management of the network. OpenFlow is the standard communication protocol for SDN switches, allowing a standard way to program switches through the control plane. However, OpenFlow still has fixed packet headers and is limited to simple forwarding logic. Programmable Data Planes (PDP) enable more flexibility for the operation of networks. With PDP, we can define processing blocks that modify the structure of header contents. Such flexibilization enables us to rethink the design of forwarding devices, placing new functionalities inside the infrastructure.

7.1 Summary and Contributions

To fully reap the benefits of programmability, it should be feasible to compose and operate multiple PDP functions into a single target switch as needed. However, existing techniques are not suitable in the sense that they use an excessive number of parser states and tables, and lack abstractions for the steering of packets through the control flows. As such, they do not support the modular composition of PDP functions. In this thesis, we presented the design and evaluation of PRIME, a composition mechanism to help the modular development and management of P4 programs. PRIME provides a base p4 program that is capable of hosting several functions. The deployment is made by composing packet parsers using a P4 programming operator and by carrying out the placement of functions on slots of the host program using source code merging. The usage of source code merging is required both to allow functions to be implemented independently and do not require developers to know how the base program works.

Furthermore, PRIME introduces the steering of packets through program modules. The steering uses P4 metadata and additional recirculations to ensure that updating the steering of packets is made consistent (i.e., without allowing the creation of intermediary states during updates).

Although composing multiple functions may promote better usage of network resources, the management becomes more complex and error-prone. Current efforts to compose various programs in a single target switch make use of an excessive number of flow

tables and parser states. Consequently, these techniques can severely limit throughput and increase latency in general-purpose hardware or do not fit in specialized hardware, such as netFPGAs or ASICs. To overcome these limitations, we reduce the number of tables the host program needs to steer packets internally to just one table. We also described how the table steers packets to the internal functions and how the composition avoids intermediary states. State-of-the-art techniques do not suffice to provide transitional consistency between steering configurations. Without transitional consistency, changes in the steering of flows through the program modules can create intermediary states, which may cause misrouting and security holes. PRIME provides techniques to allow new applications to be composed, preserving transitional packet-consistency of traffic steering without degrading the performance of the data plane operation. We provide proof that the merging does not rewrite basic steering control and show that composing new rules into the steering table is always made consistently.

We presented a case study showing the operation of our abstractions for several existing P4 programs. For the case studies, we took existing applications and composed them into a host program using PRIME. Results of simulations in a software switch evidence that the compositions have a moderate yet acceptable impact on delay and throughput. We think that this is acceptable because other tools for composition (such as P4Visor or Hyper4) require more tables, which increases the overhead linearly. The overhead obtained is due to the steering table and the recirculations. Yet, we compare PRIME with P4Visor and find that P4Visor has some drawbacks in the case studies performed. We found through two simple compositions that PRIME uses fewer tables and uses fewer parser states on these use cases. We also present the evaluation of the host programs of PRIME and P4Visor to measure latency. We found that our host program achieves lower latency on the data plane.

7.2 Limitations

PRIME still faces several limitations. In particular, inserting overlapping rules into the table can generate conflicting directions on the switch pipeline. We see as future work the development of a mechanism that filters and solves the overlaps before the insertion. This feature can be designed similarly to what is presented in Hermes (CHEN; BENSON, 2017), combined with CacheP4 (MA et al., 2017), to achieve both low update times and throughput. Another limitation is that PRIME still requires the developer of a module

to be responsible for the correctness of each independent module. There is a need for a previous verification step during network operation to ensure the consistency of each separate module. Furthermore, inner characteristics of modules, such as incrementing TTL were not addressed. This can make one switch decrements TTL counters more than once.

PRIME also requires several constraints for the composition to be correct. Firstly, we do not allow the composition of programs that use recirculations or resubmissions. These primitives rewrite the metadata being traversed and, therefore, can make the state inconsistent. Secondly, we still require developers to know the parser states of different programs and ensure their equivalence manually. Third, we also did not address the composition of checksums, which means that if the program has different checksums, we would require this composition to be manual. Fourthly, we did not do any experiment on real hardware, which make our results far for real scenarios. Finally, we do not allow dynamic compositions, requiring to shut down the device to make a switch composition.

7.3 Future Work and Perspectives

There are several other potential future research directions. In particular, exploring new compilation techniques may allow more efficient use of data plane resources by sharing resources between programs. The development of new operators to identify dependencies between modules and a formal reasoning about the steering correctness are also in perspective. Further, in addition to the local guarantees addressed in the composed P4 program, we aim to investigate global path level guarantees for automatic virtualization of PDP programs (YU et al., 2019), and placement heuristics similar to those used with Virtual Network Functions (ANWER et al., 2015) (CHARIKAR et al., 2018).

After such challenges, we aim to investigate the use cases of functions distributed on the data plane and identify end-to-end constraints to the orchestration of NFs/ICFs. The orchestration model may deploy functions on switches organized as a cluster. To this end, we can leverage checkpoint/restore (SHERRY et al., 2015) techniques to ensure that already deployed functionalities keep working after the deployment of new functionalities. This can be built using a new virtualization layer for the data plane by formalizing the orchestration using optimization techniques. Finally, we also see as future work a full exploration of distributed system replication techniques to handle failures.

REFERENCES

ANWER, B. et al. Programming slick network functions. In: **Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research**. New York, NY, USA: ACM, 2015. (SOSR '15), p. 14:1–14:13. ISBN 978-1-4503-3451-8. Available from Internet: <http://doi.acm.org/10.1145/2774993.2774998>.

BENSON, T. A. In-network compute: Considered armed and dangerous. In: **Proceedings of the Workshop on Hot Topics in Operating Systems**. New York, NY, USA: ACM, 2019. (HotOS '19), p. 216–224. ISBN 978-1-4503-6727-1. Available from Internet: <http://doi.acm.org/10.1145/3317550.3321436>.

BIFULCO, R.; RÉTVÁRI, G. A survey on the programmable data plane: Abstractions architectures and open problems. In: **Proc. IEEE HPSR**. [S.l.: s.n.], 2018. p. 1–7.

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

CASTANHEIRA, L.; PARIZOTTO, R.; SCHAEFFER-FILHO, A. Flowstalker: Comprehensive traffic flow monitoring on the data plane using p4. In: IEEE. **2019 IEEE International Conference on Communications (ICC)**. [S.l.], 2019.

CHARIKAR, M. et al. Multi-commodity flow with in-network processing. **arXiv preprint arXiv:1802.09118**, 2018.

CHEN, H.; BENSON, T. Hermes: Providing tight control over high-performance sdn switches. In: **Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: ACM, 2017. (CoNEXT '17), p. 283–295. ISBN 978-1-4503-5422-6. Available from Internet: <http://doi.acm.org/10.1145/3143361.3143391>.

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

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

ERAN, H. et al. NICA: An infrastructure for inline acceleration of network applications. In: **2019 USENIX Annual Technical Conference (USENIX ATC 19)**. Renton, WA: USENIX Association, 2019. p. 345–362. ISBN 978-1-939133-03-8. Available from Internet: <https://www.usenix.org/conference/atc19/presentation/eran>.

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. Available from Internet: <http://doi.acm.org/10.1145/2602204.2602219>.

FOSTER, N. et al. Frenetic: A network programming language. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 46, n. 9, p. 279–291, sep. 2011. ISSN 0362-1340. Available from Internet: <http://doi.acm.org/10.1145/2034574.2034812>.

FREIRE, L. et al. Uncovering bugs in p4 programs with assertion-based verification. In: ACM. **Proceedings of the Symposium on SDN Research**. [S.l.], 2018. p. 4.

GIBB, G. et al. Design principles for packet parsers. In: IEEE. **Architectures for Networking and Communications Systems**. [S.l.], 2013. p. 13–24.

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

HAN, J. H. et al. Blueswitch: Enabling provably consistent configuration of network switches. In: IEEE. **2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)**. [S.l.], 2015. p. 17–27.

HANCOCK, D.; MERWE, J. van der. Hyper4: Using p4 to virtualize the programmable data plane. In: **Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: ACM, 2016. (CoNEXT '16), p. 35–49. ISBN 978-1-4503-4292-6. Available from Internet: <http://doi.acm.org/10.1145/2999572.2999607>.

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: ACM, 2019. (NEAT'19), p. 29–35. ISBN 978-1-4503-6876-6. Available from Internet: <http://doi.acm.org/10.1145/3341558.3342202>.

HELLER, B.; SHERWOOD, R.; MCKEOWN, N. The controller placement problem. In: **Proceedings of the First Workshop on Hot Topics in Software Defined Networks**. New York, NY, USA: ACM, 2012. (HotSDN '12), p. 7–12. ISBN 978-1-4503-1477-0. Available from Internet: <http://doi.acm.org/10.1145/2342441.2342444>.

JAIN, S. et al. B4: Experience with a globally-deployed software defined wan. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 43, n. 4, p. 3–14, aug. 2013. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2534169.2486019>.

JARRAYA, Y.; MADI, T.; DEBBABI, M. A Survey and a Layered Taxonomy of Software-Defined Networking. **IEEE Communications Surveys Tutorials**, v. 16, n. 4, p. 1955–1980, Fourthquarter 2014. ISSN 1553-877X.

JIN, X. et al. Covisor: A compositional hypervisor for software-defined networks. In: **Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2015. (NSDI'15), p. 87–101. ISBN 978-1-931971-218. Available from Internet: <http://dl.acm.org/citation.cfm?id=2789770.2789777>.

JIN, X. et al. Netcache: Balancing key-value stores with fast in-network caching. In: **Proceedings of the 26th Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2017. (SOSP '17), p. 121–136. ISBN 978-1-4503-5085-3. Available from Internet: <http://doi.acm.org/10.1145/3132747.3132764>.

JOSE, L. et al. Compiling packet programs to reconfigurable switches. In: **Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation**. USA: USENIX Association, 2015. (NSDI'15), p. 103–115. ISBN 9781931971218.

KIM, C. et al. In-band network telemetry via programmable dataplanes. In: **ACM SIGCOMM**. [S.l.: s.n.], 2015.

KIM, H.; FEAMSTER, N. Improving network management with software defined networking. **IEEE Communications Magazine**, Citeseer, v. 51, n. 2, p. 114–119, 2013.

KOPONEN, T. et al. Onix: A distributed control platform for large-scale production networks. In: **OSDI**. [S.l.: s.n.], 2010. v. 10, p. 1–6.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. **arXiv preprint arXiv:1406.0440**, 2014.

KRUDE, J. et al. Online reprogrammable multi tenant switches. In: **Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms**. New York, NY, USA: ACM, 2019. (ENCP '19), p. 1–8. ISBN 978-1-4503-7000-4. Available from Internet: <http://doi.acm.org/10.1145/3359993.3366643>.

LI, S. et al. Sr-pvx: A source routing based network virtualization hypervisor to enable pof-fis programmability in vsdns. **IEEE Access**, IEEE, v. 5, p. 7659–7666, 2017.

LI, S. et al. Pvflow: Flow-table virtualization in pof-based vsdn hypervisor (pvx). In: IEEE. **2018 International Conference on Computing, Networking and Communications (ICNC)**. [S.l.], 2018. p. 861–865.

LI, S. et al. Protocol oblivious forwarding (pof): Software-defined networking with enhanced programmability. **IEEE Network**, IEEE, v. 31, n. 2, p. 58–66, 2017.

LIU, J. et al. p4v: Practical verification for programmable data planes. 2018.

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

LUO, S.; YU, H.; VANBEVER, L. Swing state: Consistent updates for stateful and programmable data planes. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: ACM, 2017. (SOSR '17), p. 115–121. ISBN 978-1-4503-4947-5. Available from Internet: <http://doi.acm.org/10.1145/3050220.3050233>.

MA, Z. et al. Cachep4: A behavior-level caching mechanism for p4. In: ACM. **Proceedings of the SIGCOMM Posters and Demos**. [S.l.], 2017. p. 108–110.

MATTOS, D. M. F.; DUARTE, O. C. M. B.; PUJOLLE, G. Reverse update: A consistent policy update scheme for software-defined networking. **IEEE Communications Letters**, IEEE, v. 20, n. 5, p. 886–889, 2016.

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. Available from Internet: <http://doi.acm.org/10.1145/1355734.1355746>.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, ACM, v. 38, n. 2, p. 69–74, 2008.

MIJUMBI, R. et al. Network function virtualization: State-of-the-art and research challenges. **IEEE Communications Surveys & Tutorials**, IEEE, v. 18, n. 1, p. 236–262, 2016.

MONSANTO, C. et al. Composing software defined networks. In: **10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)**. Lombard, IL: USENIX Association, 2013. p. 1–13. ISBN 978-1-931971-00-3. Available from Internet: <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto>.

MUSTARD, C. et al. Jumpgate: In-network processing as a service for data analytics. In: **11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)**. Renton, WA: USENIX Association, 2019. Available from Internet: <https://www.usenix.org/conference/hotcloud19/presentation/mustard>.

P4RUNTIME. [S.l.]: GitHub, 2019. <https://github.com/p4lang/p4runtime>.

PFAFF, B. et al. The design and implementation of open vswitch. In: **12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)**. [S.l.: s.n.], 2015. p. 117–130.

PRAKASH, C. et al. Pga: Using graphs to express and automatically reconcile network policies. In: **Proceedings of the 2015 ACMbloom filters for flow frequency monitoring [19]. Conference on Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2015. (SIGCOMM '15), p. 29–42. ISBN 978-1-4503-3542-3. Available from Internet: <http://doi.acm.org/10.1145/2785956.2787506>.

REITBLATT, M. et al. Abstractions for network update. In: **Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication**. New York, NY, USA: ACM, 2012. (SIGCOMM '12), p. 323–334. ISBN 978-1-4503-1419-0. Available from Internet: <http://doi.acm.org/10.1145/2342356.2342427>.

SAHA, D.; SAMANTA, A.; SARANGI, S. R. Theoretical framework for eliminating redundancy in workflows. In: IEEE. **2009 IEEE International Conference on Services Computing**. [S.l.], 2009. p. 41–48.

SAQUETTI, M. et al. Hard virtualization of p4-based switches with virtp4. In: **Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGCOMM Posters and Demos '19), p. 80–81. ISBN 9781450368865. Available from Internet: <https://doi.org/10.1145/3342280.3342314>.

SAQUETTI, M. et al. P4vbox: Enabling p4-based switch virtualization. **IEEE Communications Letters**, IEEE, 2019.

SCHWERDFEGER, A. C.; WYK, E. R. V. Verifiable composition of deterministic grammars. **ACM Sigplan Notices**, ACM, v. 44, n. 6, p. 199–210, 2009.

SHERRY, J. et al. Rollback-recovery for middleboxes. In: **Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2015. (SIGCOMM '15), p. 227–240. ISBN 978-1-4503-3542-3. Available from Internet: <http://doi.acm.org/10.1145/2785956.2787501>.

SIVARAMAN, A. et al. Packet transactions: High-level programming for line-rate switches. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. [S.l.: s.n.], 2016. p. 15–28.

SIVARAMAN, V. et al. Heavy-hitter detection entirely in the data plane. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: ACM, 2017. (SOSR '17), p. 164–176. ISBN 978-1-4503-4947-5. Available from Internet: <http://doi.acm.org/10.1145/3050220.3063772>.

SONCHACK, J. et al. Enabling practical software-defined networking security applications with ofx. In: **NDSS**. [S.l.: s.n.], 2016. v. 16, p. 1–15.

SONI, H.; TURLETTI, T.; DABBOUS, W. P4Bricks: Enabling multiprocessing using Linker-based network data plane architecture. Working paper or preprint. 2018. Available from Internet: <https://hal.inria.fr/hal-01632431>.

SUN, C. et al. Nfp: Enabling network function parallelism in nfv. In: ACM. **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. [S.l.], 2017. p. 43–56.

VANINI, E. et al. Let it flow: Resilient asymmetric load balancing with flowlet switching. In: **14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)**. Boston, MA: USENIX Association, 2017. p. 407–420. ISBN 978-1-931971-37-9. Available from Internet: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini>.

WU, D. et al. Accelerated service chaining on a single switch asic. In: **Proceedings of the 18th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: ACM, 2019. (HotNets '19), p. 141–149. ISBN 978-1-4503-7020-2. Available from Internet: <http://doi.acm.org/10.1145/3365609.3365849>.

YU, H. et al. Automatic virtualization of accelerators. In: **Proceedings of the Workshop on Hot Topics in Operating Systems**. New York, NY, USA: ACM, 2019. (HotOS '19), p. 58–65. ISBN 978-1-4503-6727-1. Available from Internet: <http://doi.acm.org/10.1145/3317550.3321423>.

ZHANG, C. et al. Mpvisor: A modular programmable data plane hypervisor. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: ACM, 2017. (SOSR '17), p. 179–180. ISBN 978-1-4503-4947-5. Available from Internet: <http://doi.acm.org/10.1145/3050220.3060600>.

ZHANG, C. et al. Hypervdp: High-performance virtualization of the programmable data plane. **IEEE Journal on Selected Areas in Communications**, IEEE, v. 37, n. 3, p. 556–569, 2019.

ZHENG, P.; BENSON, T.; HU, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In: **Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: ACM, 2018. (CoNEXT '18), p. 98–111. ISBN 978-1-4503-6080-7. Available from Internet: <http://doi.acm.org/10.1145/3281411.3281436>.

ZHOU, Y.; BI, J. Clickp4: Towards modular programming of p4. In: **Proceedings of the SIGCOMM Posters and Demos**. New York, NY, USA: ACM, 2017. (SIGCOMM Posters and Demos '17), p. 100–102. ISBN 978-1-4503-5057-0. Available from Internet: <http://doi.acm.org/10.1145/3123878.3132000>.

APPENDIX A — PUBLISHED PAPER – SBRC 2019

Redes definidas por software (SDN) e o surgimento de planos de dados programáveis permitem maior flexibilidade para a operação de redes. Essas tecnologias são capazes de permitir que os administradores de rede reconfigurem os planos de dados e de controle. A capacidade de reconfigurar e programar a rede sob demanda oferece vários benefícios, em particular possibilitando melhorar os mecanismos de segurança de rede usando a capacidade de programação. No entanto, além de promover um grau maior de flexibilidade, a programação do plano de dados levanta preocupações em relação a erros que podem criar inconsistências na função mais básica da rede, o encaminhamento de dados, interrompendo políticas previamente definidas. Neste trabalho apresentamos um framework para instalar funções em planos de dados programáveis de maneira confiável, garantindo que a instalação de tais funções preserve as propriedades básicas de encaminhando. Para isso, empregamos técnicas de composição de programas para mesclar funções modulares em um único plano de dados agregado, garantindo que o programa resultante seja correto após a mesclagem. Para mostrar a corretude de nosso método, apresentamos um estudo de caso com um firewall e um módulo de monitoramento.

- **Title:** Abordagem de Composição de Programas P4 em Redes Programáveis
- **Conference:** XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos
- **Type:** Main Track (Full Paper)
- **Qualis:** B2
- **Held at:** Gramado-RS, Brazil

Abordagem de Composição de Programas P4 em Redes Programáveis

Ricardo Parizotto, Lucas Castanheira, Alberto Schaeffer-Filho

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{rparizotto,lbcastanheira,alberto}@inf.ufrgs.br

Abstract. *Software Defined Networks (SDN) and emerging programmable data planes enable more flexibility for the operation of networks. Such technologies are capable of allowing network operators to reconfigure networks on both control and data planes dynamically. Such an ability to reconfigure and program the network on demand offers several benefits, in particular making it possible to improve network security mechanisms by using programmability. However, in addition to promoting a higher degree of flexibility, data plane programmability raises concerns with respect to bugs that can create inconsistencies in the network's most basic function, the forwarding of data, disrupting previously defined policies. In this work we present a framework to install functions on programmable data planes in a reliable manner, ensuring that the installation of such functions preserves basic forwarding properties. For this, we employ program composition techniques to merge knowingly correct modular functions into a single, aggregated data plane program, ensuring that the resulting program is correct after the merge. To show the correctness of our method, we present a case study with a firewall and a processing/monitoring module.*

Resumo. *Redes Definidas por Software (SDN) e o surgimento de planos de dados programáveis permitem maior flexibilidade para a operação de redes. Essas tecnologias permitem que os administradores de rede reconfigurem os planos de dados e de controle. A capacidade de reconfigurar e programar a rede sob demanda oferece vários benefícios, em particular possibilitando melhorar os mecanismos de segurança de rede usando a capacidade de programação. No entanto, além de promover um grau maior de flexibilidade, a programação do plano de dados levanta preocupações em relação a erros que podem criar inconsistências na função mais básica da rede, o encaminhamento de dados, interrompendo políticas previamente definidas. Neste trabalho apresentamos um framework para instalar funções em planos de dados programáveis de maneira confiável, garantindo que a instalação de tais funções preserve as propriedades básicas de encaminhamento. Para isso, empregamos técnicas de composição de programas para mesclar funções modulares em um único plano de dados agregado, garantindo que o programa resultante seja correto após a mesclagem. Para mostrar a corretude de nosso método, apresentamos um estudo de caso com um firewall e um módulo de monitoramento.*

1. Introdução

Paradigmas de redes definidas por software (SDN) possibilitam o desacoplamento do hardware (e.g. roteadores) e dos programas que nele executam (e.g. algoritmos de ro-

teamento) [Feamster et al. 2014]. Essa estratégia facilita a operação das redes, uma vez que promove programabilidade no plano de controle da rede. Recentemente, a programabilidade foi estendida também para o plano de dados. Com o intuito de possibilitar a programabilidade no hardware que reside no plano de dados, foi criada a linguagem P4 [Bosshart et al. 2014], que permite aos administradores da rede definirem o comportamento dos dispositivos de encaminhamento. Uma vantagem decorrente disso é a facilidade na criação e na implantação de novos protocolos de rede totalmente customizáveis, sem depender da indústria para que uma nova funcionalidade seja adicionada ao comportamento do plano de dados.

Tal habilidade para reconfigurar e programar a rede possui várias aplicações, que geralmente abrangem mecanismos governados pela dinâmica e pelas mudanças frequentes de políticas de rede. Isso pode envolver, por exemplo, a implantação de funções de rede adicionais e a reescrita da funcionalidade dos *switches* P4 de maneira que possam suportar mais do que apenas um serviço. Para que isso ocorra, é necessário o desenvolvimento de técnicas abrangentes que permitam que a reconfiguração da rede ocorra de maneira rápida e sem corromper propriedades básicas de operação. Alguns trabalhos recentes propõem a utilização de P4 para configurar funções virtualizadas no próprio plano de dados [Hancock and van der Merwe 2016, Zhang et al. 2017]. Tais propostas, porém, pecam de duas maneiras: em escalabilidade, devido ao uso excessivo de tabelas de controle e primitivas de recirculação de pacotes, atrasando o processamento e encaminhamento dos pacotes; e não fornecendo o isolamento necessário para as funções [Dimitropoulos et al. 2018]. Diante disso, entendemos que são necessárias abstrações e estratégias que permitam que os administradores de rede possam implantar novas funcionalidades nos seus dispositivos programáveis, sem que isso impacte negativamente no desempenho das funções de rede.

Neste trabalho, propomos uma estratégia baseada em P4 que permite que um administrador de rede possa estender o comportamento de *switches* programáveis. A nossa estratégia é composta de duas etapas: (1) a composição de programas, que deverá possibilitar que o administrador de rede componha funções modulares a um programa base, de modo que a composição resulte em um novo programa com as funcionalidades tanto do programa base como da extensão; (2) o isolamento lógico dos programas, que evita que as regras de *match+action* sejam sobrepostas e permite que a ordem em que os programas são executados seja alterada dinamicamente. Agregado à estratégia de composição, isso garante que as funções atuem de maneira isolada no processamento dos pacotes. A estratégia proposta depende de um programa base com construtores bem definidos, que permitem que as funções compostas possam ser gerenciadas de maneira dinâmica. O operador de rede poderá, então, compor o programa base com a configuração desejada e, enquanto está em funcionamento, decidir qual a ordem em que as funções serão processadas por um tipo de tráfego específico.

A estrutura desse artigo esta dividida da seguinte maneira: na Seção 2 apresentamos um background sobre programabilidade no plano de dados e as restrições no modelo de encaminhamento. Na Seção 3, apresentamos a estratégia de composição de programas, seguida pela estratégia de agregação de fluxos de controle. Na Seção 4 apresentamos um estudo de caso e avaliação de nossa estratégia. Por fim, apresentamos uma visão geral dos trabalhos relacionados e as conclusões.

2. Background e Motivação

Nesta seção, revisamos P4 e programabilidade no plano de dados, seguido por uma descrição das características principais que devem ser consideradas quando novas funcionalidades são implantadas ao plano de dados. Apresentamos também as principais restrições operacionais do modelo de encaminhamento, que dizem respeito às entradas das tabelas de *match+action* e de *parsers* de cabeçalhos de pacotes.

2.1. Abstração da linguagem P4

P4 é uma linguagem de especificação de plano de dados que permite a configuração e programação de dispositivos de encaminhamento [Garcia et al. 2018, Bosshart et al. 2014]. A sua abstração, apresentada na Figura 1, divide o comportamento do plano de dados em um *parser* de cabeçalhos de pacotes, um conjunto de tabelas de *match+action* e fluxos de controle. O *parser* é uma máquina de estados que descreve como ler os cabeçalhos de um pacote para as variáveis internas. Depois que um pacote chega a um estado final da máquina de estados do parser, o pacote é processado pelos construtores definidos no fluxo de controle. No fluxo de controle são definidas exclusivamente as estruturas das tabelas, ações e a ordem em que elas são executadas durante o processamento dos pacotes.

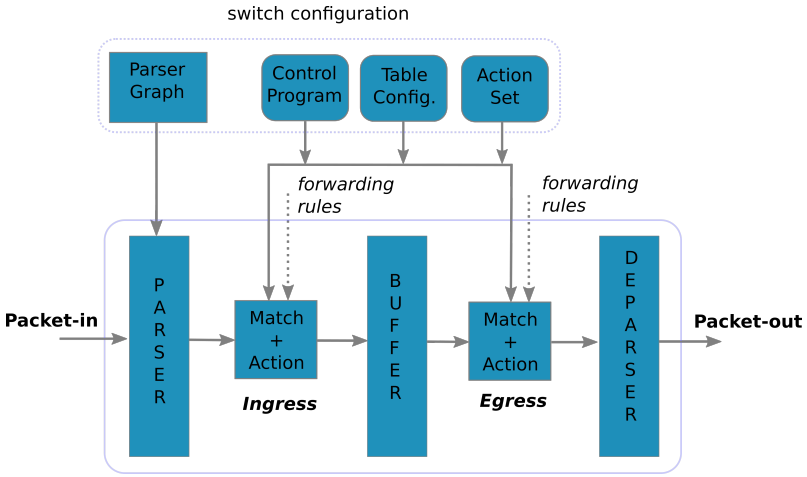


Figura 1. A abstração da linguagem P4, adaptado de Bosshart et al. [Bosshart et al. 2014]

A abstração P4 divide o modelo de encaminhamento em dois estágios sequenciais, (1) configuração do *hardware*, feita de maneira estática no programa P4 e (2) população das regras, feita de maneira dinâmica pelo controlador. Na configuração, temos que escrever o programa P4 que vai rodar no *switch* (incluindo *parser*, estágios do *match+action* e *deparser*). Tal configuração é toda feita dentro de um fonte P4, que é compilado para uma arquitetura especifica e carregado no *switch*. O estágio de população das regras acontece logo após a configuração e se manterá durante todo o *runtime* do *switch*. No estágio de população, o controlador pode alterar as regras do *switch* livremente. Essa fase é realizada pelo controlador e pelos programas que nele executam, cada um atualizando as regras que lhes é pertinente.

2.2. Restrições no modelo do plano de dados

Geralmente, programas são criados para executar no controlador, atualizando regras durante a fase de população. Para que evitemos *bugs* nesses programas, as atualizações de regras por ele geradas têm que respeitar algumas propriedades básicas, tais como: alcançabilidade e boa formação de pacotes. Na sequência, falaremos sobre cada uma delas.

Propriedades fim-a-fim Sempre que uma nova entrada é inserida em uma tabela do modelo do plano de dados, ela deve preservar propriedades (ou políticas) definidas previamente, tais como

"pacotes do switch A devem chegar ao switch B".

Ou propriedades de *safety*, como

"o fluxo i deve ser processado pelo switch X antes que ele alcance seu destino".

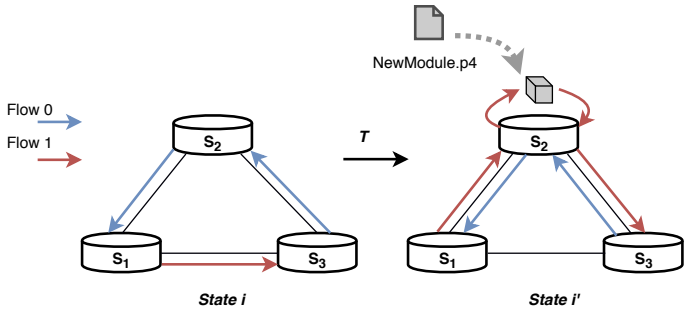


Figura 2. Transição do estado de rede

Mudanças de políticas e de roteamento no plano de dados podem ser modeladas como transições entre estados das tabelas de *match+action*. Na Figura 2, apresentamos um cenário que descreve uma transição entre dois estados de rede. No estado i, o Fluxo 1 é roteado através do caminho $(S1, S3)$. Uma transição T de um estado i para o estado i' é realizada implantando um módulo adicional ao *switch* $S2$ e mudando seu comportamento de encaminhamento para o novo módulo interceptar pacotes do Fluxo 1. Então, depois de atualizar $S1$, o mesmo fluxo é roteado por $(S1, S2, S3)$, respectivamente.

Atualizar o plano de dados não é uma operação atômica, porque *switches* não são dispositivos sincronizados. Por isso, a ordem em que cada *switch* aplica mudanças é um fator importante para alcançar uma transição do estado da rede sem inconsistências. Por exemplo, no cenário da Figura 2, se, durante a transição T o *switch* S_1 atualizar seu comportamento de encaminhamento antes de S_2, os pacotes do fluxo 1 irão enfrentar um 'buraco negro' quando alcançarem S_2 (ou os pacotes serão enfileirados) [Reitblatt et al. 2012][Katta et al. 2013] [Jin et al. 2014] [Nguyen et al. 2017].

Propriedades do switch O nível de inconsistência torna-se ainda maior com programabilidade no plano de dados, que permite que esse tipo de *bug* possa ocorrer dentro do *pipeline* de tabelas do *switch*, devido a possibilidade de mudanças da configuração de

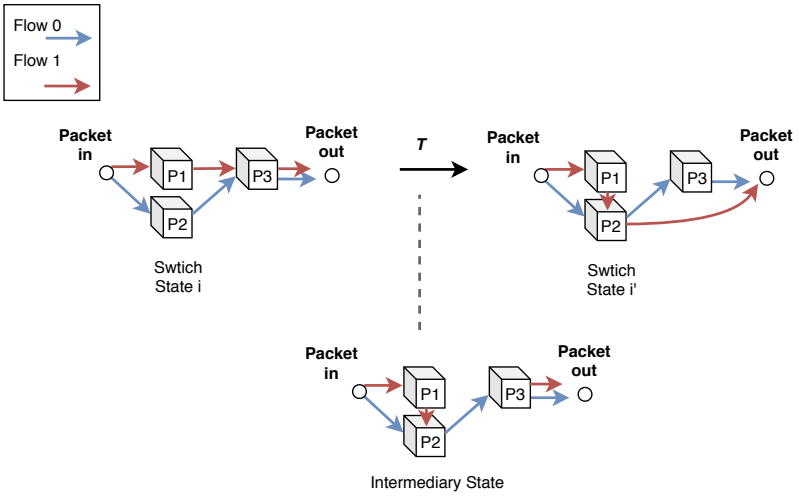


Figura 3. Transição do estado interno de um switch

tabelas e parsers de pacotes [Freire et al. 2018][Liu et al. 2018]. Ocasionalmente, se um novo módulo inserido não possuir as instruções corretas para decodificar os cabeçalhos dos pacotes, os pacotes vão ser processados de uma maneira indesejada. Um exemplo disso, é que pacotes chegariam ao pipeline sem nenhum valor instanciado ou seriam eliminados ainda no parser [Lopes et al. 2016].

A ordem em que as atualizações são gerenciadas dentro do pipeline do switch também estão sujeitas a falhas de configuração. A Figura 3 ilustra uma transição entre dois diferentes estados internos de um switch. No estado i, o Fluxo 1 é roteado pelos módulos (P_1, P_3). Uma transição do estado i ao i' é realizada mudando a sequência de módulos que processam o Fluxo 1. Depois da transição, o mesmo fluxo é roteado pelos módulos (P_1, P_2), respectivamente. Se o módulo P_1 atualizar seu comportamento de encaminhamento antes de P_2, irá formar um estado intermediário inconsistente, onde pacotes do Fluxo 1 irão enfrentar um 'buraco negro' quando chegarem ao módulo P_2.

Neste trabalho, propomos uma estratégia para compor programas P4 em redes programáveis. Para isso, apresentamos um *framework* capaz de unir características de diferentes programas em um único programa agregado, que contempla a funcionalidade de ambos e pode ser gerenciado dinamicamente.

3. Abordagem de Composição de Programas

Recentemente, diversas aplicações de redes voltaram à discussão, o que têm motivado a criação de vários mecanismos para o plano de dados. Porém, a maioria dos trabalhos trata os programas para plano de dados como monolíticos e com uma única funcionalidade específica. Neste trabalho, apresentamos uma estratégia para compor mais do que um programa em um switch P4. Nossa estratégia se utiliza de técnicas de composição de máquinas de estado para mostrar como realizar a extensão de parsers e deparsers de pacotes. Nessa seção, também apresentamos uma arquitetura base, que possui primitivas para composição de fluxos de controle de vários programas.

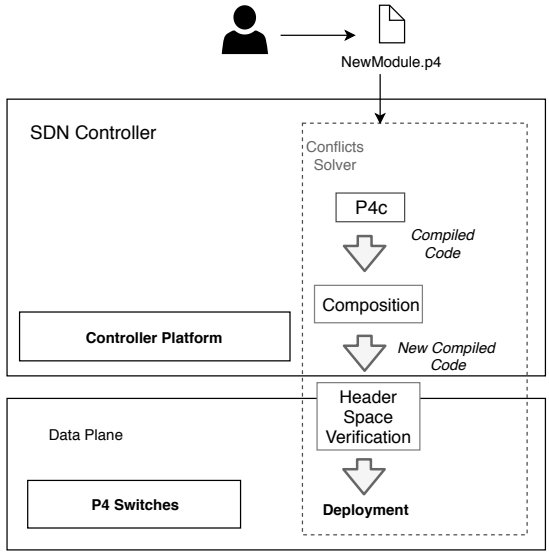


Figura 4. Arquitetura do mecanismo de composição

Visão geral da estratégia Dado um novo programa S, utilizamos uma técnica de composição de máquinas de estado para criar uma nova especificação com um conjunto de extensões do programa S. A composição é a união do conjunto de estados terminais, não-terminais, transições e as definições e instâncias de cabeçalhos definidos em cada programa. A estratégia proposta, ilustrada na Figura 4, compõe a configuração dos programas do plano de dados ainda no plano de controle, pelo administrador de rede. As funções devem ser inseridas conforme as necessidades do operador. Assim que a composição é finalizada, uma etapa de verificação checa o espaço de cabeçalhos para evitar loops e não determismo no código final gerado. Por fim, o resultado é compilado e pode ser implantado no switch. A implantação é realizada de maneira estática, isto é, exige a reinicialização do switch. Depois que a implantação é finalizada, o operador pode gerenciar dinamicamente a ordem em que as funções inseridas processam os pacotes. Isso se dá pela atualização das regras de *match+action* das tabelas que fazem parte da composição.

A composição de programas é realizada estendendo o código *host* (Figura 5, passo 1). *Parsers* e *Deparsers* são unidos utilizando composição de máquinas de estados (Figura 5, passo 2). O novo fluxo de controle é posicionado no começo do *pipeline* do programa *Host*, reescrevendo seu código fonte (Figura 5, passo 3). A seguir, essas etapas serão descritas em maiores detalhes.

3.1. Extensão de parsers de pacotes

Após a leitura do programa *Host*, o seu *parser* de pacotes é estendido para incluir as funcionalidades do *parser* de pacotes pertencente à extensão [Zheng et al. 2018]. O resultado da composição é uma nova máquina de estados que une os estados equivalentes (i.e. estruturas de cabeçalhos) e integra as transições que não estão no parser do programa a ser estendido. A Figura 5, Passo 2, apresenta a composição entre esses dois parsers. No exemplo da figura, o *parser* do programa *Host* é estendido para suportar a leitura do cabeçalho *sec* após decodificar o cabeçalho do IPv4. Depois que a composição de *parser* de pacotes é finalizada, o processo segue para a composição do fluxo de controle (passo 3). Essa estratégia permite que o administrador de rede altere dinamicamente a ordem

das funções processadas no plano de dados, simplesmente adicionando novas regras de *match+action* ao *switch*.

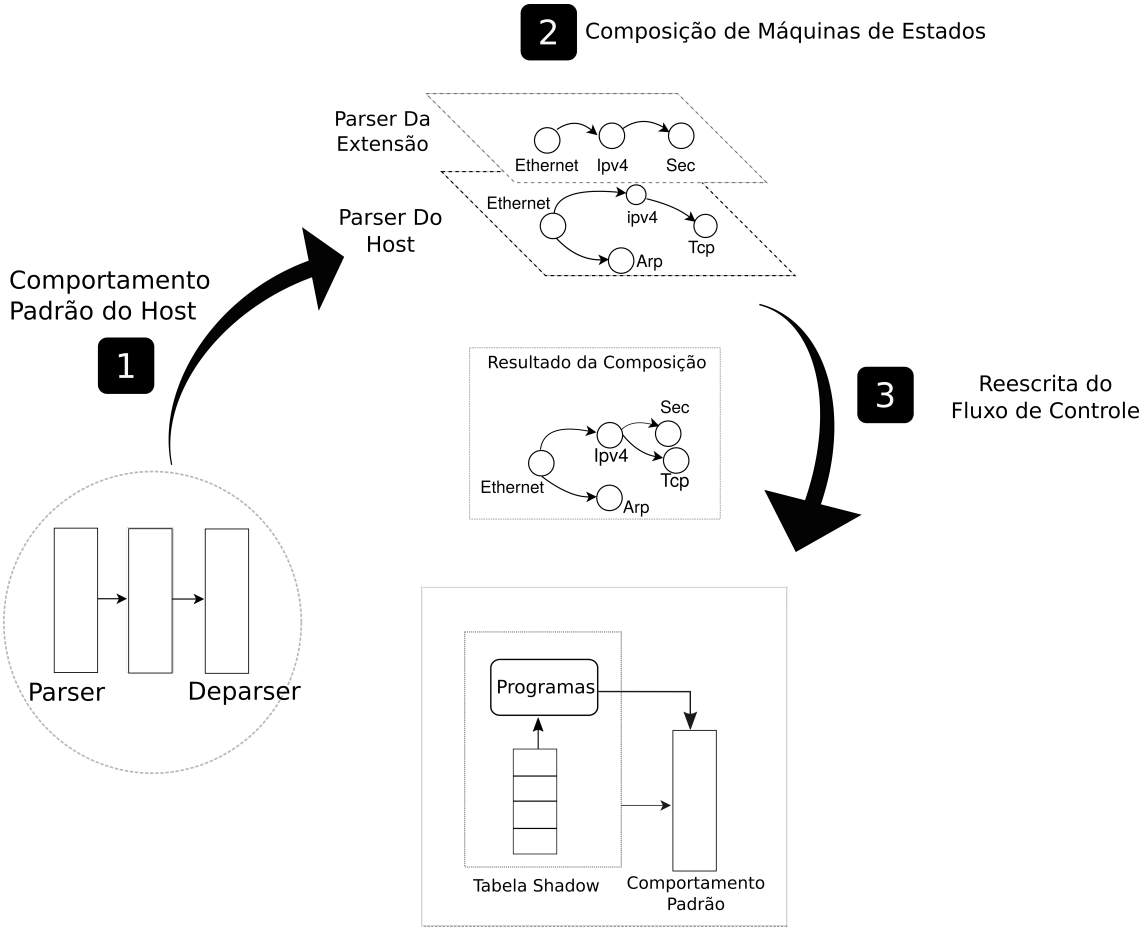


Figura 5. Visão geral da estratégia de composição de programas P4

3.2. Agregação de Fluxos de Controle

A composição de fluxos de controle permite incluir ações adicionais, definições de tabelas ou, até mesmo, ramificações adicionais ao programa *Host*. Uma maneira simples para compor fluxos de controle é unir a especificação de tabelas com o mesmo nome e tipos de atributos de *match*. Porém, isso cria a possibilidade de diferentes aplicações do plano de controle inserirem regras conflitantes, permitindo que, eventualmente, pacotes da mesma política sejam roteados por mais do que um caminho. Como uma consequência disso, propriedades básicas de processamento de pacotes podem ser violadas, como por exemplo, a propriedade de coerência de pacotes (ou de fluxos), apresentada em [Reitblatt et al. 2012]. A adição de um módulo que gera regras conflitantes requer que as aplicações do plano de controle sejam alteradas, o que, tecnicamente atrasaria o processo de desenvolvimento. Nessa seção, apresentamos o mecanismo que compõe os módulos do fluxo de controle, isolando os fluxos de controle específicos de cada aplicação.

Impacto de isolamento Com o objetivo de prevenir o problema de inserção de regras conflitantes, isolamos as tabelas e ações de cada novo módulo inserido. Para isso, é

necessário renomear tabelas e ações que tenham nomes ambíguos (para evitar loops no pipeline). Por isso, resolvemos conflitos de nomes e fixamos o novo módulo no início do pipeline do programa *host*. Essa estratégia isola as regras de *match+action* de cada aplicação, tornando o comportamento do programa *host* independente do processamento das funcionalidades inseridas. O isolamento garante que as regras instaladas não irão corromper as regras específicas das aplicações que gerenciam as outras tabelas do programa, preservando, assim, a ordem de execução das funções.

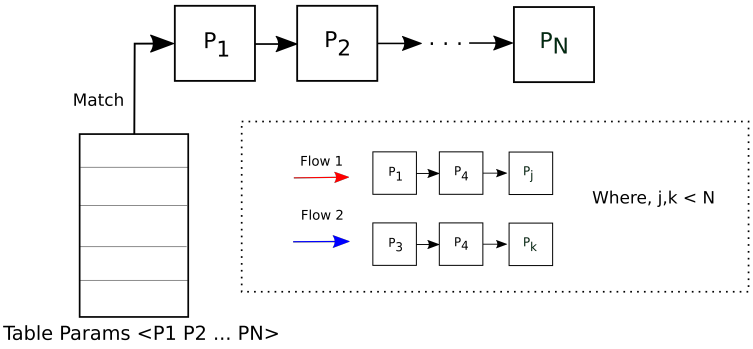


Figura 6. Encadeamento de programas no plano de dados

Encadeamento de Programas Para compor vários programas em nosso *framework*, criamos a abstração de encadeamento de funções. Essas possuem sua ordem de execução controlada a partir de regras adicionadas de uma aplicação do plano de controle. Para isso, utilizamos uma tabela, que chamamos de *'Shadow'*, a qual funciona como um grande catálogo de ponteiros para funções. Inicialmente, a tabela shadow é posicionada no inicio do pipeline de tabelas e intercepta todos os pacotes, mapeando um conjunto de fluxos para um conjunto sequencial de programas $P_1, P_2, ..., P_N$. Dessa maneira, a ordem de execução das funções pode ser diferente para cada parâmetro de *match* da tabela e alterada de maneira dinâmica pelo operador de rede. Quando o operador deseja alterar as funções executadas, ele apenas atualiza o conteúdo da tabela *Shadow*, alterando os parâmetros que dizem respeito à ordem de execução dos programas. A Figura 6 apresenta a estrutura da tabela *Shadow*. No exemplo da figura, Fluxo 1 é mapeado para ser processado apenas por P_1, P_4 e P_j, respectivamente. Enquanto o Fluxo 2 vai ser processado por P_3, P_4 e P_k em caso de *match* na tabela.

3.3. Composição de Deparsers

A composição de *deparser* é um processo mais simples. Porque o próprio *deparser* possui uma estrutura mais simples. A composição do *deparser* se dá apenas pela adição da primitiva que emite os cabeçalhos adicionais da extensão. Isto, é, os cabeçalhos que foram incluídos durante a extensão do *parser*, agora, devem ser emitidos na ordem correta.

4. Estudo de caso e Avaliação

Para validar nossa estratégia, desejamos mostrar o impacto que o mecanismo de composição traz para virtualização de vários programas P4. Para isso, construímos um

cenário de estudo de caso utilizando módulos de monitoramento e de controle de acesso (firewall) no próprio plano de dados. Adicionalmente, medimos o impacto de nossa estratégia de composição no processamento e encaminhamento de pacotes. Isso tudo, conforme a quantidade de funções (ou programas) inseridos aumenta. Como exemplo simples, apresentamos nessa seção a composição de um módulo de monitoramento a um *switch* simples, com encaminhamento de camada 2. Depois, mostramos como compor o firewall a esse mesmo programa.

Módulo de processamento/Monitoramento O módulo de processamento e monitoramento adiciona capacidades de armazenamento ao pipeline do programa. Isso permite que parte do processamento de funções de segurança aconteça no próprio plano de dados. Esse módulo armazena métricas sobre os fluxos (isto é, pacotes identificados pelo endereço IPv4 de destino e origem) e dispara um alerta para o plano de controle quando um determinado limite é atingido. O funcionamento deste módulo é baseado em estratégias de detecção de *Heavy Hitter* [Sivaraman et al. 2017] para identificar os fluxos que ultrapassam o limite. A composição do módulo de monitoramento ao programa *host*, estende os cabeçalhos do programa principal com as definições de cabeçalho IPv4 e suas respectivas definições. O fluxo de controle é composto estendendo as ações da tabela *shadow*, de maneira que elas incluam uma nova ação, a qual executa as operações do programa de monitoramento. Por fim, o *deparser* é estendido pela emissão do conteúdo do cabeçalho IPv4.

Firewall com Estado Propomos um firewall que inspeciona os cabeçalhos de camada 3. Ele funciona provendo uma interface para drop e reescrita de tipos específicos de pacotes conforme eles são interceptados. O firewall armazena o estado de novas conexões TCP no próprio switch (SYN & ACK = 1) e somente permite que a conexão seja emitida quando estabelecida. Ao contrário do que ocorre com alguns firewalls para redes definidas por software [Hu et al. 2014], em nossa proposta não há necessidade de tratar sobreposições de regras de *match+action*. Isso ocorre pois uma nova tabela é criada para o firewall e isolada das tabelas de funcionamento padrão do switch pela estratégia de composição. A composição do firewall incorpora ao *parser* de cabeçalhos o estado referente ao TCP. A definição da ação que reescreve os cabeçalhos é adicionada e o *deparser* começa a produzir cabeçalhos TCP.

4.1. Overhead de desempenho

Para avaliar nosso *framework*, utilizamos o *switch* de software bmv2 [1], em conjunto com o emulador mininet. Nós executamos os experimentos em um Intel(R) Core(TM) i3-6006U CPU @ 2.00GHz. O objetivo é mostrar o impacto da utilização de nossa estratégia para o atraso e a vazão dos fluxos processados pelo programa resultante da composição.

Para avaliar o atraso que a composição de novos programas traz para o comportamento usual do *switch*, nós configuramos um experimento que realiza 100 requisições e medimos o tempo em que uma requisição leva pra ser processada. Comparamos o atraso em um cenário utilizando o programa *host* contendo os módulos enunciados acima composto com o programa sem extensões e utilizamos tabelas 'Shadow' com 1024 entradas. Na Figura 7, identificamos como 'Shadow' o atraso gerado quando os pacotes são interceptados e combinam com alguma regra da tabela *Shadow*. Identificamos na figura

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

como ‘P4’, o atraso gerado quando os fluxos são interceptados pela tabela *Shadow*, mas não combinam com nenhuma regra, consequentemente sendo processados apenas pelas funções usuais do switch (i.e. encaminhamendo de nível 2).

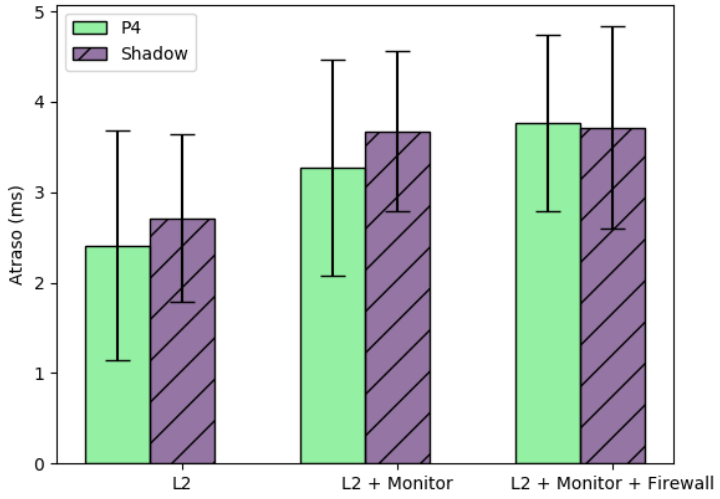


Figura 7. Desempenho e *Overhead* da Estratégia de Composição

Como pode ser visto na Figura 7, a utilização do mecanismo de composição não parece demonstrar sacrifício de performance. A variação de desempenho entre os dois cenários é limitada a uma percentagem relativamente pequena, sem ultrapassar a diferença de mais do que 0.4 ms entre o experimento utilizando a tabela *Shadow* e o experimento com P4 nativo. Isso mostra que a utilização ta tabela *Shadow* para isolar novos programas não degrada significativamente o desempenho das funcionalidades usuais do switch. Vemos como uma possibilidade de trabalho futuro comparar nossa solução com outras abordagens que proponham a composição de funções para plano de dados programáveis.

A Figura 8 apresenta o impacto que as novas funcionalidades trazem para a vazão, em Mbits por segundo. Para cada programa resultante da composição, avaliamos tanto a vazão pelo caminho usual do comportamento do switch (identificado como P4); e a vazão quando os fluxos são encaminhados e processados pelos módulos estendidos. É possível observar que a vazão reduz conforme os módulos que leem contéudos de cabeçalhos de camada mais alta são adicionados. A composição do *firewall* agregado ao módulo de monitoramento reduz pela metade a vazão quando é interceptado pela tabela *Shadow* (de 22Mbit/s para cerca de 12Mbit/s). Isso se deve tanto pela necessidade de decodificar mais bytes do cabeçalho, tanto pelo tempo de processando dos bytes no fluxo de controle. De qualquer maneira, é um preço aceitável a se pagar quando se deseja uma rede mais segura.

5. Trabalhos Relacionados

Programabilidade no plano de dados tem sido tipicamente empregada na virtualizacão de serviços que tradicionalmente eram engessados a *middleboxes* fechados ou ao circuito integrado dos *switches*. Nessa seção, apresentamos estudos já desenvolvidos sobre virtualização de funções do plano de dados e sobre estratégias de segurança e monitora-

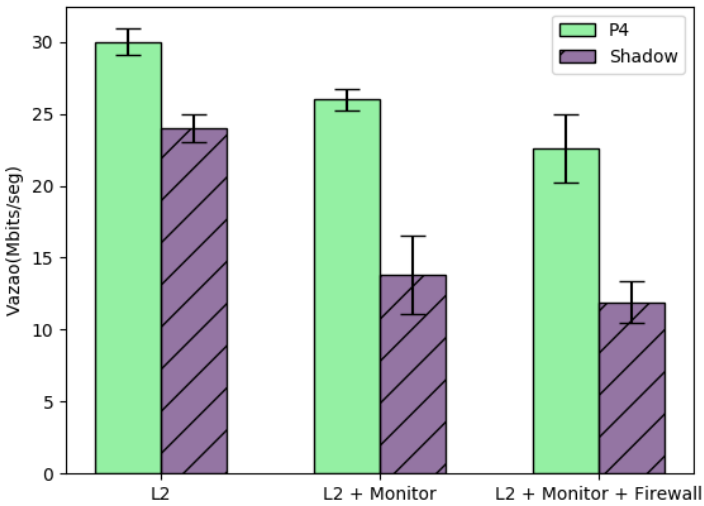


Figura 8. Impacto da composição dos módulos na vazão dos fluxos no plano de dados

mento. O esclarecimento sobre o que já foi produzido sobre o assunto ajudará a compreender a contribuição de nossa proposta para essas áreas.

Virtualização de Plano de Dados Em [Hancock and van der Merwe 2016], os autores propõem o Hyper4, um hypervisor para programas P4, cujo design permite a virtualização de vários programas P4. Dessa maneira, o Hyper4 possibilita que o administrador da rede altere dinamicamente a ordem lógica dos programas. Para isso, todavia, faz-se necessário um conjunto amplo de tabelas e primitivas de recirculação que permitam a execução de vários parsers. Em [Zhou and Bi 2017], os autores utilizam um número reduzido de tabelas, mas ainda exigem que os pacotes recirculem para novos programas serem inseridos. Em [Dimitropoulos et al. 2018], os autores propõem a virtualização de programas sem exigir recirculação de pacotes. Porém, eles não proveem isolamento entre as funções inseridas. Em [Zhang et al. 2017], também encontramos uma proposta de hypervisor utilizando P4, porém, empregando um número muito reduzido de tabelas para realizar o ordenamento topológico de maneira dinâmica. Diferentemente, nossa proposta utiliza apenas uma tabela adicional para suportar vários programas e não recorre à primitiva de recirculação.

Segurança no Plano de Dados Como qualquer outro paradigma, redes definidas por software necessitam de mecanismos para proteger seu funcionamento. Em [Hu et al. 2014] os autores propõem um firewall para redes SDN que executa em *switches* e permite resoluções efetivas de políticas de violação de firewall em redes OpenFlow. Para evitar a inserção de regras conflitantes que violem as políticas de segurança, os autores propõem uma camada para o plano de controle que resolve ambiguidades entre regras a serem inseridas. Em [Sonchack et al. 2016], os autores apresentam o OFX, sistema que permite a disposição de funções de segurança em switches, mas cuja estratégia não é adequada para processadores de pacotes genéricos. Em nosso trabalho, propomos um

mecanismo que permite implantar funções de segurança utilizando P4. Argumentamos que nosso design evita regras conflitantes entre diferentes aplicações/serviços e pode ser implantado em processadores genéricos de pacotes.

Monitoramento Trabalhos que permitem realizar o monitoramento de pacotes no plano de dados, são divididos entre aqueles que tentam fornecer abstrações para identificar e armazenar informações sobre fluxos de pacotes (heavy hitter e fluxos elefantes) [Sivaraman et al. 2017] e aqueles que fornecem mecanismos eficientes para telemetria e agregação da informação do plano de dados [Kim et al. 2015, Van Tu et al. 2017, Marques and Gaspary 2018]. Embora nosso trabalho seja quase ortogonal ao que propõem esses pesquisadores, acreditamos fornecer elementos que complementam seus estudos. Ao passo que aqueles não demonstraram a implantação de suas funcionalidades em um plano de dados de programável, entendemos que nossa proposta preenche essa necessidade.

6. Conclusões

Neste trabalho apresentamos uma estratégia de composição de programas P4 para estender a funcionalidade de dispositivos de planos de dados programáveis. A estratégia é dividida em uma etapa de composição da máquina de estados de *parser* de pacotes e em uma outra etapa complementar, em que as ações e os construtores do fluxo de controle são estendidos em uma arquitetura modular e que permite configuração dinâmica. Nós apresentamos um estudo de caso, mostrando o funcionamento do mecanismo para dois programas modulares: um módulo de monitoramento que utiliza técnicas de *heavy hitter* e um firewall que armazena o estado de conexões TCP. Os resultados das avaliações realizadas mostram que é possível compor programas para o plano de dados programável utilizando nossa estratégia sem impactar significativamente no atraso e vazão de processamento dos pacotes. Nós atribuímos isso ao uso muito reduzido de recursos, incluindo tabelas e lógica de controle.

Embora nossa estratégia garanta uma boa utilização dos recursos do switch ao compor módulos distintos, principalmente por causa da utilização de apenas uma tabela adicional, ela ainda enfrenta várias limitações. Em particular, a estratégia introduz alguns *overheads* ao plano de controle, exigindo que o desenvolvedor de um módulo seja responsável pela corretude do direcionamento dos pacotes.

Atualizações Consistentes Devido a essa limitação, existe a necessidade de uma outra etapa de verificação durante o funcionamento da rede para garantir que um pacote não passe por duas configurações distintas enquanto é processado. Isso ocorre pois os módulos internos podem possuir suas próprias tabelas e elas podem ser atualizadas de maneira que corrompa a configuração de direcionamento gerada pela tabela Shadow.

Regras Sobrepostas Embora a tabelha Shadow possa facilitar o direcionamento dos fluxos, a inserção de regras sobrepostas na tabela pode gerar direções conflitantes dentro do switch. Nós vemos como trabalho futuro o desenvolvimento de um mecanismo que filtre e resolva as sobreposições.

Operadores de Composição Esse trabalho focou em apresentar uma estratégia para compor e direcionar os fluxos pelos modulos compostos. Como parte de um trabalho em andamento nós investigamos a utilização de operadores de composição para facilitar ao administrador de redes a etapa de arranjo dos móudulos conforme suas necessidades.

Em trabalhos futuros, pretendemos utilizar a estratégia de composição em um *hypervisor* para planos de dados, onde as extensões possam ser adicionadas e removidas de maneira automática pelo operador de rede. Também desejamos construir uma interface adaptável para o plano de controle, eliminando a necessidade de reescrever as aplicações do plano de controle ao inserir um novo módulo no plano de dados. Também são necessários mecanismos que garantam que as atualizações dinâmicas não permitam que um fluxo passe por mais do que uma configuração enquanto o plano de dados é atualizado.

Referências

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

Dimitropoulos, X. A., Dainotti, A., Vanbever, L., and Benson, T., editors (2018). *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*. ACM.

Feamster, N., Rexford, J., and Zegura, E. (2014). The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98.

Freire, L., Neves, M., Leal, L., Levchenko, K., Schaeffer-Filho, A., and Barcellos, M. (2018). Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research*, page 4. ACM.

Garcia, L. F. U., Villaça, R. S., Ribeiro, M. R. N., Martins, R. F. T., Verdi, F. L., and Marcondes, C. (2018). Minicurso introdução à linguagem p4 - teoria e prática. In *XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, Campos do Jordão, Brasil. SBC.

Hancock, D. and van der Merwe, J. (2016). Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 35–49, New York, NY, USA. ACM.

Hu, H., Han, W., Ahn, G.-J., and Zhao, Z. (2014). Flowguard: building robust firewalls for software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 97–102. ACM.

Jin, X., Liu, H. H., Gandhi, R., Kandula, S., Mahajan, R., Zhang, M., Rexford, J., and Wattenhofer, R. (2014). Dynamic scheduling of network updates. *SIGCOMM Comput. Commun. Rev.*, 44(4):539–550.

Katta, N. P., Rexford, J., and Walker, D. (2013). Incremental consistent updates. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 49–54. ACM.

Kim, C., Sivaraman, A., Katta, N., Bas, A., Dixit, A., and Wobker, L. J. (2015). In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*.

Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Caşcaval, C., McKeown, N., and Foster, N. (2018). p4v: Practical verification for programmable data planes.

Lopes, N., Bjørner, N., McKeown, N., Rybalchenko, A., Talayco, D., and Varghese, G. (2016). Automatically verifying reachability and well-formedness in p4 networks. Technical report, Technical Report.

Marques, J. A. and Gaspary, L. P. (2018). Explorando estratégias de orquestração de telemetria em planos de dados programáveis. In *Simpósio Brasileiro de Redes de Computadores (SBRC)*, volume 36.

Nguyen, T. D., Chiesa, M., and Canini, M. (2017). Decentralized consistent updates in sdn. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 21–33, New York, NY, USA. ACM.

Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. (2012). Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334, New York, NY, USA. ACM.

Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. (2017). Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 164–176, New York, NY, USA. ACM.

Sonchack, J., Smith, J. M., Aviv, A. J., and Keller, E. (2016). Enabling practical software-defined networking security applications with ofx. In *NDSS*, volume 16, pages 1–15.

Van Tu, N., Hyun, J., and Hong, J. W.-K. (2017). Towards onos-based sdn monitoring using in-band network telemetry. In *Network Operations and Management Symposium (APNOMS), 2017 19th Asia-Pacific*, pages 76–81. IEEE.

Zhang, C., Bi, J., Zhou, Y., Dogar, A. B., and Wu, J. (2017). Mpvisor: A modular programmable data plane hypervisor. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 179–180, New York, NY, USA. ACM.

Zheng, P., Benson, T., and Hu, C. (2018). Shadowp4: Building and testing modular programs. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 150–152. ACM.

Zhou, Y. and Bi, J. (2017). Clickp4: Towards modular programming of p4. In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, pages 100–102, New York, NY, USA. ACM.

APPENDIX B — ACCEPTED PAPER – NOMS 2020

Programmable Data Planes (PDP) enable more flexibility for the operation of networks. The various benefits of programmability have led the community to develop new software on both academic and industrial capacities. To fully reap the benefits of programmability, it should be feasible to compose and operate multiple PDP programs into a single target switch as needed. However, existing techniques are not suitable in the sense that they: (1) use an excessive number of parser states and tables; (2) lack abstractions for the steering of packets through the control flows of programs. As such, they do not support modular composition of PDP programs. This paper proposes a composition mechanism that also addresses the fundamental needs of packet steering between PDP program modules. PRIME (Programming In-Network Modular Extensions) enables network operators to specify compositions of P4 programs and how traffic traverses these programs. The composition employs a verification phase to identify ambiguities between applications and avoid loops inside the switch pipeline. An additional table and a control plane management framework enforce the steering of packets through control flows. We present a prototype of PRIME, along with a proof of the steering correctness. The prototype shows that it is possible to achieve module-wide compositions at little additional cost in terms of delay and throughput.

- **Title:** PRIME: Programming In-Network Modular Extensions
- **Conference:** NOMS 2020 - IEEE NOMS 2020 IEEE/IFIP Network Operations and Management Symposium
- **Type:** Main Track (Full Paper)
- **Qualis:** A2
- **Held at:** Budapeste, Hungria

PRIME: Programming In-Network Modular Extensions

Ricardo Parizotto, Lucas Castanheira, Fernanda Bonetti, Anderson Santos, Alberto Schaeffer-Filho
Federal University of Rio Grande do Sul, Porto Alegre, Brazil
Email: {rparizotto, lbcastanheira, fernanda.bonetti, assilva, alberto}@inf.ufrgs.br

***Abstract*—Programmable Data Planes (PDP) enable more flexibility for the operation of networks. The various benefits of programmability have led the community to develop new software on both academic and industrial capacities. To fully reap the benefits of programmability, it should be feasible to compose and operate multiple PDP programs into a single target switch as needed. However, existing techniques are not suitable in the sense that they: (1) use an excessive number of parser states and tables; and (2) lack abstractions for the *steering* of packets through the control flows of programs. As such, they do not support modular composition of PDP programs. This paper proposes a composition mechanism that also addresses the fundamental needs of packet steering between PDP program modules. PRIME (Programming In-Network Modular Extensions) enables network operators to specify compositions of P4 programs and how traffic traverses these programs. The composition employs a verification phase to identify ambiguities between applications and avoid loops inside the switch pipeline. An additional table and a control plane management framework enforce the *steering* of packets through control flows. We present a prototype of PRIME, along with a proof of the steering correctness. The prototype shows that it is possible to achieve module-wide compositions at little additional cost in terms of delay and throughput.**

I. Introduction

Software-based paradigms for networking enable decoupling software solutions from the hardware in which they execute, making the management and operation of the network infrastructure more flexible and adaptive. Software-Defined Networking (SDN) [1] promotes the separation of the control logic from the forwarding behavior of network devices. More recently, Programmable Data Planes (PDP) offer more flexibility in the development of protocols and network functionality by allowing packet processing at line rate in the switch itself. This motivated many emerging applications, such as NetCache [2] or P4xos [3], to bring part of the processing back to the data plane to achieve economies at scale and lower operating costs. As such, operators can leverage programmable hardware to, for instance, process or analyze data [4], thereby enabling faster reactions in contrast to packet mirroring to middleboxes or controller-based applications [5][6].

Rather than writing one monolithic program, it should be straightforward for PDP software to be shared and composed into switches as needed [7][8][9]. However, existing languages for data plane programming do not support modular development. P4 ("Programming Protocol-independent Packet Processors") [10], one of the most popular languages for PDPs, requires developers to perform extensive source code modifications if they want to deploy multiple applications into a single switch. As a result, researchers have responded by offering virtualization instances that dedicate multiple PDP programs to the same physical target [11]. Virtualization typically refers to code composition techniques, which can both be utilized as a programming model [12] or for the automation of code merging. Hence, virtualization avoids rewriting code from different programs manually and maintains the semantics of the system.

While composing multiple programs may promote better usage of network resources, the management of programs becomes more complex and error-prone [13]. Current efforts to virtualize various programs in a single target switch make use of an excessive number of flow tables and parser states [11][14][15]. Consequently, these techniques can severely limit throughput and increase latency in general-purpose hardware or do not fit in specialized hardware, such as netFPGAs or ASICs [16]. Additionally, state-of-the-art techniques do not suffice to provide transitional consistency between steering configurations. Without transitional consistency, changes in the steering of flows through the program modules can create intermediary states, which may cause misroutings and security holes [17] [18] [19]. New techniques are required to allow new applications to be composed, preserving transitional packet-consistency of traffic steering without degrading performance of the data plane operation.

In this work, we present PRIME, a composition mechanism for P4 programs. Instead of building only monolithic applications, we provide abstractions for code reuse and traffic steering in a consistent manner. Specifically, PRIME implements an interpreter to parse and merge P4 programs (*e.g.,* security functions, including firewalls, access controls, and DPIs), in the manner defined by the network operator. As programmers may want to instantiate programs without rewriting their constructs (*e.g.,* tables, actions or parser states), the composition extends P4 programs, placing an ordered set of programs and isolating resources between them. A custom verification phase detects and corrects ambiguities between the control flow of modules, consequently avoiding undesired loops inside the switch pipeline. Dynamically, PRIME allows network administrators to specify the steering

978-1-7281-4973-820$31.00

of traffic through the composed programs. The key insight is to deploy programs statically and use per-packet state to steer flows using one single additional table. PRIME then provides a control plane interface to specify steering updates and send the necessary table entries to switches.

Overall, this paper makes the following contributions:

- Identifies a set of features that network operators would require in order to compose multiple programs at a single switch.
- Explores state machine composition techniques for providing a programming operator for the P4 language that merges independent programs.
- Designs a composition model for P4 programs and provides a control plane interface to steer flows through the composed programs. The interface provides the means to update the configuration without creating intermediary states.
- Implements use-cases with existing applications written as P4 programs and provides initial evidence of the feasibility and benefits of using PRIME.

The remainder of this paper is organized as follows: Section II provides a brief overview of Programmable Data Planes and traffic steering constraints. Section III describes the design of PRIME and the composition mechanism. Section IV provides a preliminary evaluation of PRIME. Finally, Section V presents related work and Section VI presents concluding remarks and future work.

II. Background

This section summarizes the P4 abstraction for PDP. We show that configuring the composition of P4 programs requires the developer to preserve steering constraints.

A. Programmable Data Planes

Data Plane Programmability has been proposed as a means to deploy new features without the need to buy new hardware. The development of specification languages such as P4 [10] enabled operators to change the behavior of programmable switches without rewriting low-level instructions (*e.g.*, the kernel of OvS [20], integrated circuits of hardware switches, or components of simulation environments). P4 allows programming and configuration of forwarding devices, including specific actions or control calls. In contrast to standard OpenFlow switches [21], P4 enables network developers to build programs that modify the structure of packet headers and can store complex network state on the data plane.

The PDP abstraction divides the data plane behavior into three main blocks: The packet Parser, Control Flows and the Deparser. The Parser is a state machine that describes how to read headers from incoming packets, where the state is the header structure and transitions are a function from header attributes to another state. Therefore, the parser specifies the order each header is instantiated to local variables. After a packet is processed by the parser it follows to a pipeline of Control Flows. Each control flow is composed by a set of logical *match+action* tables implemented using *match+action units* (MAUs). An *apply* block specifies the semantics and order that each MAU processes packets and modifies the content of header attributes instantiated by the Parser. The Deparser writes internal variables to the packet header and emits the packet to an output port (or recirculates it back to the parser).

The PDP abstraction divides the forwarding model into two stages: the *configuration* and the *population*. During the *configuration*, developers can configure the parser state machine, the structure of MAUs and the semantics of control flows. In this phase, the developer also defines header structures, metadata, and internal registers. The *population* stage allows the operator to insert, remove, or modify entries of the stateful objects, such as tables and registers that were created during the *configuration* phase. In the case of P4, the language does not dictate table update behavior. Therefore it is necessary to build tools on top of P4 to provide an update command for a different target switch, *i.e.*, when a packet matches a rule, an action is invoked with parameters supplied by a control program.

B. Traffic Steering Constraints

Virtualizing multiple P4 program modules into devices brings together the necessity of abstractions to steer flows through the composed program modules. This, in turn, creates new difficulties for the network operation. The steering configuration must be easy to manage and semantically coherent with the policy specified by the network operator [17].

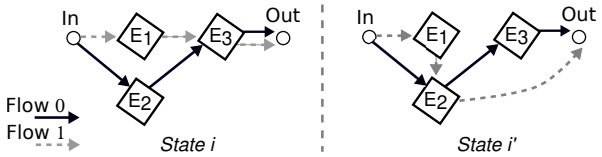


Fig. 1: Switch state transition

Figure 1 presents two different states of traffic steering. In the example, the network state i steers Flow 1 packets through extensions $(E1, E3)$ and Flow 0 through extensions $(E2, E3)$, respectively. For some reason, it might be desirable to achieve a transition between the state configuration i to state i', in which $(E1, E2)$ process Flow 1. However, this change of configuration is error-prone and can create undesirable intermediary states, *i.e.,* a packet may see part of state *i* and part of state *i'*. In the example, an intermediary state can be created by performing the update of $E1$ before updating $E2$, leading a new packet to reach $E2$ without having the proper instructions to process it.

III. Design

Data Plane programmability allows network administrators to modify the behavior of their forwarding devices. However, it is challenging to compose data plane programs deploying only the necessary functionalities in each switch without rewriting code for each different device [22] [9]. In this

section, we describe an overview of PRIME, a mechanism for network administrators to compose different PDP programs in each switch of the network. PRIME enables network operators to easily deploy only the necessary modules in each switch without rewriting code to build different configurations with the available programs.

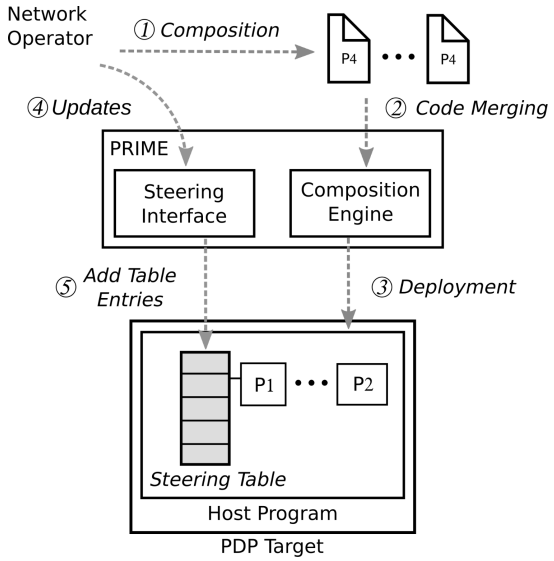


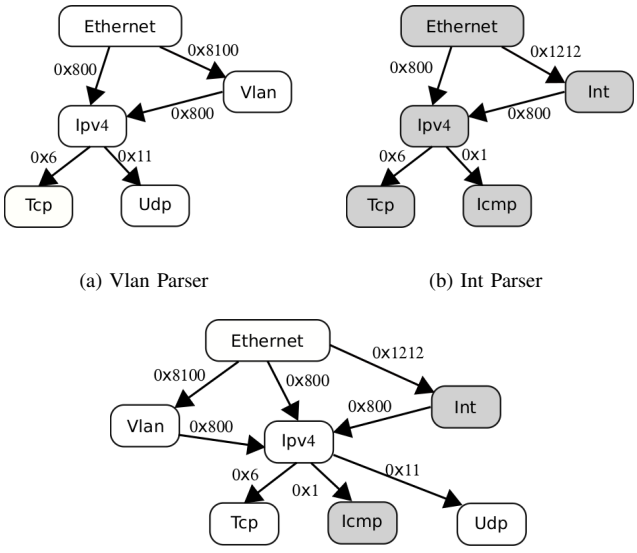
Fig. 2: High-level system architecture of PRIME

Figure 2 illustrates the scope of operation of PRIME. The functionality of PRIME is divided into two different components: a composition engine and a steering interface. The composition engine allows the network operator to compose modular P4 programs (Figure 2, Step (1)). Specifically, the system interprets P4 programs and merges them into a single code (Figure 2, Step (2)) to be deployed on the physical devices. During the composition, PRIME performs an additional verification step to identify and correct ambiguities between program modules. After the code is merged, the deployment of a new composition is performed statically, *i.e.*, requires the switch to be rebooted to instantiate a new functionality (Figure 2, Step (3)). After the deployment, the operator can utilize the steering interface to specify the steering of specific subsets of traffic through sequences of program modules during run-time (Figure 2, Step (4)). To avoid misrouting during updates of the steering configuration, we provide the means to avoid intermediary states and show why they suffice for a correct implementation (Figure 2, Step (5)). Next, we present these components in details.

A. Composing PDP Programs

The composition engine provides the means to assemble large P4 programs by merging smaller modules. We call these programs *"extensions"* and the merged program the *"host"* program. The host program is a P4 program which has an additional table, control blocks and metadata control which works as a base architecture to compose extensions. Given a set of program extensions and the host program, the composition aggregates the functionalities of the set of extensions to the host program. The system assumes that each extension is syntactically correct and verified by the standard P4 compiler to perform the composition. Then, the system computes the composition by scanning parsers and control flows and merges the respective structure definitions according to the semantics of the composition and the characteristics of the modules themselves. These aspects are explained in detail below.

Extending Parser Trees. Let the composition operation on packet parsers of an extension E and a host program H be $C : \Gamma_E \times \Gamma_H \mapsto \Gamma_L$. We define the composition of parsers as the union of the set of terminal states, non-terminals, transitions, and header definitions of the extension. The composition result is a new parser state machine Γ_L, that (1) merges states with the same ID; (2) performs the union of state transitions from the extension and the host. Figure 3c presents an example of the composition of the two parser state machines that are shown in Figure 3a and Figure 3b. The composition result merges Ethernet, which now has the transition `0x8100` to Vlan and `0x1212` to INT. Finally, State ICMP is included in the parser with a transition `0x1` from already known State IPv4. The inclusion of a new state also carries its header definitions, *i.e.*, the composition merges the definitions of packet header and the state ICMP into the composed program [23].



(a) Vlan Parser

(b) Int Parser

(c) Parser Composition

Fig. 3: The composition of parsers outputs a new parser state machine that merges state transitions and state definitions

Parser Verification. To ensure that the composition is correct, we constrain the scope of extensions, requiring the resulting parser to be deterministic and loop-free. These are restrictions we need to enforce in order to ensure

the composition of parser operates correctly. We say that Γ_E $extends$ Γ_H if Γ_E satisfies the restrictions imposed by Γ_H. For this, after the modules are interpreted and merged, PRIME performs a custom verification phase [24]. The verification performs an in-depth search in the result of the union of packet parsers. Once each packet parser is a tree, the composition can create a graph with loops and non-determinism, and in this case the verification will find existing loops. If the composed code passes this analysis, *i.e.,* Γ_E $extends$ Γ_H, then $\Gamma_L = \Gamma_E\ C\ \Gamma_H$, it can be considered "certified" and safely composed with the host program. Otherwise, the administrator is notified with a warning. We intend to investigate ways to repair those cases automatically, *e.g.*, recirculating packets through a new independent parser that could not be merged because of the host program restrictions [25].

Deparsing. Each state composed with the program must be carefully emitted to ensure packets are well-formed [26]. For instance, the system should not emit IPv4 headers before emitting TCP, which could impact on out-of-order read/writes on the next network hop. Once the structure of deparsers does not convey sufficient information to establish a dependency among them, we cannot infer the order in which packet headers must be emitted in the composed program. To avoid that disruption, the composition of deparsers must (1) unite the set of emitted headers from both programs, and (2) create a new deparser that emits headers in the same order as they are instantiated by the parser [27].

Control Flow Arrangement. Control flows of P4 programs include additional definitions of actions, tables and conditional branches (*if-else* statements) inside of control blocks. To extend functionalities of two control blocks, the composition operator introduced earlier enables the network administrator to isolate control flow blocks in a static manner [28][29]. The composition aggregates program modules into an additional table to the host program, which we call the *"steering table"* according to the semantics of the composition operator and the constructs of the P4 program. The composition operator can be utilized between two P4 programs to merge the control flow of a new extension to the beginning of the pipeline of the steering table. In practice, control flows of the programs appear in the host program in the order in which they were composed.

Constructs Disambiguation. Merging tables may promote space optimization, but creates the possibility of violating target-independent constraints, such as the equivalence between table structures, table dependencies, and loop-freeness (which is a restriction imposed by the P4 language and the data plane itself) [15]. To ensure the composition does not break target-independent constraints, this process isolates tables and registers. For this, PRIME performs a verification step to identify the equivalence of structures between the composed tables and ensure they do not violate table dependencies. For tables with ambiguous IDs, PRIME renames their IDs and rewrites the *"apply"* construct for the merged structure to use the proper ID and preserve dependencies of both modules [30]. The same isolation is performed for registers, actions, and metadata definitions with ambiguous IDs.

With the aid of the *"steering table"*, the composition produces a sequence of program modules whose execution order can be altered dynamically. For instance, the composition can change the order of execution of a firewall and a load balancing. Specifically, a firewall must be applied before load balancing incoming packets as the firewall must consider the original IP addresses. Conversely, the load balancer must first restore the original IP address before the firewall handles outgoing packets [31]. The structure of the host program and the composition assures a data plane structure that allows the configuration of both directions. Each composition translates into a configuration that works as a link for a sequence of program control flows. Next, we discuss the steering configuration in details.

B. Steering Configuration

After statically composing all the necessary modules using the technique described in Section III-A, we now discuss how to steer specific traffic flows through a subset of these programs. PRIME provides a data structure for PDP programs written in P4 and a new management system to avoid transient states between configurations. The steering operation is motivated by earlier works on testing configurations for switches and routers [32], which enable multiple testing configurations inside the switch. The steering table employed by the composition is positioned at the beginning of the pipeline of the switch and intercepts all incoming packets. The table specification can match packets using wildcards, lpm or exact and works as a large catalog of pointers from specific sets of packets to sequences of program modules merged during the composition [33], [5] (a process similar to service function chaining (SFC) in the context of Network Function Virtualization [34]).

Traffic Control. When the network administrator wishes to steer packets for a specific sequence of programs, s/he describes the identifier of the flow and the sequence of modules that must process this flow. PRIME then translates the code to the tuple of parameters of the steering table. When an incoming packet matches the table, an action which we call 'catalog' loads the parameters supplied by the administrator to the internal state. Subsequently, these user-supplied parameters will be stored as packet metadata and used by the host program to determine the order in which program modules are processed. After loading a packet metadata, the packet will be processed by several rounds. A round denotes the traversing of a packet through the pipeline of programs. In each round, only one of the programs processes the packet. The host program utilizes a traffic control module to deliver the packet to the program indexed by the next program of the catalog. After a packet reaches the egress, the next program indicator is updated and the packet recirculates to start another round. This repeats until

the packet is processed by all the programs indexed by the catalog.

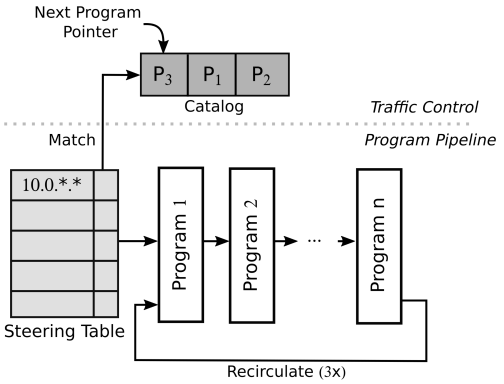


Fig. 4: Traffic steering through program modules

Example. Figure 4 presents an example of how the steering table can map flows to sequences of programs. In the example, packets that match `10.0.*.*` are mapped to be processed by programs P_3, P_1 and P_2 respectively. For this, after matching the table, the catalog forwards the packet to the ingress control-flow of P_3 and then to its egress. Next, the packet recirculates and follows to P_1 ingress and egress. Finally, the packet recirculates a third time to P_2. It is important to note that the same data plane structure supports the execution in a different order if the network administrator wishes.

Enforcing Correctness. To ensure packets will not face intermediary states of steering configurations, we reduce our problem to the *transitional packet-consistent updates* problem [17]. We explicitly state the invariants enforced by the host program and the merging itself. The host program maintains the following invariants:

- (H1) All programs in the pipeline are ordered linearly in the order they are merged, and each packet follows the pipeline in order.
- (H2) The configuration of steering is only loaded into a packet metadata in the first round, thus preventing the configuration of a transient flow from being changed by the table on other rounds.
- (H3) No changes are accepted into the steering configuration while it is already updating (this occurs because the action that loads the configuration is atomic).
- (H4) Standard metadata (*e.g*, output ports) are copied into user-metadata before recirculating and restored into new standard values to index programs into the correct processing order.
- (H5) The active steering configuration and packet header are recirculated only when *program ID* $<$ *Total # of Programs*. The next program to be processed is then updated according to the values of steering.

To ensure the composition does not violate these invariants, the merging must act accordingly. For this, the program merging satisfies the following invariants:

- (M1) Metadata definitions are verified and disambiguated to ensure no program modifies the catalog structure.
- (M2) Each table and action is disambiguated to ensure composed programs do not rewrite the catalog or apply block of the steering table.

We now can use these invariants to prove transitional packet-consistent updates for the steering configuration in the switch pipeline.

Proof. We denote pkt as the first packet entering the switch tagged with a new steering configuration S, written $S_i \rightarrow ... \rightarrow S_{i'}$.

- H2, M1, M2 ensure that when S is loaded, the steering of pkt is not modified until emitted by the egress.
- H1 ensures that a packet pkt with steering configuration S always crosses every program in the pipeline and, by the previous conclusions, finds each program in the steering configuration S.
- H3 ensures that while S is being loaded by the steering table, no other update can be performed into the steering configuration of pkt.
- H4 ensures that pkt keeps the same steering configuration S after recirculating.
- H5 ensures that packets do not recirculate forever.
- H1 proves that when pkt exists in the pipeline, all the programs in the pipeline are updated to a new configuration, even if pkt is marked to be dropped by a previous program in the pipeline.
- Hence pkt and all subsequent packets tagged with S are processed with the new configuration.

□

We claim that although the implementation of the composition of multiple programs in the same switch pipeline appears straightforward, configuring the traffic steering requires the switch to preserve certain invariants. Consistency is made possible because P4 provides per-packet states (metadata). However, metadata still needs to be copied into user-metadata before recirculating. We hope that our work provides a good motivation to rethink the design of the metadata system to facilitate the correct steering conceptually. In this section, we have shown what these invariants are, and why they suffice for a correct implementation of a packet-consistent steering configuration [17].

IV. Evaluation

In this section, we present in detail the evaluation of PRIME. We implemented a prototype of the composition mechanism to support development of programs written in $P4_{16}$. The interpreter is still in development, given that the P4 language is also in constant change. The system parses and composes the original code to P4 source code. Therefore

it maintains compatibility with any target switch. In the end, we build the new source with the `p4c` compiler to obtain the specific target code. We take large P4 programs in which we compose in different use case scenarios. Next, we present simulation results in which we show the steering performance.

A. Use Case Compositions

To validate the feasibility of PRIME, we composed existing P4 applications with a simple L2 switch program which corresponds to our Host program. Next, we discuss the details of these existing applications and the final configuration of the compositions we performed.

FlowStalker. A monitoring mechanism which encodes metrics and stores them on data plane devices. Specifically, FlowStalker [35] monitors per-flow and per-packet metrics (*e.g.,* byte counts, packet counts, timestamps) defined by the operator. The storage system employs a hash table of registers to index information for the exact flow or packet. A reactive system detects if specific flows violate local thresholds and raises a warning in the case that the threshold is crossed. Thresholds are implemented as a Heavy Hitter Detection mechanism [36], which has a pipeline of registers indexed by a 5-tuple that represents the flow. After the warning is sent, the controller can inject courier packets to collect data from data plane registers.

In-Band Network Telemetry (INT). A framework that allows the collection and reporting of network state by the data plane, without requiring intervention from the control plane. INT [37] is being utilized as a tool for several security mechanisms to troubleshoot, perform congestion control, or even notify the control applications about traffic anomalies. INT extends the packet parser with a new header, which encapsulates monitored items (*e.g.,* timestamps, buffer times, and switch identification). Monitored items are appended into a new header, which is unique for each switch. This means that a new header is instantiated and emitted by all switches in the path to an end host. The last hop removes INT headers and sends the standard packet to the end host.

LetFlow. LetFlow [38] is a load balancer that executes on switches. Letflow picks paths at random for each flowlet and balances traffic on different paths of the network. A flowlet is a burst of packets that is separated in time from other bursts by a sufficient gap (timeout). When a packet arrives, LetFlow uses a table to map flowlets to paths. Each table entry contains two fields: the last seen time and a path id. When a packet arrives, the program computes a hash (CRC-16) of the source IP, destination IP, source port and destination port. This hash is used as the key to the flowlet table. If the packet is part of an already existing flowlet, the packet is sent on the path identified by the path id, and last seen time is set to the current time. Otherwise, the packet begins a new flowlet and may be assigned to a new path at random.

P4Xos. A consensus protocol running on the data plane. P4Xos [3] is divided into three different P4 programs: the coordinator (leader), the acceptor and the learner. The coordinator ensures only one process sends messages to instances of the protocol, guaranteeing message ordering: it writes the current instance number and an initial round number into the message header; increments the instance number for the next invocation; stores the value of the new instance number; and broadcasts the packet to acceptors. Acceptors choose a value (vote) for each instance of the consensus before forwarding them. Acceptors keep a history of votes to ensure they do not vote for the same value on the same instance of consensus. Finally, learners require a quorum of messages from acceptors and "deliver" a value.

	Sce. 1	**Sce. 2**	**Sce. 3**	**Sce. 4**	**Sce. 5**	**Sce. 6**
LoC	366	489	617	779	759	649
States	3	5	5	7	7	7
Tables	4	6	12	15	15	16

TABLE I: Code metrics for PRIME compositions

We compose these programs incrementally to analyze the impact of each composition independently. Scenario 1 is a composition of FlowStalker with the host program. The idea is to build a switch with support to the analysis of security threats using the metrics collected by FlowStalker. Scenario 2 composes scenario 1 with the In-Band Network Telemetry (INT). This composition allows debugging the network state (*e.g.,* identifying the source of bugs in the network). Scenario 3 merges LetFlow to scenario 2. The idea to compose LetFlow is to allow flows to be balanced, mainly when the network performance is low. Finally, we build scenarios 4, 5 and 6 by composing scenario 3 with P4Xos. In particular, scenario 4 is a composition with the acceptor; scenario 5 is a composition with the coordinator; and scenario 6 is a composition with the learner. The composition of P4xos raises a warning during compilation, because it uses different names for packet instances. We renamed header instances of P4xos manually for the composition to be correct. Table I presents the respective number of states on the packet parser, tables and lines of code (LoC) of the composed programs in each respective scenario. As PRIME merges equivalent states between different programs, the composition tries to minimize the number of states and lines of code. Next, we discuss in details a run-time analysis of each scenario.

B. Benchmark

To evaluate PRIME we executed each scenario using the behavioral model in an Intel(R) Core(TM) i3-6006U CPU @ 2.00GHz. We performed a thousand requests, and collected packet timestamps to measure latency, and utilized `iperf` to measure throughput [16].

Figure 5 presents the throughput that the composition achieves in the data plane. When the switch steers packets through scenario 1 (*i.e.*, packets match the steering table), throughput is nearly 8 Mbits/sec. Throughput reduces as we compose more modules in scenarios 2-6.

Figure 6 presents the latency in ms with only one rule installed on the steering table (*i.e.*, the rule that matches the

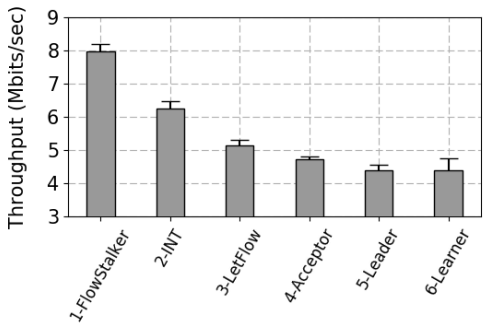


Fig. 5: Throughput

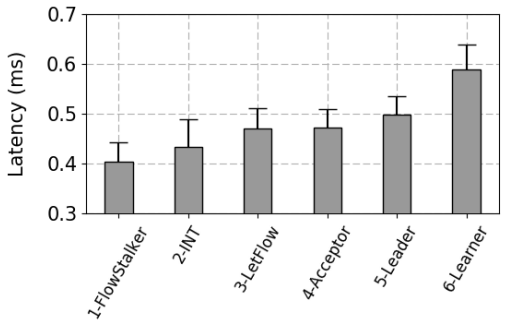


Fig. 6: Latency

end host). As we compose more program modules, latency increases. This happens because the insertion of additional states to the parser increases CPU consumption. The latency in scenario 1 is nearly 0.4ms. As we compose more program modules, such as in scenario 5, latency increases to nearly 0.5ms. We see as future work deploying PRIME compositions into high-performance packet-processing ASIC and FPGA.

C. Comparison to State-of-the-Art

We compare PRIME with one of the state-of-the-art approaches, P4Visor [15], to compose programs. Specifically, we utilized the Differential testing Operator of P4Visor to compose programs. We could not build the case studies we presented earlier because P4Visor does not currently support the composition of more than two programs. Thus we show two simple scenarios: a production version of a router with a testing version of the same program, and LetFlow with the simple router program.

	PRIME		**P4Visor**	
	Router	LetFlow	Router	LetFlow
Parser States	3	4	5	6
Tables	7	10	12	13

TABLE II: Metrics of PRIME and P4Visor Compositions

Table II presents the number of states of the parser and tables of the programs. The composition of the simple router creates fewer states using the PRIME approach. P4Visor does not merge IPv4 states, therefore creating two different states for equivalent header instances. The number of tables in PRIME is also smaller for these compositions. Although PRIME does not support abstractions to merge tables between programs, the traffic steering control has only one table. Other features to steer packets are performed only by interacting on the catalog and *if-else* statements and metadata access.

V. Related work

In this section, we review the main research efforts in data plane virtualization.

Hyper4 [11] is a hypervisor for programmable data planes. It provides a virtualization layer to run several instances of P4 programs. Although Hyper4 enables modularization, the system imposes high overhead on the forwarding because the Hyper4 base program includes several additional tables to support composition. Conversely, PRIME runs as a single P4 program, thus avoiding the virtualization-layer and utilizes only one additional forwarding table to compose modules.

MPVisor [39][40] is a hypervisor that uses P4 but provides a base program much smaller than Hyper4. The system offers high-level operators for programming P4 targets. However, their operators produce large pipelines of programs and are not sufficient for the correct operation of steering. This hinders the deployment of configurations that support multiple steering configurations. MPVisor also reduces the number of tables required to virtualize P4 programs when compared to Hyper4, but the number is still large compared with PRIME, which uses only one additional table.

P4Bricks [27] is a system for multi-processing P4 programs. The system provides parallel and sequential operators, and restructures the logical pipeline according to control flow dependencies. P4Bricks provides a low-level compilation for the target switch, which makes the system target specific and limits the utilization of the system. Although the operators proposed by P4Bricks may enable multi-processing, P4Bricks performs out-of-order readings and writes while processing packets, which can create inconsistencies and compromise the developer logic.

P4Visor [15] is a system to merge and test P4 programs. The system provides AB and Differential testing operators, which both isolate testing traffic from the composed programs. The merging of control flows tries to minimize resource sharing between modules by merging equivalent tables. The traffic isolation requires parsers from different program modules to have disambiguation states even if the merged states are equivalent. In contrast, PRIME merges equivalent states by uniting their transitions. This feature consequently reduces the number of states necessary to parse incoming packets.

Dejavu [41] is a programming model that connects and hosts several functions in a single switch pipeline. The system leverages recirculation to route packets between chains of functions and tries to minimize the number of recirculations. However, although allowing optimization of the amount

of recirculation, this can perform out-of-order processing, as functions usually are composed of ingress and egress capabilities. PRIME takes a more intuitive approach, which ensures the correct ordering of read/writes between programs.

PRIME is similar to works on virtualization, but the steering operation provides the means to achieve per-packet consistency, which is, to the best of our knowledge, a new contribution to the data plane virtualiazation.

VI. Conclusions

In this paper, we presented the design and evaluation of PRIME, a composition mechanism to help the modular development and management of P4 programs. The deployment is made by composing parsers and employing a new table into the PDP program. Furthermore, it introduces the steering of packets through the modules by using P4 metadata. We presented a case study showing the operation of our abstractions for several modular programs. Simulation results evidence that the compositions have a moderate yet acceptable impact on delay and throughput.

PRIME still faces several limitations. In particular, inserting overlapping rules into the table can generate conflicting directions on the switch pipeline. We see as future work the development of a mechanism that filters and solves the overlaps before the insertion. This feature can be designed similarly to what is presented in Hermes [33], combined with CacheP4 [42], to achieve both low update times and throughput. PRIME also requires the developer of a module to be responsible for the correctness of each independent module. There is a need for a previous verification step during network operation to ensure the consistency of each separated module.

There are several other potential future research directions. In particular, exploring new compilation techniques may allow more efficient use of data plane resources by sharing resources between programs. The development of new operators to identify dependencies between modules and a formal reasoning about the steering correctness are also in perspective. Further, in addition to the local guarantees addressed in the composed P4 program, we aim to investigate global path level guarantees for automatic virtualization of PDP programs [43], and placement heuristics similar to those used with Virtual Network Functions [44] [45]. Finally, we also see as future work a full exploration of distributed system replication techniques to handle failures.

Acknowledgments

We would like to thank Theophilus Benson for his helpful feedback. We would like to thank CNPq for research grants 407899/2016-2 and 312091/2018-4. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and also by NSF CNS-1740911 and RNP/CTIC (P4Sec) grants.

References

[1] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602219

[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 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 121–136. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132764

[3] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 2, pp. 18–24, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2935634.2935638

[4] C. Mustard, F. Ruffy, A. Gakhokidze, I. Beschastnikh, and A. Fedorova, "Jumpgate: In-network processing as a service for data analytics," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotcloud19/presentation/mustard

[5] J. Sonchack, J. M. Smith, A. J. Aviv, and E. Keller, "Enabling practical software-defined networking security applications with ofx." in *NDSS*, vol. 16, 2016, pp. 1–15.

[6] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "NICA: An infrastructure for inline acceleration of network applications," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 345–362. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/eran

[7] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, "Uncovering bugs in p4 programs with assertion-based verification," in *Proceedings of the Symposium on SDN Research*. ACM, 2018, p. 4.

[8] 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," 2018.

[9] T. A. Benson, "In-network compute: Considered armed and dangerous," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: ACM, 2019, pp. 216–224. [Online]. Available: http://doi.acm.org/10.1145/3317550.3321436

[10] 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. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[11] D. Hancock and J. van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '16. New York, NY, USA: ACM, 2016, pp. 35–49. [Online]. Available: http://doi.acm.org/10.1145/2999572.2999607

[12] Y. Zhou and J. Bi, "Clickp4: Towards modular programming of p4," in *Proceedings of the SIGCOMM Posters and Demos*, ser. SIGCOMM Posters and Demos '17. New York, NY, USA: ACM, 2017, pp. 100–102. [Online]. Available: http://doi.acm.org/10.1145/3123878.3132000

[13] R. Parizotto, L. Castanheira, and A. Schaeffer-Filho, "Abordagem de composição de programas P4 em redes programáveis," in *Anais do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. Porto Alegre, RS, Brasil: SBC, 2019, pp. 1028–1041. [Online]. Available: https://sol.sbc.org.br/index.php/sbrc/article/view/7420

[14] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "Mpvisor: A modular programmable data plane hypervisor," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 179–180. [Online]. Available: http://doi.acm.org/10.1145/3050220.3060600

[15] P. Zheng, T. Benson, and C. Hu, "P4visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: ACM, 2018, pp. 98–111. [Online]. Available: http://doi.acm.org/10.1145/3281411.3281436

[16] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, "Whippersnapper: A p4 language benchmark suite," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 95–101. [Online]. Available: http://doi.acm.org/10.1145/3050220.3050231

[17] J. H. Han, P. Mundkur, C. Rotsos, G. Antichi, N. Dave, A. W. Moore, and P. G. Neumann, "Blueswitch: Enabling provably consistent configuration of network switches," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2015, pp. 17–27.

[18] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 323–334. [Online]. Available: http://doi.acm.org/10.1145/2342356.2342427

[19] D. M. F. Mattos, O. C. M. B. Duarte, and G. Pujolle, "Reverse update: A consistent policy update scheme for software-defined networking," *IEEE Communications Letters*, vol. 20, no. 5, pp. 886–889, 2016.

[20] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 117–130.

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

[22] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[23] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design principles for packet parsers," in *Architectures for Networking and Communications Systems*. IEEE, 2013, pp. 13–24.

[24] A. C. Schwerdfeger and E. R. Van Wyk, "Verifiable composition of deterministic grammars," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 199–210, 2009.

[25] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "Pga: Using graphs to express and automatically reconcile network policies," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787506

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

[27] H. Soni, T. Turletti, and W. Dabbous, "P4Bricks: Enabling multiprocessing using Linker-based network data plane architecture," Feb. 2018, working paper or preprint. [Online]. Available: https://hal.inria.fr/hal-01632431

[28] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 87–101. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789777

[29] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 43–56.

[30] D. Saha, A. Samanta, and S. R. Sarangi, "Theoretical framework for eliminating redundancy in workflows," in *2009 IEEE International Conference on Services Computing*. IEEE, 2009, pp. 41–48.

[31] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, 2013, pp. 1–13. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto

[32] R. Alimi, Y. Wang, and Y. R. Yang, "Shadow configuration as a network management primitive," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 111–122, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1402946.1402972

[33] H. Chen and T. Benson, "Hermes: Providing tight control over high-performance sdn switches," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: ACM, 2017, pp. 283–295. [Online]. Available: http://doi.acm.org/10.1145/3143361.3143391

[34] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[35] L. Castanheira, R. Parizotto, and A. Schaeffer-Filho, "Flowstalker: Comprehensive traffic flow monitoring on the data plane using p4," in *2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019.

[36] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 164–176. [Online]. Available: http://doi.acm.org/10.1145/3050220.3063772

[37] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[38] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 407–420. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini

[39] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2017, pp. 1–9.

[40] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou, "P4sc: Towards high-performance service function chain implementation on the p4-capable device," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2019, pp. 1–9.

[41] D. Wu, A. Chen, T. S. E. Ng, G. Wang, and H. Wang, "Accelerated service chaining on a single switch asic," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, ser. HotNets '19. New York, NY, USA: ACM, 2019, pp. 141–149. [Online]. Available: http://doi.acm.org/10.1145/3365609.3365849

[42] Z. Ma, J. Bi, C. Zhang, Y. Zhou, and A. B. Dogar, "Cachep4: A behavior-level caching mechanism for p4," in *Proceedings of the SIGCOMM Posters and Demos*. ACM, 2017, pp. 108–110.

[43] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach, "Automatic virtualization of accelerators," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: ACM, 2019, pp. 58–65. [Online]. Available: http://doi.acm.org/10.1145/3317550.3321423

[44] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 14:1–14:13. [Online]. Available: http://doi.acm.org/10.1145/2774993.2774998

[45] M. Charikar, Y. Naamad, J. Rexford, and X. K. Zou, "Multi-commodity flow with in-network processing," *arXiv preprint arXiv:1802.09118*, 2018.