Multi-Tenant Programmable Switch Virtualization Leveraging Explicit Resource Sharing

Ivan Peter Lamb[1], Pedro Arthur Pinheiro Rosa Duarte[1], Marcelo Caggiani Luizelli[2], Luciano Paschoal Gaspary[1], José Rodrigo Azambuja[1], Weverton Luis da Costa Cordeiro[1]

[1]Universidade Federal do Rio Grande do Sul (UFRGS), [2]Universidade Federal do Pampa (UNIPAMPA)

***Abstract*—With the migration of traditional computer networks to the Software-defined Networking paradigm, flexibility is a core feature that novel technologies must provide. In this context, virtualization is gaining traction in Programmable Data Planes (PDPs) as a means of achieving greater flexibility, with several solutions in the literature for instantiating virtual programmable switches on the same host device. Virtualization brings numerous advantages, enabling multi-tenancy in programmable data/research center networks and greater device resource utilization. Nevertheless, enabling a complete multi-tenant solution, in which the tenants have disjoint sets of virtual devices, requires management and security considerations not yet approached in previous investigations. Previous works focus mainly on the core underlying technology necessary to deploy multiple devices in the same physical host. This paper presents a PDP virtualization architecture based on program composition and access control for securely managing virtual switches from different tenants. Additionally, we define extensions to PDP programmability, allowing tenants to specify shared elements, such as tables, between their virtual devices. Our experiments highlight the ability to transparently manage multiple virtual switches hosted in the same physical device in networking scenarios with multiple tenants.**

I. Introduction

Software-Defined Networking (SDN) and Programmable Data Planes (PDPs) paradigms are the computer networks innovations that provide the most flexibility to researchers and operators. With PDPs, one is able to define how a data plane switch interprets and processes each individual packet flowing through it [1]. Due to this flexibility, many functionalities and services that were delegated to commodity servers following a Network Function Virtualization (NFV) paradigm can now be implemented directly in the data plane in this In-Network Computing paradigm. Examples include DNS caching, flow monitoring, storage, and distributed consensus [2]–[4]. This strategy reduces the workload on commodity servers and end hosts, leveraging PDP line-rate hardware processing power. More importantly, the delays in communication between the data and control planes are removed.

Recently, PDP virtualization has emerged as a tool to enable, among other solutions, these kinds of development environments [5]. It allows for hardware slicing of the data plane devices similar to traditional virtual machines on commodity servers [6]. That way, a single physical programmable switch, which can cost tens of thousands of dollars depending on its specifications, could be used to implement multiple virtual switches simultaneously.

However, the technological advances in PDP virtualization lack an established architecture with clear interface definitions covering the compilation, deployment, and runtime operation steps necessary to transform raw device source code into a deployable network switch [7]. Moreover, existing virtualization solutions do not provide an architecture that fully enables multi-tenancy without heavy interference on the defined behaviors of the deployed devices, while maintaining isolation between virtual switches that belong to different tenants. Additionally, a desirable property of a virtualization solution is for it to not require changes in the data and control plane source codes from the tenants. Existing solutions either acknowledge those limitations and leave them for future work [5], [8] or assume that all the virtual switches belong to one tenant, therefore not requiring additional management abstractions [9]. Multi-tenant solutions focus on implicit resource sharing between virtual switches but are either limited to a few fixed and pre-determined functionalities (such as IPv4 firewall), defeating the purpose of a fully programmable device introduced by PDPs [10], or require a complete change of the tenant's source codes [11].

This paper proposes a complete multi-tenant and target-independent virtualization architecture based on switch code composition and an abstraction layer for runtime access control. The proposed architecture allows (i) the transparent deployment of multiple programs (virtual switches) from different tenants through compostion in the same host, (ii) the mapping of the elements of the composed program, such as match-action tables and registers to the elements of the virtual switches, and (iii) an access control scheme for the tenants to securely and transparently access the elements of their virtual switches. With our architecture, no source code rewriting is required by the tenants. Existing code for both the data and control planes of the network can be deployed as virtual switches without any modifications. The source code and artifacts required for the reproducibility of our proof-of-concept implementation of the architecture and experiments are publicly available on GitHub [12]. In summary, we:

This work was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and grant #88887.954253/2024-00. This study was also financed in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CNPq) - grant #314506/2023-3 and #405940/2022-0. It also received funding from Grants #20/05183-0 (Skynet), #21/00199-8 (SMARTNESS), #23/00673-7 (IDRCIC), and #23/00816-2, São Paulo Research Foundation (FAPESP).

- Propose a comprehensive architecture to deploy and manage virtual programmable switches belonging to different tenants on any hardware target that supports standard interfaces such as P4Runtime.
- Introduce a P4 code composition strategy that allows for complete isolation of the virtual switches.
- Present a runtime module that transparently manages the virtual switches in a multi-tenant environment.

The paper is organized as follows. Section II provides a brief background on SDN, PDPs, and virtualization in the context of PDPs. Section III describes our proposed architecture along with details of our public reference implementation. Section IV describes our experimental evaluation with the reference implementation and discuss the results. The main related works are briefly mentioned and described in Section V. Finally, Section VI concludes and provides directions for future work.

II. Background and Related Works

A. Software Defined Networking (SDN)

The disruptive introduction of the concept of Software-Defined Networking (SDN) brought more configuration options to devices in the core of the network (initially with fixed protocols such as OpenFlow [13] and later with Programmable Data Planes). Since then, innovations in computer networks, whether in research or industry, have had the intrinsic objective of providing network operators with greater flexibility in managing and controlling their infrastructure's hardware and software objects.

Essentially, the SDN flexibility results from the separation of the network's functionalities into three planes:

- **Application Plane:** Composed of multiple applications that interact with the underlying network through an interface with the Control Plane. Each application has a specific functionality, such as routing, monitoring, or load balancing.
- **Control Plane:** Translates the requests from the applications to the data plane and provides other functionalities such as authentication and security. The control plane is also responsible for providing an abstract view of the data plane (that is, the connections in the network and the resources of each forwarding device) to the services that execute on the application plane.
- **Data Plane:** Contains the implemented forwarding devices of the network. Those devices receive the traffic processing logic from the control plane. They can be implemented using different technologies, such as ASICs, FPGAs, and virtual switches, among others. The standard interface with the control plane allows all those implementations to be compatible.

The virtualization solution proposed in this work focuses on (i) the data plane, where the virtualized devices are implemented, and (ii) the control plane, which must act together with each virtualized switch to provide an abstraction that allows secure isolation of virtual devices. The abstraction layer ultimately must "see" the devices in the data plane in a virtualization-agnostic way.

B. Programmable Data Planes (PDPs)

Continuing the trend of providing operators and researchers with more flexibility in the configuration and management of network devices introduced with OpenFlow, the idea of PDPs was consolidated. PDP is a model where the processing logic of the data plane packets is fully customizable (or programmable) by the operator. At first, the motivation was to eliminate the dependency on updating the set of supported protocols in the OpenFlow specification for them to be used, decoupling the definition of packet headers from future OpenFlow specifications. With PDPs, operators can independently develop, implement, and test new protocols and algorithms without the need for a third-party specification.

Operators define the logic of the device through Domain Specific Languages (DSLs) designed for that purpose. The *Protocol-Oblivious Forwarding* (POF) [14] language was the first consolidated work to introduce this concept of decoupling the processing hardware (and management API such as OpenFlow) from the processing logic of specific network protocols. From then on, network devices migrated from a *closed-box* paradigm, in which manufacturers' roadmaps and subscription plans determine their capabilities, to an *open-box* paradigm, in which the inner workings are available to operators because they define them.

After that, other works were carried out to evolve and gain adherence to this new concept, in particular the introduction of P4 (Programming Protocol-independent Packet Processors) [15], which is another DSL for defining the behavior of a generic programmable switch. This language provides full programmability of the forwarding devices, allowing operators to implement any traditional functionality of network devices (e.g., firewalls, IP routers, load balancing, congestion control, among others) in addition to innovative functionalities in a portable way to different hardware. Contrary to POF and other DSLs, P4 has had wide market adoption, with several commercial products available, including Intel®Tofino™, NetFPGA-SUME (Xilinx), and SmartNICs (Netronome and NVIDIA BlueField-2).

In the P4 architecture, the device loads a target-specific configuration binary created by a P4 compiler. Then, the device exposes the custom resources to the control plane through an interface. Those standard interfaces, such as P4Runtime [16], provide methods for manipulating the state of the programmable device during its operation, such as inserting or removing rules in the forwarding tables or reading device counters/registers. P4Runtime and other interfaces, such as Tofino's BfRuntime, are implemented using a gRPC server, an open-source Remote Procedure Calls (RPC) framework. The manufacturers typically install this server and run it inside the physical devices. The external control plane then connects as a client to carry out the requests.

To a large extent, the architecture presented in this work involves the runtime interface between the control and the data planes since it is the way to access elements of switches during operation. Still, our solution is independent of the specific target (physical hardware or software emulating a forwarding device) as long as it implements a standard interface.

C. Virtualization in PDP

Virtualization is a technique to split a host's resource (in most cases, hardware) between multiple tenants (guests). An entity called *hypervisor* is responsible for dividing the host's resources among the guests. A couple of years after the introduction of the P4 language, researchers started to study the possibility of applying virtualization in the context of PDPs [5]. The main idea is that multiple virtual devices, described by different P4 codes, could simultaneously run in the same physical target. This would be equivalent to a conventional server running multiple isolated VMs.

The first works in this line focused on emulating the behavior of P4 devices using a general-purpose program. These programs contain carefully formulated actions and tables so that it is possible to implement the operations of any other P4 program via the insertion of rules in these tables during the operation of the host device. The two main works in this line are Hyper4 [5] and HyperV [8]. The main limitations of these works are the lack of support for the whole list of operations in the P4 language (imposed by the virtualization scheme), the limited size of the virtualized programs (due to the limited number of stages of the physical devices) and the tremendous impact on the performance of virtual switches.

P4Visor [9] introduced a new approach for implementing the virtualization of PDPs with a composition-based strategy. Composition-based virtualization involves merging multiple programs into a single program that implements the logic of all composed programs. P4Visor is focused on the composition of two versions of the same program (a current version and a test version) to facilitate prototyping. Since few instructions and additional *match & action* tables are added in the composition process, the performance of this technique is much better than emulation-based ones. The tradeoff of this approach is the loss of the ability to remove and deploy virtual switches without interrupting the host operation, which is associated with the possibility of implementing virtual switches through rules added to tables in runtime [5].

There is yet work to be done towards a general architecture that addresses the limitations mentioned so that a large-scale virtualization solution can be made concrete, especially in terms of manageability and security. These aspects have been given poor attention as the previous works mainly focus on implementing multiple virtual switches belonging to a single user (tenant). Only in recent years have advancements in virtualization begun to tackle multi-tenancy aspects. However, those works rely on a fully controlled environment with limitations such as programs described as a specific composition of functionalities [10] or that enforce the use of a new programming language designed for virtualization [11]. Ideally, a virtualization solution should not impose limitations or require any change in the code that describes the behaviors intended by the programmers, both for the data plane and the control plane.

III. Virtualization Architecture

Currently, no readily available virtualization solution can be deployed with unmodified source code and control plane software. That is mainly because previous works on virtualization do not address the practical aspects of implementing a complete multi-tenant solution. Instead, they focused on scenarios where a single tenant would like to deploy multiple virtual switches under its control on a single target.

The architecture presented in this work leverages composition-based virtualization technologies presented in Section II, taking a step further and expanding upon these techniques to design a complete end-to-end multi-tenant solution that is manageable, secure, modular, and low overhead. The proposed solution considers that each virtual switch (defined by a P4 program) may belong to a different tenant, and its functionalities can be completely different from the others.

Our solution leverages *lightweight* virtualization to increase hardware usage efficiency further. To this end, common resources among the virtualized programs (tables, registers, etc.) are shared in the composed code whenever possible, with a security module that allows for that sharing to be transparent to each client, enforcing resource isolation. Other researches show that hardware optimizations for TCAMs can take advantage of aggregating tables, thus providing sub-linear memory usage as the size of those tables increases [9].

The complete architecture for our virtualization solution is depicted in Fig. 1. It shows that multiple tenants can implement a virtual switch on the target by uploading their programs to the code composition module and connecting their controllers to the runtime module. There is an interface for the system operator to populate security policies and additional data for each tenant registered in the system. The illustrated modules have the following functions:

- **Code Composition Module:** This module is responsible for composing multiple P4 programs. This module generates a new P4 program that implements the logic of all programs supplied in parallel.
- **P4Runtime Module:** The P4Runtime module implements a P4Runtime server that behaves as if it were the server running on the tenant's target. In effect, the module intercepts the RPCs and translates them to be compatible with the composite code executing on the target.
- **Security Module:** Responsible for ensuring that the tenant's requests comply with the security policies defined by the system administrator and for detecting other possible attacks on virtual switches.
- **Manager Interface:** Interface for the system administrator to define the security policies and to populate the database containing the tenant's data.
- **Dataplane Update Module:** This module implements the P4 program in the target devices, which involves procedures dependent on the target architecture, such as calls to specific compilers and drivers.

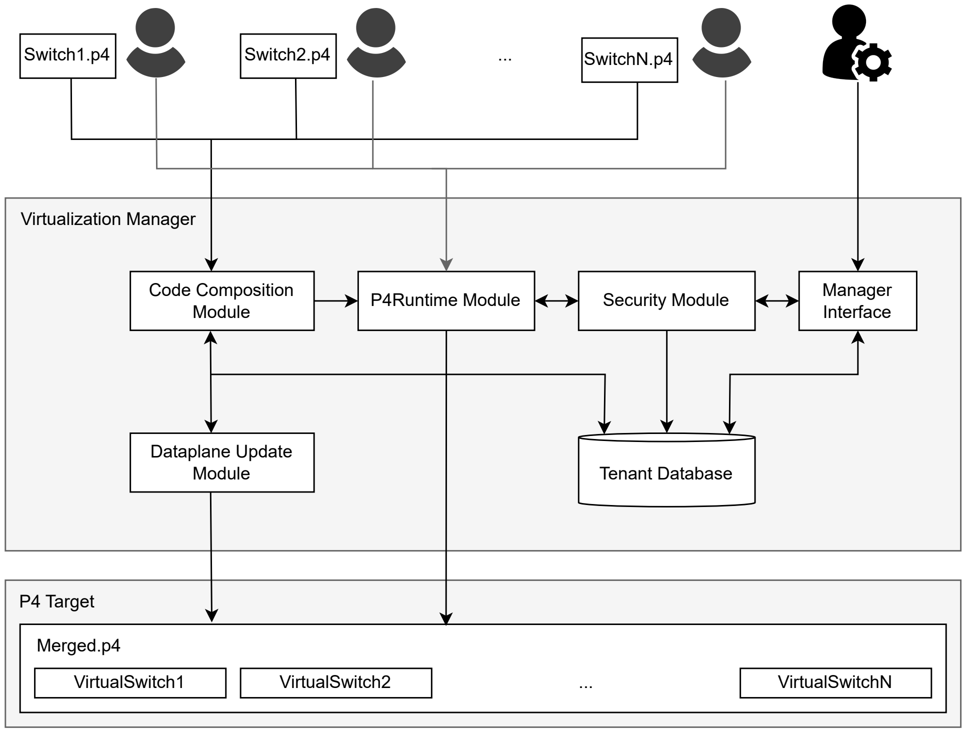


Fig. 1. Virtualization architecture.

- **Tenant Database:** Database containing tenant's information, such as available physical resources (number of ports, memory size, etc.) and access permissions to entities of the composite program, among others.

The remainder of this section provides more details on the program composition strategy of the code composition module (III-A), the abstraction during device operation provided by the runtime module (III-B), and the explicit resource-sharing mechanism (III-C).

A. Program Composition

In our proposed architecture, we employ a strategy for composing programs providing logical isolation between their resources. This composition is performed directly in the source codes (P4 language files) to achieve a *target-independent* solution. Still, each specific target may have additional requirements or restrictions on the set of operations of the composed program. An example of such a restriction is only allowing checksum computation on a specific control block in the *v1model* architecture.

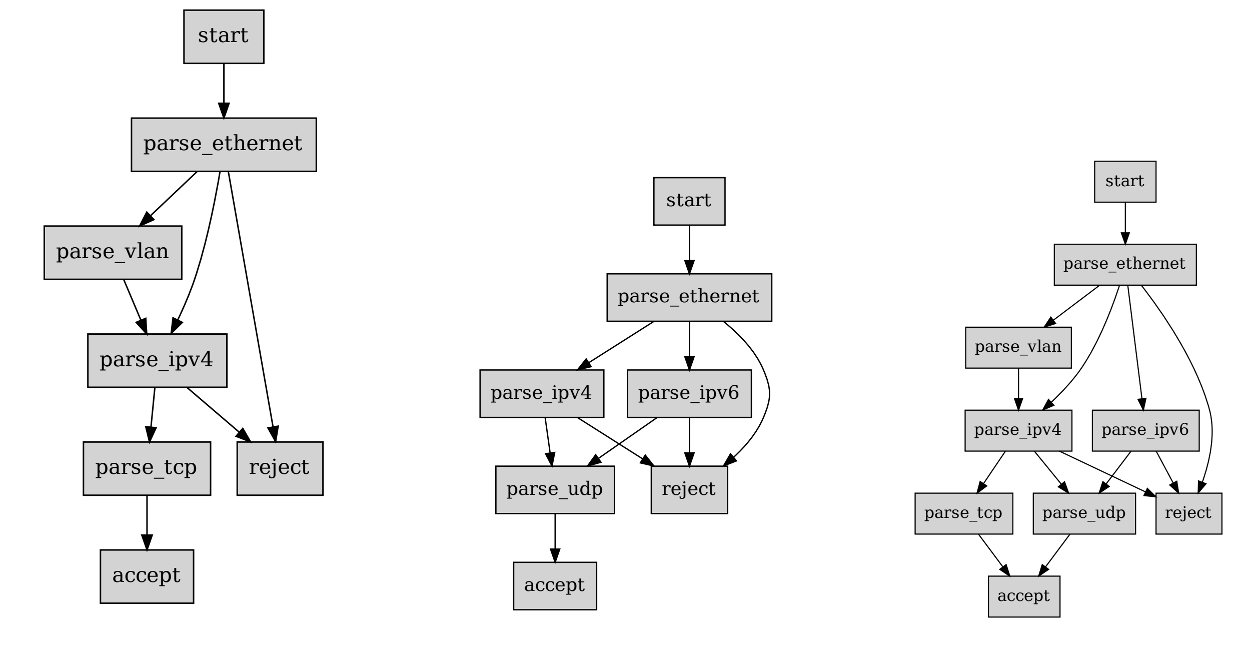
The code composition module of the proposed architecture is the core of the virtualization solution. It enables multiple P4 programs to execute simultaneously on a single target in an isolated manner. The composition is inspired mainly by the strategies proposed by Lyra [17] and P4Visor [9].

The Lyra framework can divide the programs into isolated modules (such as an IPv4 processing module and another module that processes ARP requests and replies, for example) and merge them into a single P4 program. However, the whole Lyra pipeline assumes a single tenant. By doing so, a single person or organization controls all the modules, the resulting P4 program, and whichever target they intend to deploy this P4 program. Similarly, but with the end goal of a multi-tenancy solution, the architecture described in this work focuses on how to take multiple complete P4 programs (generated by Lyra or any other method) from different tenants and merge them into a new P4 program before the deployment in the target. P4Visor has a similar workflow but with an emphasis on testing different versions of a program.

In our solution, each virtual switch has a disjoint set of ports of the physical switch that belongs to them. This way, a packet is processed by the virtual switch associated with its input port. With this strategy, it is not necessary to have external mechanisms in the network to tag packets to identify virtual switches. The result is a complete *in-target* virtualization scheme that is deployable without significant changes in the network infrastructure.

Our virtualization architecture is also lightweight in the sense that the target resources are shared by the virtual devices (implicitly or explicitly). The elements of the P4 language that are shared are (i) the states of the *parsers* and (ii) the tables and actions in the control blocks. Tenants can choose if those elements of virtual devices belonging to them are shared (explicitly) or the composition module can share them in the merging process (implicitly), and the runtime module provides the logical isolation required.

1) Parser Composition: From a formal perspective, Finite State Machines (FSMs) can model P4 parsers. Therefore, the



(a) A parser for VLAN, IPv4, and TCP (b) A parser for IPv4, IPv6, and UDP (c) The merging result

Fig. 2. Example of the merging of the parsers in (a) and (b) into (c).

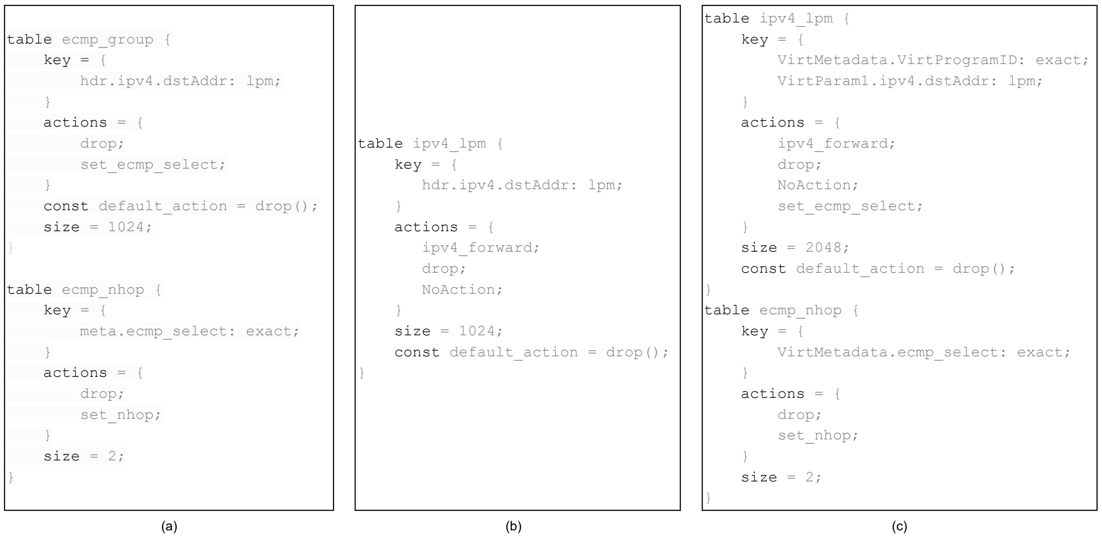
composition of P4 parsers reduces to aligning FSMs while unifying equal states. This approach results in a smaller amount of states than creating a new parser with the states of each of the parsers when there is an overlap between parsers. For instance, suppose more than one program implements a state for parsing Ethernet headers. In this case, the composed program needs only one state for these individual programs. We consider two states equal if they have the same *statements*, except for the last one, indicating the transition to the next state. The transition of merged states to the next is modified to suit the operating logic of the input parsers, using the input port to disambiguate the next state. That way, additional control mechanisms or a tag system are not required.

Fig. 2 shows an example of the composition of different parsers. It illustrates the importance of identifying the virtual switch the packet belongs to during the transition to the next state. For example, suppose a packet belonging to program *(a)* has a UDP header at the transport layer. In this case, it must go to the *reject* state in the merged program *(c)*, as opposed to the *parse_udp* state, since that state is not defined and that protocol is unavailable in program *(a)*.

2) Control Block Composition: The shared elements in our virtualization architecture on the control blocks are the *match & action* tables and the actions. Actions are the simplest elements to merge since they are just code segments executed when a *match* occurs, and two actions can merge if they are the same. In our model, we consider that two tables can merge if their match keys are the same and their default action (taken in cases of table misses) is equal and constant. We can create a new table in the composed program with an additional key field identifying the virtual switch to differentiate entries. The other elements described in the P4 language specification are the list of actions and the number of entries, which merge trivially through the union of sets of actions and the sum of the sizes of the tables. The runtime module requires additional measures to maintain the expected behavior from the tenant's controller's perspective, such as guaranteeing that the amount of entries that each tenant can insert on the tables is equal to their original declared size.

Selecting all compatible tables to perform the *merge* is impossible since the resulting control flow may contain cycles, which neither the RMT design nor the P4 language specification allows. A suitable merge must be chosen from the set of possible table merges for each control block. Currently, as a proof of concept for the architecture, our implementation selects an arbitrary merge that satisfies the dependencies instead of a more elaborate algorithm that tries to select the best possible merge (in terms of total resource usage). Selecting the best possible merge has been shown to be an NP-complete problem [9]. Still, further investigation is required to verify if common PDP code produces instances small enough to be solved by modern algorithms in a suitable time.

When tables are selected to be merged, the resulting table receives a new field to identify the program to which an entry belongs since the logical isolation of the entries of each program must be maintained. Fig. 3 shows an example of the result of composing the control block tables of two different



```
table ecmp_group {
    key = {
        hdr.ipv4.dstAddr: lpm;
    }
    actions = {
        drop;
        set_ecmp_select;
    }
    const default_action = drop();
    size = 1024;
}

table ecmp_nhop {
    key = {
        meta.ecmp_select: exact;
    }
    actions = {
        drop;
        set_nhop;
    }
    size = 2;
}
```

(a)

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

(b)

```
table ipv4_lpm {
    key = {
        VirtMetadata.VirtProgramID: exact;
        VirtParam1.ipv4.dstAddr: lpm;
    }
    actions = {
        ipv4_forward;
        drop;
        NoAction;
        set_ecmp_select;
    }
    size = 2048;
    const default_action = drop();
}
table ecmp_nhop {
    key = {
        VirtMetadata.ecmp_select: exact;
    }
    actions = {
        drop;
        set_nhop;
    }
    size = 2;
}
```

(c)

Fig. 3. Example of the composition of tables.

programs[1]. Table *ecmp_group* from program *(a)* is unified with table *ipv4_lpm* from program *(b)* forming table *ipv4_lpm* in the composed program *(c)*. The names of the tables in the composite program are for internal use only, being indifferent to the tenants, who will use the original names and IDs to refer to them in their runtime requests.

3) Target-Specific Composition: The parser and control-block composition previously discussed are target-independent composition operations that can be performed to unify multiple P4 programs for different architectures. Additional submodules may be required to merge specific parts of a program that only exist in a particular architecture (e.g., the checksum verification control blocks of the v1model architecture). We implemented and tested the Composition Module with two target-specific submodules: (i) one for the v1model architecture (used by the bmv2 software switch) and (ii) one for the Tofino Native Architecture (used by Intel®Tofino™ switches).

Still, some targets may require adaptation of the target-independent submodules if they do not support common P4 language features that those modules use (e.g, switch statements, integer division...). However, it's important to note that these modules are still considered target-independent as they are implemented using operators defined in the P4 language specification and its core module (core.p4).

B. Control Plane Runtime Abstraction

The main objective of the runtime module is to provide an interface to the underlying virtualization solution for the controllers. It offers a logical view for each tenant that their virtual device is deployed on the host device just as it would be if no virtualization solution were in place. This is done by providing the controllers of each tenant with an information file (named *p4info*) that lists the set of *match & action* tables and actions that could be handled similarly to the file that a traditional P4 compiler would generate. The composite program that is actually implemented in the target has a set of elements described in its runtime file (p4info), which can come from a single program (non-shared resource) or the result of the composition of elements from multiple programs (shared resource). The program composition stage generates a mapping file of the individual program elements to the shared program elements containing information such as the declared original size of each table and register and the available actions for match units.

By providing this logical view, the tenant's controllers that manage virtual devices can be the same as those used to manage "normal devices". This transparency is critical to allow tenants to use standard control plane software such as ONOS [18]. It would be unreasonable for the tenants to rewrite control plane software to comply with virtual switches.

One of the design principles of the proposed architecture is *target-independency*. As such, this module can also be referred to as the *runtime management* module to decouple its purpose (provide runtime management) from a specific CDPI implementation (P4Runtime). Still, as mentioned in Section II, P4Runtime is an open interface implemented by most commercial P4 hardware targets.

Fig. 4 shows the operation of the runtime module. In a traditional PDP-based operation scenario, the tenant application connects directly to the P4Runtime running in the target switch. Instead, in our virtualization solution, the tenant controller application connects to an instance of the runtime module responsible for proxying the communication with the target switch. Then, when the controller application sends a request to its virtual switch, the runtime module translates resource references to those of the merged switch program. Besides that, since the runtime module is the only one con-

[1]The codes are excerpts from programs from P4's tutorials on "basic forwarding" and "load balancing" available at: https://github.com/p4lang/tutorials/tree/master/exercises

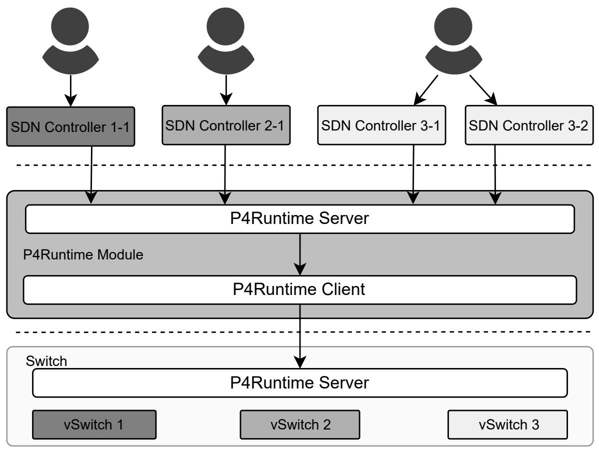


Fig. 4. Runtime module operation scenario.

necting to the target switch, it enforces access control and resource utilization restrictions. The figure also depicts that multiple controllers can manage a single virtual switch, as seen in vSwitch #3. The P4 standard anticipates such a scenario to provide fault tolerance and high-availability capabilities to the controller applications, and the runtime module supports them to avoid breaking tenants' expectations regarding these essential features and to avoid changes in the application controllers. In practice, the runtime module instances comprise a P4Runtime server that handles tenants' connections and a P4Runtime client that dispatches the requests to the appropriate physical devices.

The authentication is provided by the security module and the tenant database. As previously mentioned, P4Runtime utilizes gRPC as its underlying client/server model implementation, an open RPC framework. This framework allows for secure communication using Transport Layer Security (TLS). Therefore, it is possible to utilize a TLS certificate scheme to identify the tenants of each controller. This way, the module only forwards requests to specific elements deployed in the data plane if the virtual switch and the SDN controller belong to the same tenant. All operations defined in the P4Runtime specification are currently available in the public implementation, except for the *SetForwardingPipelineConfig*, which changes the configuration (i.e., the program) running on the switch.

C. Resource Sharing

While our proposed architecture provides complete logical isolation for the elements of different virtual devices by default, it is possible for tenants to explicitly share some resources between them. Usually, it is not possible to share resources among devices belonging to different tenants, but the architecture is flexible enough to enable this behavior should the system administrator require it.

By opting to share elements such as tables and registers, devices that belong to the same tenant could have a unified firewall table that blocks suspicious connections or a shared local DNS cache [3], for instance. This further increases hardware savings and allows for cooperation through information sharing between the virtual devices.

For the P4 language, explicit resource sharing is implemented through additional annotations on the shared elements. The use of this kind of annotations in the language allows for easy extensions, thus maintaining compatibility with other programs and compilers.

IV. Experimental Results

To validate the proposed architecture, we implemented a prototype of the architecture composed of the Code Composition, Runtime, and Security modules. We performed two experiments to ascertain the virtual switches' isolation and security, measure the solution's additional runtime overhead in terms of latency between the control and data planes (the time it takes for the RPCs to complete), and evaluate the virtualization scheme's resource usage overhead (the ammount of physical memory used in the host). The experiments aim to answer the research questions: (i) What is the system's additional runtime latency for the controllers? (ii) Is it possible to save resource usage with the proposed lightweight virtualization scheme?

Additionally, our second experiment highlights topology emulation as a potential use case for a virtualization architecture. PDP innovations must undergo a prototyping phase under controlled environments before deployment. Traditionally, software emulators such as Mininet [19] are employed for this phase. While not ideal due to performance limitations, those software emulators must be employed when researchers have limited resources, such as a limited number of physical switches. With virtualization, multiple tenants can simultaneously emulate a whole topology of switches with guaranteed hardware performance.

We employed a programmable Intel Tofino switch with an aggregate throughput of 6.4 Tbps for the experiments. The source code for the implementation in C++ with 14000 lines of code and scripts to foster reproducibility are available on a GitHub repository [12].

A. Experiment #1: Composition and Latency

This experiment consists of two P4 programs which contain a table to forward packets based on the destination IP address. The programs were designed to evaluate the solution's ability to merge programs that pose some challenges, such as (i) different headers for packet-in and packet-out runtime messages, (ii) a table that can be shared to reduce TCAM hardware usage but such sharing must be transparent to the tenants, (iii) different parser and control flow logic, and (iv) runtime idle timeout messages that must be translated to the controller's expected formats.

The first program was extended to be able to receive packets from the controller, which are processed by populating metadata fields containing information about the switch's queue occupation. The controller is designed to operate as a simplified in-band telemetry application, which periodically sends packets to the switch to get information. The experiment measures the delay between the telemetry request (packet-out) and the response (packet-in).

The second program was extended to send any packets that resulted in a table miss to the controller for further inspection, along with metadata that informs the packet's ingress port. This way, the controller can insert rules in the switch's table as needed, similar to a simplified reactive forwarding application. The rules inserted by the controller also contain an *idle timeout* of 5 seconds, which will cause the switch to generate a message to warn the controller of inactive entries. The delay we want to measure is the RTT of the first packet of a ping between two connecting hosts.

TABLE I
DELAYS MEASURED ON THE LATENCY EXPERIMENT.

	Virtualization (ms)	**No Virtualization (ms)**
C1 (Telemety)	4.121	2.231
C2 (Forwarding)	19.09	11.06

Table I shows the delays for the telemetry packets for tenant 1 and the RTT of the first packet of a ping between two hosts of tenant 2. For comparison, the table also shows the same delays when only the code of one of the switches is running (i.e., no virtualization). The results are an average of 30 repetitions. The controller applications used with and without virtualization are identical because our solution provides a transparent virtualization scheme. The resulting overhead of the experiments shows that the virtualization scheme performs well for the chosen applications. The telemetry application only perceives a 2ms average additional delay every 3 seconds, and the forwarding application only incurs an 8ms average additional delay every time a new flow needs to be established. The additional delays differ, since it is dependent on the operations performed by each application.

In summary, our solution was able to compose programs with different pipeline and packet metadata specifications and manage them in parallel in a completely transparent way to the controllers. The overhead delay, acceptable for the applications, is a small price for the advantage of running multiple virtual switches in parallel with real hardware line rate processing and throughput.

B. Experiment #2: Block RAM Usage

The proposed virtualization architecture is able to share hardware resources while providing logical isolation to the tenants. Additionally, tenants may opt to explicitly share their resources among their own virtual switches. Due to hardware optimizations, sub-linear space growth can be achieved when the declared table sizes grow.

With the developed prototype, a tenant can specify not only a single virtual switch to be instantiated but also multiple devices and virtual links between them. It does so by sending a JSON file with this information to the Code Composition module instead of a single P4 file. The modular design of the architecture allows for such modifications to fit each operator-specific needs. Resources and tables can be shared between those devices, such as a common firewall table.

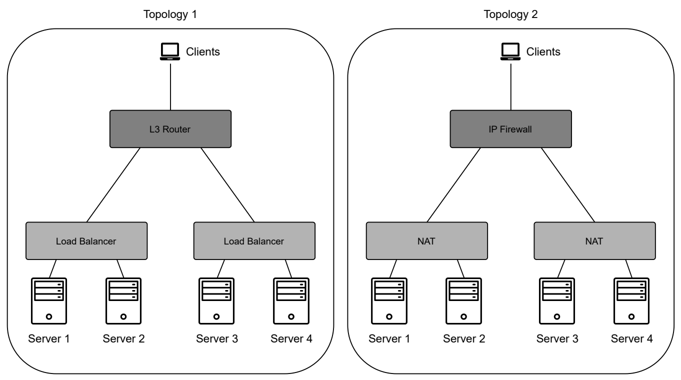


Fig. 5. Emulated topologies for the hardware usage test.

To ascertain that it is possible to save hardware resources, we instantiated six virtual switches divided into two topologies, each topology belonging to a different tenant. Fig. 5 depicts the emulated topologies.

Table II shows the Block RAM (BRAM) usage for each emulated switch when their programs are compiled individually. The table also shows the same values for the composed program. The results show that it is possible to save hardware space when tables are unified. The total number of BRAMs used (46) is 21% less than a composition without table unification (58)[2].

V. Related Work

The first works in PDP virtualization (Hyper4 [5], HyperV [8] and P4Visor [9]) were described in Section II, since they introduced important concepts such as emulation and composition-based PDP virtualization. While those works laid the foundations for further research, much has changed in the processing power and architecture of PDP hardware since then, requiring significant updates. The new specification of the P4 language, $P4_{16}$, is incompatible with their original implementations. Most recently, multi-tenancy PDP virtualization is gaining traction, with solutions targeting modern hardware.

P4MT [20] recognizes that prototyping and testing are critical steps in the development of innovative network protocols and services. They reinforce the importance of a complete testbed to ensure that most problems are identified and fixed before those innovations are deployed in production. To increase the support for this part of development, they proposed i-P4EN (International P4 Experimental Networks). The limitation of that approach is the heavy target dependency of the solution. It needs hardware support for multiple pipelines to accommodate multiple tenants. This limitation would also imply that the maximum number of tenants supported by the system is the number of pipelines available in the target

[2]Equivalent to the sum of the individual tables (53) and the overhead tables of the virtualization solution (5).

TABLE II
BRAM USAGE FOR INDIVIDUAL AND COMPOSED PROGRAMS.

Program	ipv4_t	firewall	src_t	dst_t	nhop	lb_table	Overhead	Total
L3 Router	5							5
Load Balancer					4x2	4x2		16
Firewall	4	4						8
NAT	4x2		4x2	4x2				24
Composed	14	4	8	6	4	5	5	46

hardware. To circumvent this limitation, P4MT adopts a *shared table* design in which they cannot freely design the NFVs. The tenants must choose from a set of available functions to compose their programs.

In MTPSA [10], the authors propose a virtualization solution to emphasize the need for programs belonging to different tenants to not interfere with each other's processing. Again, the architecture is heavily target-dependent, tailored for the reference software switch implementation of P4, bmv2, and FPGA-based P4 switches from Xilinx. Moreover, the architecture defines the entity of a "Superuser", which is a program that is appended at the ingress pipelines of the switches and defines which headers the users (i.e. tenants) programs can parse, such as Ethernet, IPv4, TCP, UDP, etc.

Most recently, SwitchVM [11] aims to revisit emulation-based approaches leveraging hardware advancements. The authors achieve an acceptable performance by mixing static packet processing functions with user-defined functions with code that is encapsulated in the packet headers (similar to the concept of active networks [21]). The main disadvantage of SwitchVM is the constraints imposed on the data-plane programming logic due to the newly designed programming language. Additionally, the tenants' pre-existing base code must be rewritten using that language.

VI. Final Considerations

This work presented a complete end-to-end multi-tenant PDP virtualization architecture based on program composition that provides logical isolation for each tenant's virtual devices. The architecture is target-independent and does not require changes in the tenant's code for the data and control planes. In future work, we intend to explore new applications of the virtualization architecture, such as virtual programmable switches as a service.

References

[1] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Comput. Surv.*, vol. 54, no. 4, may 2021. [Online]. Available: https://doi.org/10.1145/3447868

[2] Y. Tokusashi, H. Matsutani, and N. Zilberman, "Lake: The power of in-network computing," *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2018.

[3] H. T. Dang, "P4dns: In-network dns," *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019.

[4] J. Woodruff, "P4xos: Consensus as a network service," *IEEE/ACM Transactions on Networking*, vol. 28, 2020.

[5] D. Hancock and J. Van Der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *CoNEXT'16*. ACM, 2016, pp. 35–49.

[6] A. S. Tanaembaum and H. Bos, *Modern Operating Systems*, 4th ed. Prentice Hall, 2014.

[7] G. Bueno, M. Saquetti, P. Rodrigues, I. Lamb, L. Gaspary, M. C. Luizelli, M. F. Zhani, J. R. Azambuja, and W. Cordeiro, "Managing virtual programmable switches: Principles, requirements, and design directions," *IEEE Communications Magazine*, vol. 60, no. 2, pp. 53–59, 2022.

[8] 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 *Computer Communication and Networks (ICCCN), 2017 26th Int'l Conference on*. IEEE, 2017, pp. 1–9.

[9] P. Zheng, T. A. Benson, and C. Hu, "Building and testing modular programs for programmable data planes," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 7, pp. 1432–1447, 2020.

[10] R. Stoyanov and N. Zilberman, "Mtpsa: Multi-tenant programmable switches," in *Proceedings of the 3rd P4 Workshop in Europe*, ser. EuroP4'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 43–48. [Online]. Available: https://doi.org/10.1145/3426744.3431329

[11] S. Khashab, A. Rashelbach, and M. Silberstein, "Multitenant In-Network acceleration with SwitchVM," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 691–708. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/khashab

[12] Authors, "Project Source Code and Scripts," 2024. [Online]. Available: https://github.com/Ivanatorion/CNSM-Virtualization

[13] T. e. o. McKeown, Nick e Anderson, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 67–79, 2008.

[14] H. Song, "Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 127–132. [Online]. Available: https://doi.org/10.1145/2491185.2491190

[15] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[16] P4Org, "P4runtime specification," Oct. 2008, ver. 1.3.0. [Online]. Available: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

[17] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 435–450.

[18] ONOS. (2024) Open network operating system. apps and use cases. [Online]. Available: https://wiki.onosproject.org

[19] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1868447.1868466

[20] B. Chung, C. Chen, C.-C. Tseng, J. H. Chen, and J. Mambretti, "P4mt: Designing and evaluating multi-tenant services for p4 switches," in *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2021, pp. 267–272.

[21] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, 1997.