

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 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].

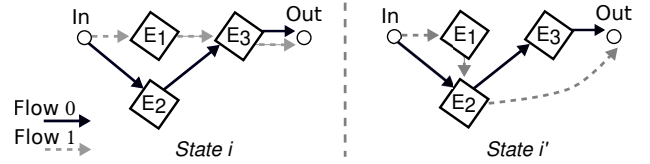


Fig. 1: Switch state transition

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.

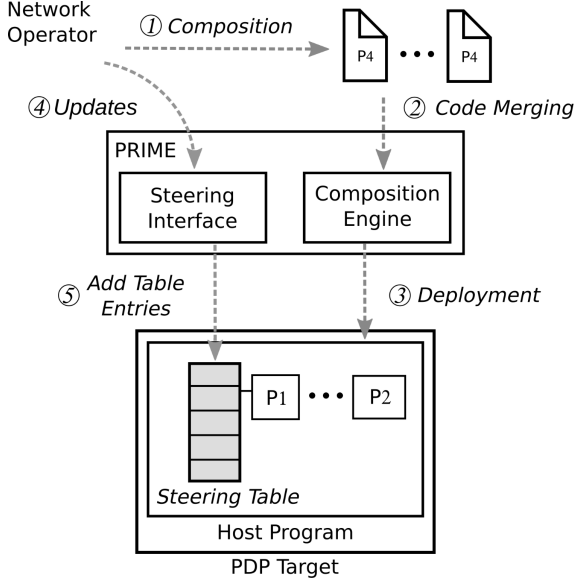


Fig. 2: High-level system architecture of PRIME

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 runtime (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].

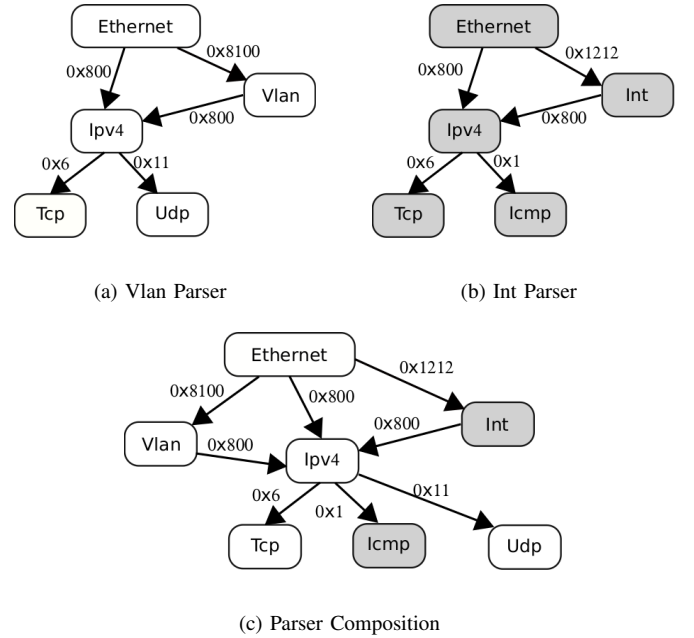


Fig. 3: The composition of parsers outputs a new parser state machine that merges state transitions and state definitions

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 $\Gamma_L = \Gamma_E \circ \Gamma_H$, 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.

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

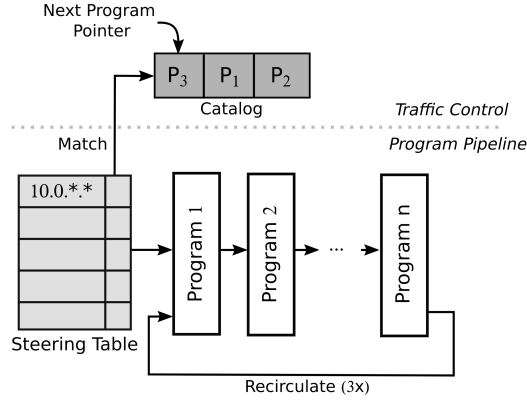


Fig. 4: Traffic steering through program modules

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 $S_i \rightarrow \dots \rightarrow S_{i'}$.

- 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.

We compose these programs incrementally to analyze the impact of each composition independently. Scenario 1 is a

	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

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].

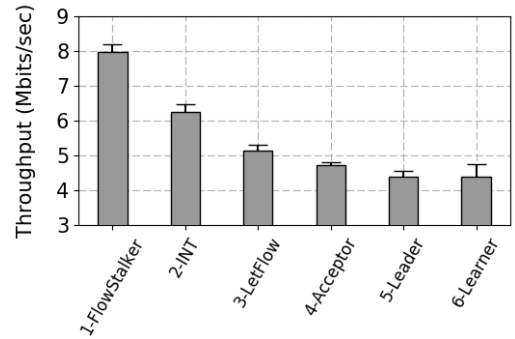


Fig. 5: Throughput

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

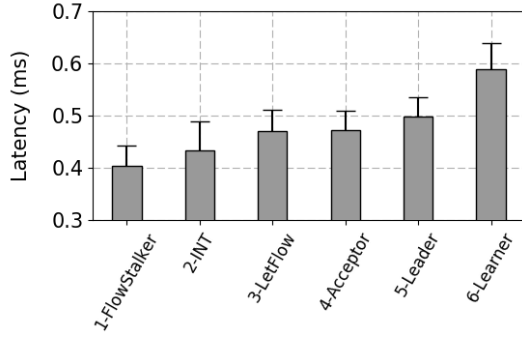


Fig. 6: Latency

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 virtualization.

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. Caşcal, 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 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/10.1145/3123878.3123000>
- [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.